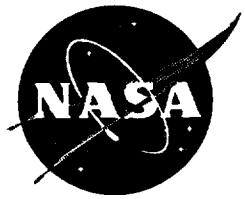
CR-189411

SOFTWARE ENGINEERING LABORATORY SERIES

SEL-94-006

PROCEEDINGS OF THE NINETEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

DECEMBER 1994



National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

Proceedings of the Nineteenth Annual Software Engineering Workshop

November 30–December 1, 1994

GODDARD SPACE FLIGHT CENTER

Greenbelt, Maryland

FOREWORD

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

NASA/GSFC, Software Engineering Branch

The University of Maryland, Department of Computer Science

Computer Sciences Corporation, Software Engineering Operation

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

The SEL is accessible on the World Wide Web at

http://groucho.gsfc.nasa.gov/Code_550/SEL_hp.html

Single copies of this document may be obtained by writing to:

Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

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

CONTENTS

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

CONTENTS (cont'd)

Session 1: The Software Engineering Laboratory

Changes and Challenges in the Software Engineering Laboratory
Rose Pajerski, NASA/Goddard

Domain Analysis for the Reuse of Software Development Experiences
Vic Basili, University of Maryland

Building an Experience Factory for Maintenance
Jon Valett, NASA/Goddard

Closing the Loop on Improvement: Packaging Experience in the Software Engineering Laboratory
Sharon Waligora, Computer Sciences Corporation

Changes and Challenges in the Software Engineering Laboratory

Rose Pajerski

Software Engineering Branch, Code 552
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771

Background

Since 1976, the Software Engineering Laboratory (SEL) has been dedicated to understanding and improving the way in which one NASA organization, the Flight Dynamics Division (FDD), develops, maintains, and manages complex flight dynamics systems. The SEL is composed of three member organizations: NASA/GSFC, the University of Maryland, and Computer Sciences Corporation. During the past 18 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.

Software Improvement Approach

The SEL's basic approach toward software process improvement is to first understand and characterize the process and product as they exist to establish a local baseline. Only then can new technologies be introduced and assessed (phase two) with regard to both process changes and product impacts. Typically, several studies/assessments are in progress at any one time, each with a duration of approximately 1–3 years. The third phase of the SEL approach (packaging) synthesizes the results of the first two phases and feeds them back into the cycle to assist software development on subsequent projects. Packages include products such as process tailoring guidelines, training courses, tools, and guidebooks. The SEL's process improvement approach has proven very effective in the FDD, with the organization's software product showing substantial improvements in error rates, reuse, and cycle time; and it has been recognized throughout the software engineering community. In 1994, the SEL received *the IEEE Computer Society Award for Software Process Achievement* and a *Federal Technology Leadership Award* for its application of these innovative concepts in a production environment.

SEL Operational Changes

The SEL's development and maintenance environments differ somewhat in their characteristics. On development efforts, the languages and processors used reflect a

movement toward workstation-based systems: languages are FORTRAN (70%), Ada (15%), and C (15%) spread over 65% mainframe systems and 35% workstation systems. Maintenance efforts are dominated by FORTRAN with 85% usage, followed by Ada (10%), with the remaining systems implemented in a variety of languages, such as assembler and Pascal. Platforms of systems under maintenance are predominantly mainframes (80%), with the remaining 20% maintained on workstation-based processors.

Based in part on these environmental factors, and on recent plans for changes in this environment, several significant paradigm shifts occurred in the SEL's operation. This has led to changes in three areas:

- Organizational goals
- Operations and development environments
- Resources

Change #1: Organizational Goals

From its inception, the SEL has focused on both increasing software reliability and reducing life cycle costs. Over the past 8 years, the SEL has achieved measured gains in both areas: reliability of delivered systems has increased threefold and current mission support costs are half that of older systems. However, with "time to deploy" pressure increasing, SEL goals now emphasize development time as well as cost. In response to this, the FDD, with the SEL's support, is expanding development of high-reuse, generalized systems to encompass more flight dynamics application areas. The SEL is investigating a variety of joint team development processes as well as cataloguing and assessing existing maintenance processes to identify potential time-savers.

Change #2: Operations and Development Environments

The change here—the transition from mainframes to workstations—has already been discussed. In support of this trend, the SEL provides historical data on completed system rehosting activities for management planning of subsequent efforts. Data collection and measurement activities are also being revisited to determine whether these procedures must be modified. In addition, new computer-aided software engineering (CASE) tools are being investigated for use on the available workstations.

Change #3: Resources

From 1989 through 1994, resources increased by about 10% per year, enabling the SEL to undertake several NASA-wide initiatives: developing guidebooks and assessment reports on specialty topics such as measurement, NASA-wide software characteristics, domain identification, and technology transfer activities. These experience exchanges facilitated the spread of SEL concepts not only throughout NASA, but beyond, to other government organizations and industry. However, resources for 1995 have been significantly reduced, prompting a reevaluation of both internal and external efforts. The SEL has decided to

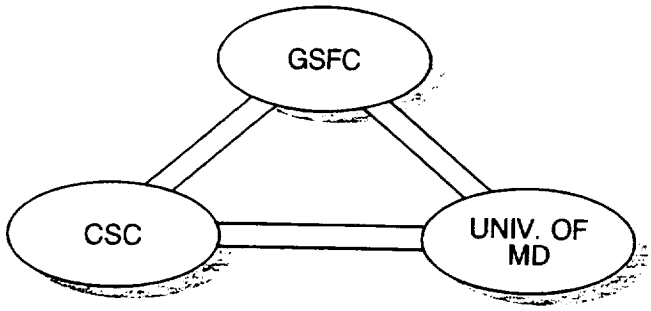
focus external outreach efforts on similar domains within NASA and to investigate new processes likely to provide direct cost benefits in the FDD production environment.

Impact and Observations

Given the above changes, what are the lessons?

- The first experience lesson is that new process technologies must be integrated within the existing process framework. The SEL approach of understanding, assessing, and packaging is effective at instilling large, as well as small, process changes because it yields a fundamental understanding of process and product.
- Next, the move to workstations will create a tighter link between process tools, measurement, and process analysis. This should assist SEL analysts in providing more timely feedback to development groups.
- Last, the importance of understanding software domains has been reemphasized in the SEL's work. An ability to compare and contrast domains is critical for technology transfer and tailoring guidance activities.

SOFTWARE ENGINEERING LABORATORY (SEL)



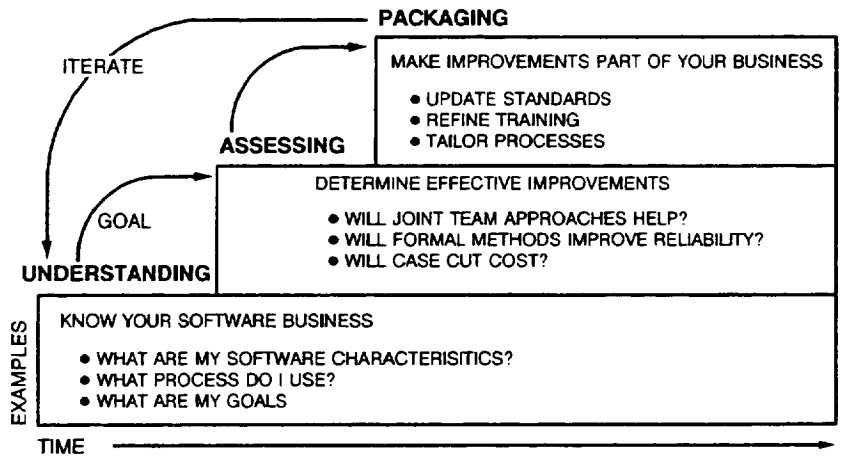
C649.003

THE SEL FROM 1976 - 1994

- GOALS
  - UNDERSTAND THE SOFTWARE PROCESS IN A PRODUCTION ENVIRONMENT
  - DETERMINE IMPACT OF AVAILABLE TECHNOLGIES
  - INFUSE IDENTIFIED/REDEFINED METHODS INTO DEVELOPMENT PROCESS
- APPROACH
  - APPLY TECHNOLOGIES AND EXTRACT DETAILED DATA IN PRODUCTION ENVIRONMENT (EXPERIMENT)
  - MEASURE IMPACT (COST, QUALITY, DEVELOPMENT TIME,...)
  - PACKAGE RESULTS (STANDARDS, PROCESSES,TRAINING...)

C649.004

SEL PROCESS IMPROVEMENT APPROACH



C649.005

SEL PRODUCTION ENVIRONMENT

SOFTWARE CHARACTERISTICS: SCIENTIFIC, GROUND BASED, INTERACTIVE

	DEVELOPMENT	MAINTENANCE
LANGUAGE	70% FORTRAN 15% Ada 15% C	85% FORTRAN 10% Ada 5% OTHER
PROCESSORS	65% MAINFRAME 35% WORKSTATION	80% MAINFRAME 20% WORKSTATION
	PER PROJECT:	PER RELEASE:
DURATION	12-30 MONTHS	3-12 MONTHS
EFFORT	10-25 STAFF YEARS	1-5 STAFF YEARS
SIZE	100K-300K SLOC	WIDE VARIATION

C649.006

1994 -- 1995 CHALLENGES

- ORGANIZATIONAL GOALS
- OPERATIONS AND DEVELOPMENT ENVIRONMENTS
- RESOURCES

C649.007

CHANGE #1: ORGANIZATIONAL GOALS

PREVIOUS EMPHASIS ON RELIABILITY & COST ⇨ CURRENT EMPHASIS ON COST & TIME TO DELIVER

- SEL RESPONSE
  - EXPAND OBJECT-ORIENTED, GENERALIZED DEVELOPMENT TO OTHER APPLICATIONS
  - ASSESS JOINT DEVELOPMENT PROCESSES WITHIN CURRENT SEL METHODOLOGY
  - UNDERSTAND MAINTENANCE PROCESS/PRODUCT

INTEGRATE NEW PROCESS TECHNOLOGIES WITHIN EXISTING FRAMEWORK

C649.008

CHANGE #2: OPERATIONS AND DEVLOPMENT ENVIRONMENTS

MAINFRAME APPLICATIONS ⇨ WORKSTATION APPLICATIONS

- SEL RESPONSE
  - PROVIDE MANAGEMENT SUPPORT FOR "REHOST VS NEW" DECISIONS
  - REVISIT MEASURES AND DATA COLLECTION MECHANISMS
  - EXPAND COMPUTER-AIDED SOFTWARE ENGINEERING (CASE) TECHNOLOGY STUDY

POTENTIAL EXISTS FOR GREATER INTEGRATION OF SEL ANALYSIS WITH DEVELOPMENT

C649.009

CHANGE #3: RESOURCES

GROWING SUPPORT FOR EXTERNAL OUTREACH (RESOURCES INCREASING) ⇨ FOCUS ON INTERNAL NEEDS (RESOURCES DECREASING)

- SEL RESPONSE
  - USE OUR DETAILED PROCESS UNDERSTANDING TO SELECT TECHNOLOGIES LIKELY TO IMPACT COST
  - CONTINUE DOMAIN IDENTIFICATION EFFORTS TO FIND "LIKE" DOMAINS FOR EXPERIENCE EXCHANGES
  - PROMOTE "SEL-APPROACH" TO GSFC/NASA AREAS (PRIMARY) AND OTHER ORGANIZATIONS

DOMAIN IS A KEY DRIVER FOR SEL EXTERNAL OUTREACH EFFORTS

C649.010

Domain Analysis for the Reuse of Software Development Experiences[1]

V. R. Basili*, L. C. Briand**, W. M. Thomas*

* Department of Computer Science
University of Maryland
College Park, MD, 20742
USA

** CRIM
1801 McGill College Avenue
Montreal (Quebec), H3A 2N4

1. Introduction

We need to be able to learn from past experiences so we can improve our software processes and products. The Experience Factory is an organizational structure designed to support and encourage the effective reuse of software experiences [Bas94]. This structure consists of two organizations which separates project development concerns from organizational concerns of experience packaging and learning. The experience factory provides the processes and support for analyzing, packaging and improving the organization's stored experience. The project organization is structured to reuse this stored experience in its development efforts. However, a number of questions arise:

- What past experiences are relevant?
- Can they all be used (reused) on our current project?
- How do we take advantage of what has been learned in other parts of the organization?
- How do we take advantage of experience in the world-at-large?
- Can someone else's best practices be used in our organization with confidence?

This paper describes approaches to help answer these questions. We propose both quantitative and qualitative approaches for effectively reusing software development experiences.

2. A Framework for Comprehensive Software Reuse

The ability to improve is based upon our ability to build representative models of the software in our own organization. All experiences (processes, products, and other forms of knowledge) can be modeled, packaged, and reused. However, an organization's software

[1] This research was in part supported by NASA grant NSG-5123, NSF grant 01-5-24845, and CRIM

experience models cannot necessarily be used by another organization with different characteristics. For example, a particular cost model may work very well for small projects, but not well at all for large projects. Such a model would still be useful to an organization that develops both small and large projects, even though it could not be used on the organization's large projects. To build a model useful for our current project, we must use the experiences drawn from a representative set of projects similar to the characteristics of the current project.

The Quality Improvement Paradigm (QIP)[Bas85,Bas94] is an approach for improving the software process and product that is based upon measurement, packaging, and reuse of recorded experience. As such, the QIP represents an improvement process that builds models or packages of our past experiences and allows us to reuse those models/packages by recognizing when these models are based upon similar contexts, e.g., project and environmental characteristics, relative to our current project. Briefly, the six steps of the QIP are:

1) Characterize the current project and environment
2) Set goals for successful performance and improvement
3) Choose processes and methods appropriate for the project
4) Execute the processes and the measurement plan to provide real-time feedback for corrective action
5) Analyze the data to assess current practices, determine problem areas, and make recommendations for future projects
6) Package the experience in a form suitable for reuse on subsequent projects

The QIP must allow the packaging of context-specific experience. That is, both the retrieval (Steps 2 and 3) and storage (Step 6) steps need to provide the identification of the context in which the experience is useful. Step 1 helps determine this context for the particular project under study. When a project selects models, processes, etc., (step 3), it must ensure that the chosen experience is suitable for the project. Thus the experience packaging (step 6) must include information to allow future projects to determine whether the experience is relevant for use in their context. The problem is that it is not always an easy task to determine which experience is relevant to which project contexts. We would like to reuse the packaged experience if the new project context is "similar" to the projects from which the experience was obtained.

Basili and Rombach present a comprehensive framework for reuse-oriented software development [BR91]. Their model for reuse-oriented software development is shown in Figure 1. It includes a development process model, which is aimed at project development, and a reuse process model, which enables the reuse of the organization's experiences. The development process will identify a particular need (e.g., a cost model). The reuse process model can then find candidate reusable artifacts (e.g., candidate cost models) from the experience base, selecting (and adapting if necessary) the one most suitable for the particular project. The artifacts may have originated internally (i.e., was developed entirely from past projects in the organization, e.g., the meta-model approach to cost estimation [BB81]), or externally (e.g., the COCOMO cost model [Boe81]). In any event, a evaluation activity of the reuse process is what determines how well the candidate artifact meets the needs specified by the development process.

Our focus in this paper is on the selection and evaluation of reusable artifacts. One approach to do so is to determine "domains" in which a particular experience package may

be reusable. Then, by assessing the extent to which a project is a member of the domain, one can determine whether a given experience package is suitable for that project.

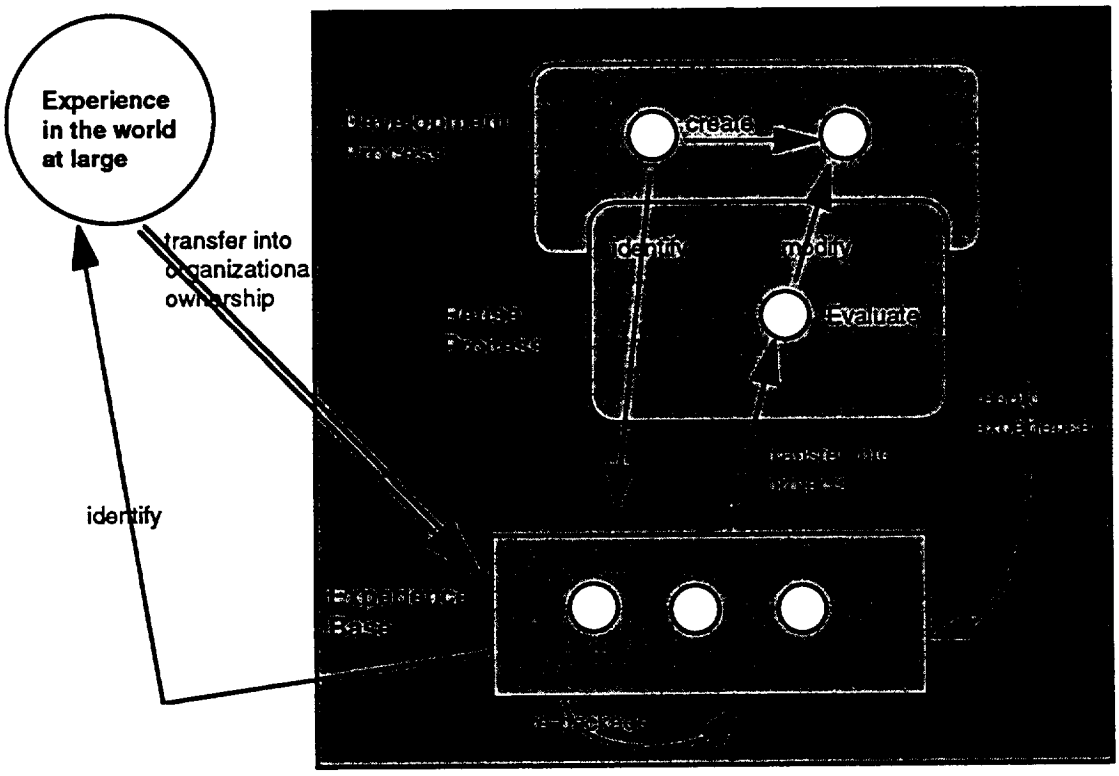


Figure 1: Reuse Oriented Software Development model

3. Experience Domain Analysis

We will use *experience domain analysis* to refer to identifying domains for which reuse of project experiences can be effective, i.e., identifying types of projects for which:

- Similar development or maintenance standards may be applied, e.g., systems for which DoD-Std-2167 is applicable in NASA
- Data and models for cost, schedule, and quality are comparable, e.g., projects for which the productivity can meaningfully be compared within all the branches of NASA

Once domains have been identified, common processes, standards and databases may be shared with confidence by various software organizations, e.g., NASA branches within broader organizational structures such as NASA centers. The problem can be viewed as the

need to determine whether an experience package developed in one context is likely to be effective when reused in a new, different context.

Let us introduce a basic set of definitions to clarify the concepts to be used:

Definition 1: Domain Analysis Goals

> The goal(s) of a domain analysis procedure is (are) to assess the feasibility of reusing or sharing a set of software artifacts within or across organizations, e.g., can we use a cost model developed in another organization, based on a different set of past projects than those of our development organization?

Definition 2: Domains

> With respect to a particular domain analysis goal (i.e., a particular artifact to reuse), *domains* are "types" of software development projects among which certain common artifacts may be reused or shared.

Definition 3: Domain Characteristics and Characterization Functions

> *Domain characteristics* represent characteristics that may determine whether or not one or several artifacts can be reused or shared within or across organizations. The *characterization functions* of domains are mappings of projects, described by characteristics, into domains. For example, in a given organization, large Ada flight simulators may represent a domain with respect to cost modeling and prediction. In this case, the project characteristics involved in the domain characterization function are the project size, the programming language, and the application domain.

It is important to note that for different reuse goals, there are different domains, and as such, different domain characteristics and characterization functions. In this context, the first step of domain analysis is to determine the kind(s) of artifacts one wants to reuse. As an example, someone may want to reuse a cost model or a design inspection procedure developed in a different development environment.

We present in Table 1 a general taxonomy of software artifacts that can conceivably be reused (or shared) within or across organizations. It is important to note that the taxonomy presented here encompasses more than just software products (the usual realm of domain analysis). Also, one could further refine the taxonomy within each specific organization.

Certain kinds of artifacts are more likely to be reusable or sharable than others because they naturally have a broader realm of application. For example, many high-level concepts are universally applicable, e.g., tracking project progress across the development life cycle through data collection helps monitor the schedule and resource consumption of the system being developed. Other kinds of artifacts may have a somewhat more restricted realm of application, e.g., a waterfall process model is applicable as long as the application domain is well known and the solution space is reasonably understood. Also, some kinds of artifact have a very narrow realm of application, e.g., artifacts related to application domain specific programming languages and operating systems (e.g., a real-time UNIX variant for real-time applications).

Data/Models	Standards/Processes	Products
Descriptive models	Requirements	Requirements
Predictive models	Specifications	Specifications
Cost models	Design	Architecture
Schedule models	Coding	Design
Reliability growth models	Testing	Code
Error models	Inspections	Test plans/data
Change models	Change management	
Quality evaluation models		
Lessons learned		
Raw data		

Table 1: Taxonomy of Reusable Software Artifacts

After establishing organizational goals for domain analysis, it is necessary to examine the characteristics of the organization and the projects to help identify appropriate domains. For example, if one wants to reuse a cost model for a new project, the following questions become relevant:

- Are the products developed by the new project comparable to the ones on which the cost model is based ?
- Is the development process similar?
- If not, are the differences taken into account in the model?
- Can the model be imported to this new environment?

The issue is now to determine if the new project belongs to the "same domain" as the projects on which the cost model is built . In order to do so, the characteristics of these projects need to be considered and differences across projects need to be analyzed. Similarly, if one tries to reuse a design inspection process, the following questions are relevant: is staff training sufficient on the new project? Do budget and time constraints allow for the use of such inspections? Is the inspection process thorough enough for the new project reliability requirements?

Table 2 shows a general taxonomy of potentially relevant project characteristics to consider when reusing or sharing software artifacts. The characteristics are grouped according in three broad classes, product, process, and personnel. The table is not intended to be a complete description of all relevant project characteristics, but rather its intent is to provide some guidance as to the characteristics that should be considered. In some environments certain characteristics may not be relevant, and others that are not currently in this table will be. Each organization needs to determine which ones are the most important.

Product	Process	Personnel
Requirements stability	Lifecycle/Process model	Motivation
Concurrent Software	Process conformance	Education
Memory constraints	Project environment	Experience/Training:
Size	Schedule constraints	• Application Domain
User interface complexity	Budget constraints	• Process
Programming language(s)	Productivity	• Platform
Safety/Reliability		• Language
Lifetime requirements		Dev. team organization
Product quality		
Product reliability		

Table 2: Potential Characteristics Affecting Reuse

In the following sections we discuss two techniques to support experience domain analysis, one based on quantitative historical data, and the other based on qualitative expert opinion. Both techniques will provide models that can be used to assess the risk of reusing a particular experience package in a new context.

4. A Quantitative Approach

With sufficient historical data, one can use automated techniques to partition the set of project contexts into domains relative to various levels of reuse effectiveness for a given experience package. The goal of partitioning here is to maximize the internal consistency of each partition with respect to the level of effectiveness. In other words, we want to group projects according to common characteristics that have a visible and significant impact on effectiveness.

Such partition algorithms are based on the multivariate analysis of historical data describing the contexts in which the experience package was applied and the effectiveness of its use. For example, Classification Trees [SP88] or Optimized Set Reduction [BBH93, BBT92] are possible techniques. The measured effectiveness is the dependent variable and the explanatory variables of the domain prediction model are derived from the collection of project characteristics (Table 2) possibly influencing the reuse of the experience package.

As an example, suppose you want to know whether you can use the inspection methodology *I* on project *P*. The necessary steps for answering that question are as follows:

- Determine potentially relevant characteristics (or factors) for reusing *I*, e.g., project size, personnel training, specification formality, etc.
- Determine the measure of effectiveness to be used, e.g., assume the rate of error detection as the measure of effectiveness of *I*. This is used as the dependent variable by the modeling techniques mentioned above.
- Characterize *P* in terms of the relevant characteristics, e.g., the project is large and the team had extensive training.

- Gather data from the experience base characterizing past experience with *I* in terms of the project characteristics and the actual effectiveness. For instance, we may gather data about past inspections that have used methodology *I*.
- Construct a domain prediction model with respect to *I* based on data gathered in the previous step. Then, one can use the model to determine to which domain (i.e., partition) project *P* belongs. The expected effectiveness of *I* on project *P* is computed based on the specific domain effectiveness distribution.

Table 3 shows, for each past project with experience with *I* (A, B, C, D, etc.), the recorded effectiveness of *I* on the project, and a collection of characteristics relevant to the effective reuse of *I* (Size, the amount of training, the formality of the specifications). The last row in the table shows the question that need to be answered for the new project *P*: How effective is *I* likely to be if it is applied to project *P*?

Project	Det. Rate	KSLOC	Training	Formality
A	.60	35	Low	Informal
B	.80	10	High	Formal
C	.50	150	Medium	Informal
D	.75	40	High	Formal
...				
P	*??*	*20-30*	*High*	*Formal*

Table 3: Examples for a Quantitative Approach

From such a table, an example of domain characterization that might be constructed by performing a partition of the set of past projects (more formally: a partition of the space defined by the three project characteristics):

$$(\text{KSLOC} < 50)\ \&\ (\text{Training=High}) \Rightarrow \mu(\text{Detection Rate}) = 75\%$$

This logical implication indicates that for projects with less than 50 KSLOC and a high level of training, the detection rate is expected to be 75 percent.

When a decision boundary is to be used (e.g., if the level of reuse effectiveness is above 75% of the I's original effectiveness, one reuses I), an alternative form of model is more adequate:

$$(\text{KSLOC} < 50)\ \&\ (\text{Training=High}) \Rightarrow \text{Probability}(\text{Detection Rate} > 75\%) = 0.9$$

Here the logical implication indicates that for relatively small projects (less than 50 KSLOC) where there is a high level of training, the detection rate with *I* is likely to be greater than 75 percent.

To evaluate the reusability of some experience packages, a quantitative approach to domain analysis is feasible, mathematically tractable, and automatable. Unfortunately, there are a number of practical limitations to such an approach. It is not likely to be effective when:

- There is not sufficient data on past experience
- The effectiveness of reuse is not easily measured
- There is significant uncertainty in the characterization of the new project.

In these cases we may wish to resort to a more heuristic, expert opinion based approach. This is the topic of the next section.

5. A Qualitative Approach

Given the practical limitations to the quantitative approach, a qualitative solution based on expert opinion is needed. As with the quantitative solution, the basic assumption in this approach is that an experience package will be reusable (with similar effectiveness) in a new context if the new context is "similar" to the old context. Rather than defining "similarity" through analysis of historical data, the approach here is to capture and package expert opinion as to what makes an artifact "similar." As we previously noted, what the notion of similarity depends upon the reuse objective. For example, two projects may have widely different cost characteristics (e.g., in the COCOMO classification, one being organic and one embedded) but have very similar error characteristics.

When identifying domains without the support of objective data, one can use several ordinal evaluation scales to determine to what extent a particular characteristic is relevant and usable given a specific reuse goal. These ordinal scales help the analyst introduce some rigor into the way the domain identification is conducted. We propose several evaluation scales that appear to be of importance. The first one determines how relevant is a project characteristic to the use of a given artifact, e.g., is requirements instability a characteristic that can affect the usability of a cost model? A second scale captures the extent to which a characteristic is measurable, e.g., can we measure requirement instability? Other scales can be defined to capture the sensitivity of the metric capturing a domain characteristic, i.e., whether or not a significant variation of the characteristic always translates into a significant variation of the metric, and the accuracy of the information collected about the domain characteristics.

These evaluation scales should be used in this order: (1) determine whether the project characteristic is relevant, (2) assess how well it can be measured, and (3) determine how sensitive and accurate measurement is. If a characteristic is not relevant, then one need not be concerned with whether and how it can be measured. We will define only a relevancy evaluation scale and a measurability evaluation scale here. The two other scales are more sophisticated and beyond the scope of this document.

The relevance scale is intended to indicate the degree to which a characteristic is important with respect to the effective reuse of a particular artifact. The following ordinal scale can be used for this purpose:

1: Not relevant, the characteristic should not affect the use of the artifact of interest in any way, e.g., application domain should not have any effect on the use of code inspections.

2: Relevant only under unusual circumstances, e.g., application domain specific programming language generate the need for application domain specific coding standards whereas, in the general case, the characteristic *application domain* does not usually affect the usability of coding standards.

3: It is clearly a relevant characteristic *but* the artifact of interest can, to some extent, be adjusted so it can be used despite differences in the value. For example, size has an effect on the use of a cost model, i.e., very large projects show lower productivity. Assume that an organization occasionally developing large scale projects wants to reuse the cost model of an organization developing mostly medium-size projects. In this case, the cost model may be calibrated for very large projects. As another example, consider the Cleanroom process [D92]. If not

enough failure data are available during the test phase, the Cleanroom process may be used without its reliability modeling part.

4: It is clearly a relevant characteristic *and* if a project does not have the right characteristic value, the use of the considered artifact is likely to be inefficient. For example, it may not be cost-effective to use specification standards that require formal specifications in a straightforward data processing application domain. Moreover, the artifact is likely to be difficult to tailor.

5: It is clearly a relevant characteristic *and* if a project does not have the right characteristic value, the use of the considered artifact is likely to generate a major failure. For example, if the developed system requires real-time responses to events occurring in its operational environment, requirement analysis and design approaches from a non real-time development environment cannot be used. Moreover, the artifact is likely to be very difficult to tailor.

There are other metrics that are of interest to one interested in the reusability of an artifact. Some of the characteristics may be quite relevant but very difficult to measure. As such, there may be increased risk in the assessment of the reusability of an artifact due to the uncertainty (due to the difficulty in measurement) in the characterization. A measurability scale for the characteristics can be defined as follows:

1: There is no known measure of the characteristic, e.g., development team motivation and morale are hard to measure.

2: There are only *indirect* measures of the characteristic available , e.g., state transition diagram of a user interface can be measured to offer an approximate measure of user interface complexity.

3: There are one or more *direct* measures of the characteristic, e.g., size can be measured on a ratio scale, programming language on a nominal scale.

Two other issues have to be considered when a project characteristic is to be used to differentiate domains. First, we cannot ensure that every significant variation of the characteristic is going to be captured by the measurement, i.e., that the metric is sensitive enough. As a consequence, it may be hard in some cases to tell whether or not two projects are actually in the same domain. Second, a metric may be inherently inaccurate in capturing a characteristic due to its associated data collection process. These two issues should always be considered.

Table 4 shows an example in which we assess the relevance of a subset of the characteristics listed in the taxonomy of domain analysis characteristics, i.e., the product characteristics. Their relevance is considered for the reuse of testing standards, i.e., standards, processes, and procedures defining unit, system, and acceptance test.

Product Characteristic	Relevance Score

Unstable Requirements	3
Concurrent Software	4
Memory Constraints	3
User Interface Complexity	4
Reliability/Safety Requirements	5
Long Lifetime Requirements	3
Product Size	4
Programming Language	1
Intermediate Product Quality	4
Product Reliability	3

Table 4: Product Characteristic Relevancy Scores for the Reuse of Testing Standards

The following paragraphs provide some justification for the relevancy scores assigned for the reuse of testing standards. Detailed justification and explanations about scores for other artifacts will be provided in a subsequent report.

- Unstable or partially undetermined requirements:

  There should be a degree of stability achieved by the start of testing. However, unstable requirements will have a great impact on test planning, and if there are many changes occurring during the test phase, testing will be impacted. Some ways to avoid some of these problems is to have more rigorous inspections, focusing on identifying inconsistencies, and to carefully partition testing activities so that a somewhat stable base is verified, and the impact of the instability is lessened. Also, with the large number of changes late in development, better support for regression test is required. A score of 3 is assigned.

- Concurrent Software:

  In the presence of RT constraints, in addition to verifying functional correctness, it is also needed to verify the necessary performance characteristics of critical threads. These performance requirements must be validated early to allow more options in rectifying problems and to lessen the chance of cost and budget overruns if the requirements are not being met. Also, it would be inefficient for a project that does have such constraints to apply a process that includes early identification and testing of critical threads. A score of 4 is assigned.

- Memory constraints:

  As with real-time constraints, problems in meeting memory constraints should be identified early. However, it is typically easier to verify memory use than

performance characteristics. It is likely that a standard could be adapted to provide for an earlier verification of the critical memory use. A score of 3 is assigned.

- User interface complexity:

  A user interface with a large number of states requires significant verification effort. Certain techniques that are well suited to a smaller number of states (e.g., test every operation in every state) do not scale well to applications with large, complex interfaces. A score of 4 is assigned.

- High Reliability /Safety Requirements:

  In safety-critical software, correctness of the implementation is a primary concern. Approaches such as formal verification, fault-tree analyses, and extensive testing are often necessary. A score of 5 is assigned.

- Long Lifetime Requirements:

  Testing should be more comprehensive for long-lived software. The goal is not only to ensure current operational suitability, but to allow for operation long after development. With the expectation of a number of changes over the product lifetime, it becomes much more important that the delivered product be thoroughly tested. For example, to ensure a better test coverage, procedures can be put in place to require testing of all paths. A score of 3 is assigned.

- Size:

  Certain techniques that are well suited to small applications do not scale well to large applications. More automation and support is likely to be needed. For example, exhaustive path-testing may be useful in smaller applications, but in large applications it is not feasible due to the significant resources that would be required. A score of 4 is assigned.

- Programming Language:

  No impact.

- Product quality:

  A lesser quality product may be subjected to additional verification procedures so as to ensure a consistent level of quality prior to beginning a certain test activity. This additional verification may not be as cost-effective for products that are known to be of very high quality. For example, applying the same procedures designed for verification of new code to reused software (known to be of high quality) is likely to be less cost-effective. A score of 4 is assigned.

- Reliability:

  Testing an unreliable product is a difficult task, as the unreliability may result in a number of changes to correct errors late in development. If one knows that a lower quality product is to be expected (through modeling or comparison with other similar projects), procedures can be used to lessen their impact. For example, more rigorous inspection procedures can be used, targeting the types of defects expected to in the product. Also, additional support for regression testing can be used to help re-integrate changed modules into a particular build. A score of 3 is assigned.

The table can be used in the following manner. For the experience package (e.g., testing standard) of interest, one would examine the table to find which characteristics are of particular importance. Then information about the context of use that characterizes the reusable package in terms of these important characteristics should be obtained. The current

project must also be characterized in the same way. These two characterizations can be compared, and a subjective assessment of the risk of reusing the artifact in the context of the new project can be made. The higher the score for a given characteristic on the relevancy evaluation scale, the higher the risk of failure if project sharing or reusing artifacts do not belong to the same domain, i.e., do not show similar values or do not belong to identical categories with respect to a given relevant characteristic. In situations where the risk is high, the reuse/sharing of artifacts will require careful risk management and monitoring [B88]. Sometimes, in order to alleviate the likelihood of failure, the shared/reused artifacts will have to be adapted and modified. Also, if projects appear to belong to the same domain based on an indirect measure (see measurability scale) of the project characteristics, risk can increase due to the resulting uncertainty.

The following example illustrates the approach. The IBM Cleanroom method was considered by NASA/GSFC, code 550, for developing satellite flight dynamics software. In the following paragraph, we give examples of characteristics (and their scores according to the characteristic evaluation scale) that were actually considered before reusing Cleanroom.

- First, it was determined that not enough failure data (Reliability characteristic in Table 2) were produced in this environment in order to build the reliability growth models required by the Cleanroom method. As a consequence, reliability estimates based on operational profiles could not be used to build such models. So Reliability gets a relevancy evaluation metric of 3 and a measurability evaluation metric score of 3.
- There was, despite intensive training, a lack of confidence in the innovative technologies involved in the Cleanroom method, in particular, regarding the elimination of unit test (Personnel Motivation in Table 2: relevancy evaluation score of 3, measurability evaluation score of 1). Therefore, once again, the process was modified: unit test would be allowed if the developer felt it was really necessary and requested it. Interestingly, after gaining experience with the method, it was found that unit test was not being requested, so this change was later removed. Also, there were doubts about the capability of the Cleanroom method to scale up to large projects. As such, the technique was first used on a small scale project (Product Size in Table 2: relevancy evaluation score of 3, measurability evaluation score of 3).
- On the other hand, the use of FORTRAN (versus COBOL in IBM) was not considered as an issue (Programming language in Table 2: relevancy evaluation score of 1, measurability evaluation score of 3).

Once tables such as those shown in Table 4 have been defined for all reuse goals of interest in a given organization, they can be used to help assess whether a software artifact can be reused. For example, suppose that one wants to reuse design standards from other projects on which they appeared to be particularly successful. The relevancy table for design standards may tell us that characteristics such as size and programming language are not very relevant, but that the level of concurrency and real-time in the system are extremely important characteristics to consider.

Suppose that the set of projects where these design standards were assessed as effective can be described as follows: stable requirements, heavy real-time and concurrent software, no specifically tight memory constraints, and very long lifetime requirements (i.e., long term maintenance). If the project(s) where these standards are to be reused present some differences with respect to some of these important project characteristics, it is likely that the design standards will require some level of tailoring in order to be reusable, if reusable

at all. For example, tight memory constraints would generate the need to include in the design standards some strategies to minimize the amount of memory used, e.g., standard procedures to desallocate dynamic memory as soon as possible.

There are a number of weaknesses to this qualitative approach. Perhaps the most important is that it does not adequately express the influence of a factor in a particular context, or, in other words, the interactions between factors. For example, suppose a particular factor, such as tight memory constraints, has an impact on the reusability of a testing methods only in a particular context (e.g., large-scale projects). The table could tell us that tight memory constraints is an important characteristic, but it would not convey the information about the specific context in which it is important. In addition, the table does not quantify the factor's influence on a ratio scale.

6. Conclusions

A wide variety of software experiences are available for reuse within and across most organizations. These experiences may be of local validity (e.g., an error model from NASA/GSFC Code 550), meaningful in a large organization (NASA), or of some value across several development environments (e.g., the COCOMO cost model). However, it is not always clear to what extent the experience package may be reused on a given project. In this paper we described experience domain analysis as an approach to solve this problem. We described two distinct approaches, one quantitative and one qualitative.

The quantitative approach is feasible; however, there are likely to be practical limitations to the approach, primarily due to the difficulty in obtaining sufficient and adequate historical data. The qualitative approach appears more practical; however, it has some drawbacks that may limit its effectiveness. We are working towards a solution that will combine the formality of the quantitative approach with the subjective aspects of qualitative expert opinion. Ideally, we could express rules, derived from expert opinion, which describe the reusability of a package in much the same format as the patterns of the quantitative approach. We are investigating the use of expert systems and fuzzy logic as a means for capturing and representing expert opinion in such a format. Some of the issues with such an approach being addressed include:

- How to acquire expertise?
- How to formalize and package the expert opinion so that it is potentially reusable by other people?
- How to provide a means for dealing with the inherent uncertainty in the expert knowledge?
- How can we check the consistency and completeness of the acquired knowledge?
- How can we combine several expert opinions?

7. References

[B88] B. W. Boehm, Software Risk Management, Prentice–Hall, 1988.

[Bas94] V. Basili et al., "The Experience Factory", Encyclopedia of Software Engineering, Wiley&Sons, Inc., 1994

[BB81] J. W. Bailey and V. R. Basili, "A Meta-model for Software Development Resource Expenditures, *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, 1981.

[Boe81] B. W. Boehm, Software Engineering Economics, Prentice–Hall, 1981.

[BR91] V. R. Basili and H. D. Rombach, "Support for Comprehensive Reuse," *Software Engineering Journal*, 6 (5), September, 1991.

[Bas85] V. Basili, "Quantitative Evaluation of Software Methodology", *Proceedings of the First Pan-Pacific Computer Conference*, Australia, July 1985.

[BR88] V. Basili and H. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments", *IEEE Trans. Software Eng.*, 14 (6), June, 1988.

[BBH93] L. Briand, V. Basili and C. Hetmanski, "Developing Interpretable Models with Optimized Set Reduction for Identifying High-Risk Software components", *IEEE Trans. Software Eng.*, 19 (11), November, 1993.

[BBT92] L. Briand, V. Basili and W. Thomas, "A Pattern Recognition Approach for Software Engineering Data Analysis", *IEEE Trans. Software Eng.*, 18 (11), November, 1992.

[D92] M. Dyer, The Cleanroom Approach to Quality Software Development, Wiley, 1992.

[SP88] R. Selby and A. Porter, "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis", *IEEE Trans. Software Eng.*, 14 (12), December, 1988.

EXPERIENCE DOMAIN ANALYSIS FOR SOFTWARE REUSE

Victor R. Basili, Lionel Briand, Bill Thomas

Institute for Advanced Computer Studies
Department of Computer Science,
University of Maryland, College Park, Maryland

Presented at the Nineteenth Annual Software Engineering Workshop
November 30 - December 1, 1994

REUSING EXPERIENCES

The Problem

We need to be able to learn from past experiences so we can improve our software processes and products

BUT

What past experiences are relevant?

Can they all be used (reused) on our current project?

How do we take advantage of what has been learned in other parts of the organization?

How do we take advantage of experience in the world-at-large?

Can someone else's best practices be used in our organization with confidence?

REUSING EXPERIENCES

The Quality Improvement Paradigm

The Quality Improvement Paradigm (QIP) represents an improvement process that **builds models/packages** of our past experiences and allows us to reuse those models/packages by recognizing when these **models are based upon similar contexts**, e.g., project and environmental characteristics, **to our current project**

For example:

- We can use a cost model with confidence when it has been generated by projects with similar characteristics
- We can use a method with confidence when it has been effective on similar projects

REUSING EXPERIENCE

QIP Assumptions

The ability to improve is based upon our ability to **build representative models** of the software in our own organization

All experiences (**processes, products**, and **other forms of knowledge**) can be modeled, packaged, and reused

An organization's software **experience models cannot necessarily be used** by another organization with different characteristics

To build a usable model, we must**find a representative set of projects** similar to the characteristics of our current project

REUSING EXPERIENCES

Potentially Reusable Experiences

Data/Models

Descriptive Models	Predictive Models
(cost, schedule, reliability, error, change models)	
Quality evaluation Models	Lessons Learned
Raw Data	

Standards/Processes

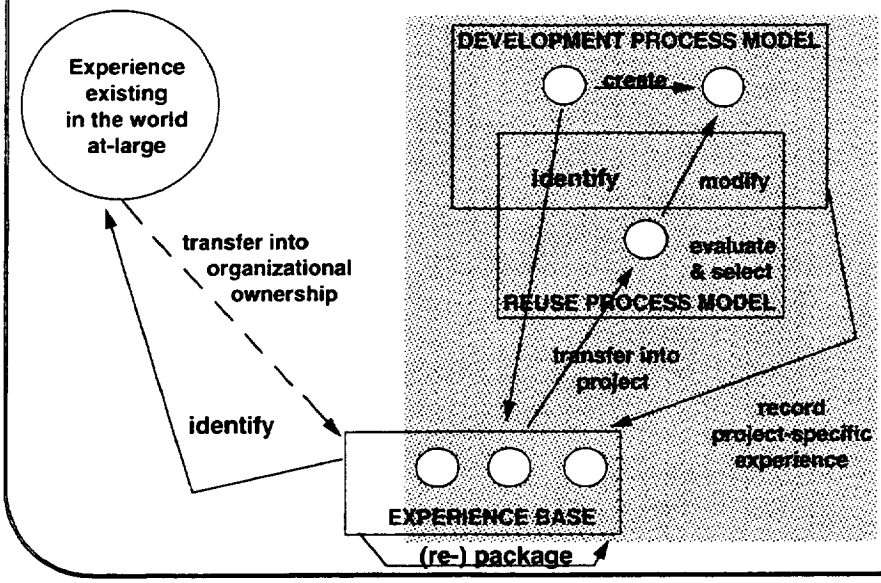
Requirements	Specifications
Design	Coding
Testing	Inspections
Change Management	

Products

Requirements	Specifications
Architecture	Design
Code	Test Plans/Data

REUSING EXPERIENCES

Comprehensive Reuse Development Model



REUSING EXPERIENCES

Toward a Solution

We need a mechanism to help us **recognize groups of similar projects** with respect to the experience we want to use (reuse)

We need to **identify different software experience domains**

Projects may be identified as "similar" to other projects if we can **learn from them with respect to the reuse of a given experience** with confidence

E.g., I might be able to use data from other experiences bases to build a cost model if I can select the set of projects in that experience base with similar characteristics to my own project

We need to develop a set of rules, based upon project characteristics, that **define software domains for particular experience packages**

EXPERIENCE DOMAIN ANALYSIS

Potential Project Factors Affecting Reuse

- **Product**
  - Requirements stability
  - Memory constraints
  - User interface complexity
  - Safety/Reliability requirements
  - Intermediate product quality
  - Concurrent Software
  - Size
  - Programming languages(s)
  - Lifetime requirements
  - Product reliability
- **Process**
  - Lifecycle/Process model
  - Project environment
  - Budget constraints
  - Process conformance
  - Schedule constraints
  - Productivity
- **Personnel**
  - Motivation
  - Education
  - Experience/training
    (Application domain, Platform, Process)
  - Development team organization

EXPERIENCE DOMAIN ANALYSIS

Definition

We use **experience domain analysis** to refer to identifying areas for the reuse of experience, i.e., identifying groups of systems for which:

- **similar development or maintenance standards may be applied**, e.g., systems for which DoD-Std-2167 is applicable in NASA
- **data and models for cost, schedule, and quality are comparable,** e.g., systems for which the productivity can meaningfully be compared within all the branches of NASA

Once domains have been identified, common processes, standards and databases may be shared with confidence

- by various software organizations, e.g., NASA branches
- within broader organizational structures, e.g., NASA centers

EXPERIENCE DOMAIN ANALYSIS

Restating the Problem

Problem:

- Determine whether an experience package developed in one context is likely to be effective when reused in a new, different context

Potential Solutions:

- A quantitative approach
- A qualitative/heuristic approach
- Formalizing the heuristic approach

EXPERIENCE DOMAIN ANALYSIS

A Quantitative Approach

Problem:

- Develop a **quantitative approach** for determining whether an experience package, **ep**, developed in one context is likely to be effective when reused in a new, different context

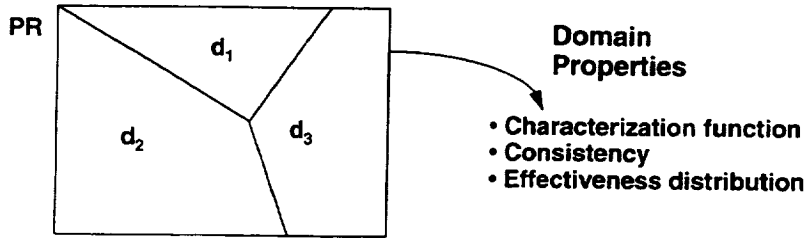
A Solution:

- Partition the set of **project contexts** into domains relative to the **effective use** of an experience package, **ep**. This partition forms a **domain model** with respect to the experience package that is to be reused
- Construct domain models (using multivariate analysis of historical data describing the contexts in which the **ep** was applied). The measured **effective use** in those contexts is the dependent variable in the multivariate analysis

EXPERIENCE DOMAIN ANALYSIS

A Quantitative Approach

- For an experience package **ep**, partition the set of projects **PR** into domains $D=\{d_i\}$ relative to the **effective use** measure for **ep**
- For new project **P** and each domain d_i, calculate the expected effectiveness of **ep** for project **P** as a function of the
  - probability that **P** belongs to d_i
  - consistency of the domain d_i
  - recorded effectiveness distributions in the domains



EXPERIENCE DOMAIN ANALYSIS

An Example

Suppose you want to know whether you can use the existing inspection methodology on project **P**

Determine potential relevant factors

e.g., project size, personnel training, specification formality, etc.

Determine the learning criterion

e.g., assume the **rate of error detection** as the measure of effective use (the learning criteria)

Characterize P in terms of the relevant factors, and gather data from the experience base characterizing past experience with inspections in terms of the factors and the actual effectiveness

(Con't)

EXPERIENCE DOMAIN ANALYSIS

Example

Run the learning algorithm to determine the expected effective use of inspections on project **P**; use an algorithm with interpretable models that makes no assumptions on the form of the model, e.g., OSR

Project	Det. rate	KSLOC	Training	Formality ...
A	.60	35	Low	informal
B	.80	10	High	formal
C	.50	150	Medium	informal
D	.75	40	High	formal
...				
P	*??*	*20-30*	*High*	*formal*

Example Pattern:
(KSLOC < 50) & (Training=High) => Det. Rate > .70

EXPERIENCE DOMAIN ANALYSIS

Limitations to Quantitative Analysis

To evaluate the reusability of some experience packages, a quantitative approach to domain analysis is

- feasible
- mathematically tractable
- automatable

However, it is not likely to be effective when

- There is **not sufficient data** on past experience
- The effective use of an **ep** is **not easily measured**
- There is **significant uncertainty** in the characterization of the new project

In these cases we may wish to resort to a more heuristic, expert opinion based approach

EXPERIENCE DOMAIN ANALYSIS

A Qualitative Approach

Problem:

- Develop a **qualitative approach** for determining whether an experience package, **ep**, developed in one context is likely to be effective when reused in a new, different context

Solution:

- Define some **subjective metrics** that will help experts determine to what extent a particular characteristic is **relevant** and **usable** when reusing a particular experience package, based upon their opinion and experience.
- Build a table to associate the effect of these factors with various experience packages

EXPERIENCE DOMAIN ANALYSIS

Relevance Evaluation

Relevance: How relevant is the factor to the use of a particular experience package, e.g., is requirements instability a factor that can affect the usability of a cost model?

1. **Not relevant**, the factor should not affect the use of the ep in any way, e.g., application domain should not affect use of inspections
2. **Relevant only under unusual circumstances**, e.g., application domain doesn't usually affect the use of coding standards, except when using an application domain specific programming language
3. **Relevant factor but the ep can be adjusted**, so it can be used despite value differences, e.g., size has an effect on the use of a cost model
4. **Relevant factor and if a project does not have the right factor value, the use of the ep is likely to be inefficient**, e.g., not cost effective to use specification standards that require formal specifications in a simple data processing problem
5. **Relevant factor and if a project does not have the right factor value, the use of the ep is likely to generate a major failure**, e.g., if the project is real-time, requirements and design approaches from a non-real time environment cannot be used

EXPERIENCE DOMAIN ANALYSIS

A Qualitative Approach

Relevance Metric: How relevant is the factor to the use of a particular experience package?

Based upon the ordinal scale, the **higher the score** for a given factor on the relevancy evaluation scale, **the higher the risk** of failure if projects reusing experience packages do not belong to the same domain

Other Sample Subjective Metrics:

Measurability: To what extent is the factor measurable , e.g., can we measure requirements instability?

Sensitivity: Does a significant variation in the factor always translate into a significant variation of the metric

Accuracy: What is the accuracy of the information collected about the factor

EXPERIENCE DOMAIN ANALYSIS

Formalizing the Qualitative Approach

Problems:

- How to acquire expertise?
- How to formalize and package the expert opinion so that it is potentially reusable by other people?
- How to provide a means for dealing with the inherent uncertainty in the expert knowledge. There are three types of uncertainty:
- How can we check the consistency and completeness of the acquired knowledge?
- How can we combine several expert opinions?

Solution:

We are studying an approach based upon expert systems and fuzzy logic to try to answer these questions

EXPERIENCE DOMAIN ANALYSIS

Conclusion

We believe **experience domain analysis** is a fundamental problem in software engineering, especially as related to learning and improvement, as expressed in the Quality Improvement Paradigm

We are working on ways to perform experience domain analysis so that software domains may be defined

not solely on the bases of local organizational (EF) partitioning

but according to the factors that characterize development processes, technologies, products, constraints, goals, and risks associated with the projects

If organizations can effectively share data, lessons learned and best practice information, they can improve faster and further than they could in isolation

Building an Experience Factory for Maintenance

Jon D. Valett

Software Engineering Branch
NASA Goddard Space Flight Center
Greenbelt, Maryland 20771

Steven E. Condon

Computer Sciences Corporation
10110 Aerospace Road
Lanham-Seabrook, Maryland 20706

Lionel Briand, Yong-Mi Kim, Victor R. Basili

Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

This paper reports the preliminary results of a study of the software maintenance process in the Flight Dynamics Division (FDD) of the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC). This study is being conducted by the Software Engineering Laboratory (SEL), a research organization sponsored by the Software Engineering Branch of the FDD, which investigates the effectiveness of software engineering technologies when applied to the development of applications software.

This software maintenance study began in October 1993 and is being conducted using the Quality Improvement Paradigm (QIP), a process improvement strategy based on three iterative steps: understanding, assessing, and packaging. The preliminary results presented in this paper represent the outcome of the understanding phase, during which SEL researchers characterized the maintenance environment, product, and process.

Findings indicate that a combination of quantitative and qualitative analysis is effective for studying the software maintenance process; that additional measures should be collected for maintenance (as opposed to new development); and that characteristics such as effort, error rate, and productivity are best considered on a "release" basis rather than on a project basis. The research thus far has documented some basic differences between new development and software maintenance. It lays the foundation for further application of the QIP to investigate means of improving the maintenance process and product in the FDD.

Introduction

Goddard Space Flight Center (GSFC) manages and controls NASA's Earth-orbiting scientific satellites and also supports Space Shuttle flights. For fulfilling both these complex missions, the Flight Dynamics Division (FDD) developed and now maintains over 100 different software systems, ranging in size from 10 thousand source lines of code (KSLOC) to 250 KSLOC, and totaling 4.5 million SLOC. Of these systems, 85% are written in FORTRAN, 10% in Ada, and 5% in other languages. Most of the systems run on IBM mainframe computers, but 10% run on PCs or UNIX workstations.

The Software Engineering Laboratory (SEL) has been researching and experimenting in the FDD since 1976 with the goal of understanding the software development process in this environment; measuring the effect of software engineering methodologies, tools, and models on this process; and identifying and applying successful practices (Reference 1). The SEL has developed an approach to process improvement known as the Quality

Improvement Paradigm (QIP) and has established a supporting organizational structure, the Experience Factory, for maintaining the experience base, which is a key element of this work. These concepts, and their application specifically in this study of software maintenance are described in detail in Sections 1 and 2 of this paper.

One of the key features of this research is the combination of qualitative and quantitative approaches used to characterize the current practice of software maintenance in the FDD. These methods affected the design of the experience base developed for the study, by influencing which maintenance products and projects would be examined and which specific measures would be collected. The structure of the study is described in Section 3. Sections 4 and 5, respectively, elaborate on the qualitative analysis of the maintenance process and the quantitative analysis of the product and process characteristics. Section 6 discusses lessons learned and early recommendations for process improvement, and Section 7 poses questions that will guide future direction for this research.

1. The Quality Improvement Paradigm

The QIP is a three-step iterative process that provides an organization with a framework for continuously improving its methods of doing business. These steps–*understanding*, *assessing*, *packaging*–are shown in Figure 1.

The QIP begins with *understanding*, because before an organization can begin planning for improvement, it must thoroughly understand its current processes, products, and environmental characteristics. At the current time, the FDD maintenance study is completing its first pass through this step.

During the second phase of the maintenance study, corresponding with the *assessing* step of the QIP, improvement goals will be set, experiments conducted, and their results assessed. The experiments will test new methods or tools that show promise of helping this organization achieve its improvement goals. If these experiments demonstrate significant improvements in the process or products, these lessons will be incorporated into the overall FDD organization.

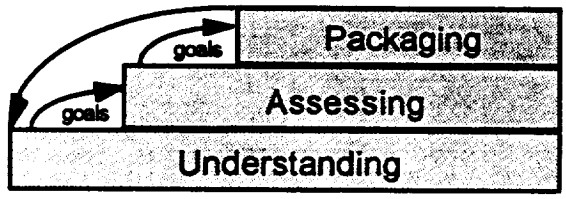


Figure 1. Quality Improvement Paradigm

This third and final phase of the QIP, the *packaging* step, requires significant investment to truly capitalize on the time and money spent in the understanding and assessing steps. It may require developing new standards as well as implementing and fielding comprehensive training in these new standards.

After completing the packaging step, researchers will baseline the new process by returning to the understanding step, to verify the positive effect of process evolution on the system. Thus begins a new iteration of the QIP.

1.1 The QIP and Software Development Projects

The QIP has been used many times within the SEL to investigate the potential of new tools or processes on software development projects. In its more detailed application, the QIP consists of six steps (Reference 2):

1. Characterize the current project and its environment with respect to models and measures. Begin by characterizing the development project relative to the environment. What kind of product is being developed? How large is the project? What is the schedule? How is the project similar to and different from previous projects? This is used to provide models of similar experiences from similar projects.
2. Set quantifiable goals for successful project performance and improvement. Is the goal to shorten cycle time, reduce errors, achieve higher software reuse?

3. Choose an appropriate process model and supporting methods and tools for this project. Choose processes for the project that show promise of achieving the stated goals based upon past experience with projects of this type. Identify projects with similar characteristics and similar goals.
4. Execute the processes, construct the products, collect and validate the prescribed data, and analyze them to provide real-time feedback for corrective action.
5. Analyze the data to evaluate the current practices, determine problems, record findings, and make recommendations for future project improvements.
6. Package the experience as updated and refined models and other forms of structured knowledge gained from this project and prior projects. Save it in an experience base to be reused on future projects.

1.2 The QIP and Software Maintenance

For maintenance, the implementation of the QIP is slightly different, because past releases of the same project provide additional experience. The underscored phrases below indicate maintenance-specific foci of the QIP.

1. Characterize the current project release and proposed set of modifications and its environment.
2. Set quantifiable goals for successful project performance and improvement and the future evolution of this product. Remember that this release will soon be followed by another release and yet another release.
3. Choose an appropriate process model and supporting methods and tools for this project based on both domain class and specific product knowledge. When studying maintenance, there is an advantage over applying the QIP to new development projects because knowledge and experience are available about this specific product.
4. Execute the processes, construct the products, collect and validate the prescribed data, and analyze them to provide real-time feedback for corrective action, including real-time preventive maintenance on the current project.
5. Analyze the data to evaluate the current practices and their effects on this product. Characterize the current product, determine problems, record findings, and make recommendations for this product and future project improvements.
6. Package the experience as updated and refined models and other forms of structured knowledge gained from this project and prior projects. Save it in an experience base for future projects and the evolution of this product.

2. The Experience Factory

The SEL researchers and database team act as an *experience factory* for the software developers in the FDD (Reference 3). The experience factory organization is separate from the project organization. It serves the project organization by analyzing and synthesizing knowledge into models that support the improvement of software development (see Figure 2). It does so by concentrating on the analysis and packaging activities of the QIP, while the project organization focuses on developing the software. The project organization supplies process and product data to the experience factory and carries out experiments under the guidance of the experience factory team. The experience factory collects and analyzes the data from the project organization. It stores these data and analyses in an experience database. It also packages the best of these experiences into products, guidelines, and models, which it feeds back to the project organization to help improve its process.

The experience factory for maintenance operates the same as the experience factory for development, with three differences: First, the experience factory for maintenance

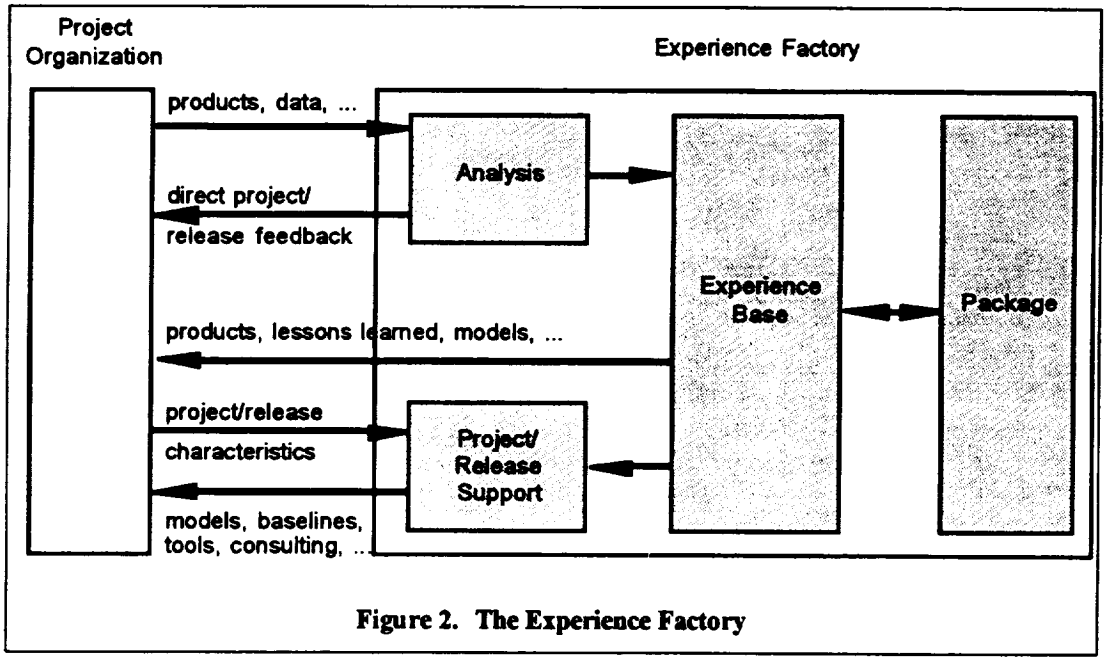


Figure 2. The Experience Factory

must address releases. Second, analysis for release feedback requires quicker response; development life cycles are on the order of 18–24 months, whereas maintenance release cycles are on the order of 6 months. Third, software maintenance emphasizes product evolution more than software development does, so experience includes past experience on the same project.

3. Building the Experience Base for Software Maintenance

Because there are many similarities between software development and software maintenance, the SEL experience of software development was used as a starting point for understanding maintenance. The measurement program for maintenance was modeled on the measurement program that is used for understanding software development. This influenced both the goals that were set and also the specific data that were identified for collection. To characterize the process, data were collected on maintenance effort distribution by activity, similar to the measures collected for new development, with some tailoring for maintenance-specific activities. To characterize the products, data were collected on a number of measures, including the amount of code modified for a release and the number of errors introduced by the maintenance work. The specific measures are discussed in more detail below.

The study team consisted of a team leader from NASA, three researchers from the University of Maryland, and one researcher from Computer Sciences Corporation. The team leader drew up the initial study plan containing the overall goals, the specific questions to be answered, and the list of maintenance measures to be collected for analysis. Data were collected on eleven maintenance projects. In addition, researchers closely monitored four of these projects and stayed in close contact with the maintenance teams on those projects. The entire study team met regularly throughout the study to refine the study plan and assess progress. These meetings also resulted in some revisions to the collected measures.

Following the lead of Lionel Briand, one of the University of Maryland researchers, a general qualitative analysis methodology was adopted, tailored, and applied to the four closely monitored maintenance projects (Reference 4). This methodology provided an objective but qualitative project characterization that complemented the quantitative

characterization that was provided by the measurement data. By supplying the researchers with a characterization of the organization structures, processes, issues, and risks of the maintenance environment, the qualitative analysis also helped them refine the data collection measures. In return, the quantitative data helped researchers to understand the qualitative data. This qualitative analysis methodology also provided a process for determining the causal links between maintenance problems, on the one hand, and flaws in the maintenance process or maintenance organization, on the other hand. The following two sections describe the combined qualitative and quantitative approach in detail.

4. Six-Step Process to Qualitative Understanding

The qualitative analysis methodology consisted of six steps, depicted in Figure 3. Researchers accomplished each step by reviewing release documents and process description documents, and also by interviewing maintenance team members.

Steps 1 through 3 provided an understanding of the maintenance organization and the release process followed by the project. With this information for several projects, researchers were able to draw comparisons between projects and to check each project for adherence to maintenance policies. Steps 4 through 6 provided the mechanism for identifying where problems existed for each project and for demonstrating flaws in the maintenance organization or the maintenance process (as followed by the project).

4.1 Understanding Steps (1-3)

Step 1 called for identifying the organizational entities involved in the maintenance process. Researchers identified distinct teams, their roles, and the information flows among these teams. For example, for each project, release approval passed from the configuration control board to the maintenance team.

In Step 2, researchers identified the phases of the release process and the major milestones that bounded these phases. For example, the *change analysis phase* culminated in the Release Contents Review meeting, and the *solution analysis & design phase* culminated in the Release Design Review meeting.

Step 3 required identifying the activities involved in each phase. Researchers selected a list of generic maintenance activities and

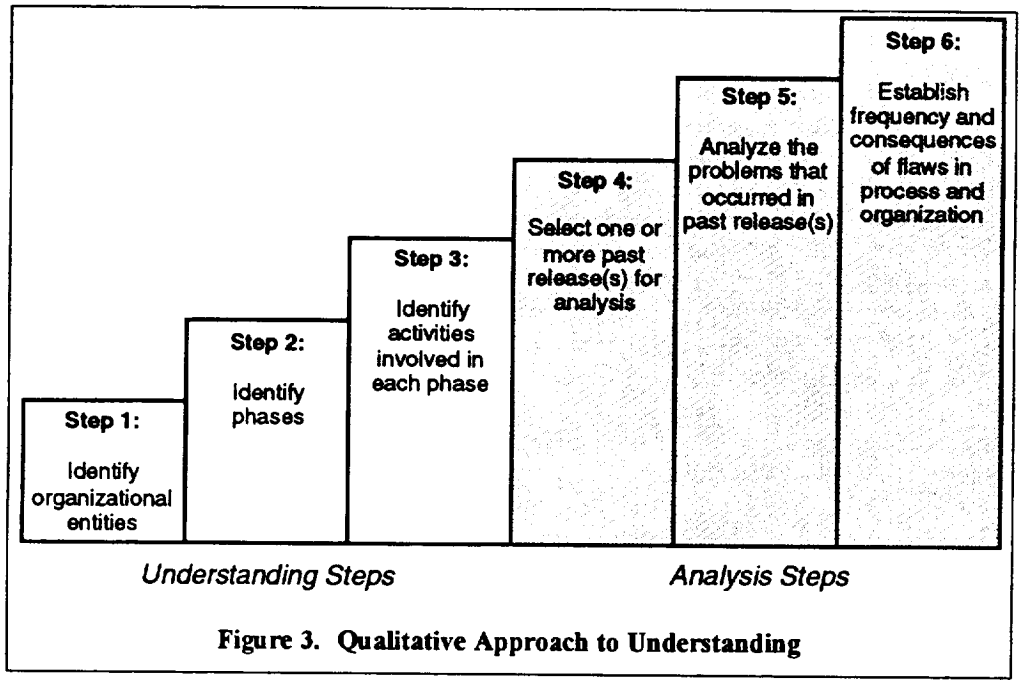


Figure 3. Qualitative Approach to Understanding

mapped them into the various phases identified in Step 2. In Step 3, researchers also identified the inputs and outputs for each phase. For example, in one project, the *solution analysis & design phase* activities included release scheduling and planning, understanding the requirements of changes, changing the designs, some coding, and some quality assurance. Inputs included the Release Contents Review document; offline discussions among maintainers, users, analysts, and testers; and answers to formal questions submitted to analysts. The outputs included the preliminary designs, test plans, prototypes, release schedule, and size estimates.

4.2 Analysis Steps (4-6)

In Step 4, researchers chose a previous software maintenance release for analysis. Researchers took care to select a recent release, so that the studied release reflected the current process, and so that complete release documentation was available. This choice also made it more likely that the technical lead from the release would be accessible for interviews.

In Step 5, researchers studied the release documentation and interviewed the appropriate parties to define and analyze the problems encountered in developing this release. For each software change request in the release, researchers determined the size of the change, assessed the relative difficulty of the change, and identified any errors or delays that resulted from implementing this change request. If errors or delays resulted from this work, researchers then attempted to determine the maintenance process flaws (if any) that caused these. For example, in one project, a change request for a major enhancement resulted in 11 subsequent errors, substantial rework, and up to 1 month of lost effort on the release. The errors stemmed initially from incomplete or ambiguous change requirements written by the users. The maintainers designed the enhancement based on these written requirements. The fact that the requirements were deficient and that design nevertheless proceeded on the enhancement, was judged by researchers to represent a maintenance process flaw. The effect of this flaw, however, was then compounded by a subsequent lack of communication between the users and maintainers. The users neglected to attend the Release Contents Review and then voiced no objections to the design presented by the maintainers at the Release Design Review. When later, at the Release Acceptance Test Readiness Review, the users finally objected to the implementation of the enhancement, much time had been lost. This lack of communication revealed either an unclear definition of release responsibilities or a lack of adherence to the defined responsibilities.

In Step 6, researchers assessed the frequency and the consequences of flaws in the maintenance process and organization as provided by the data gathered in Step 5, and made recommendations for improvements to the process. For this study, the analysis led to three recommendations: 1) provide guidelines for content and format of change requests; 2) explicitly define the content of documents and review materials; 3) enforce stricter adherence to the maintenance process, especially attendance at review meetings and review/approval of designs.

5. Quantitative Approach to Understanding

In past studies of development projects, tracking the developers' estimates of effort, product size, and schedule has been useful, so similar data were collected for maintenance releases. For maintenance, however, the schedule milestones are somewhat different from development. Thus data were collected on effort hours between release start, release contents review, release design review, release acceptance test readiness review, and release operational readiness review. Researchers monitored and attempted to model the effort that programmers, testers, and managers expend on a maintenance release by breaking the effort down into types of software activity, such as coding, documenting, regression testing, and acceptance testing. Additional activities specific to (or more prominent in) maintenance were included, such as impact analysis, cost benefit analysis, and error isolation time.

The purpose of the quantitative approach was to define and collect those measurements that would most meaningfully characterize the maintenance process and products. Analysis of these data should establish a baseline model of the current maintenance process that answers the following questions:

1. What is the distribution of effort among software activities during maintenance?
2. What are the characteristics of a maintenance release?
3. What are the characteristics of maintenance errors?
4. What are the error rates and change rates?

To achieve the maintenance study goal and to answer these specific questions, the following data were collected:

1. Effort by activity (i.e., impact analysis/cost benefit analysis, isolation, change design, code/unit test, inspection/certification/consulting, integration test, acceptance test, regression test, system documentation, user/other documentation, other hours)
2. Effort by type of maintenance change (i.e., adaptation, error correction, enhancement)
3. Error and change data
   - Time spent (i.e., effort to isolate, effort to fix)
   - Source of error (i.e., previous change, code, design, requirements, other)
   - Class of error (i.e., initialization, logic, external interface, internal interface, computational, or other)
4. Release estimates and actuals (i.e., schedule, effort, number of lines of code, number of modules)
5. Size of software under maintenance (lines of code)

In January 1994, the SEL began collecting data on the eleven target maintenance projects. A new software release estimates form was created and introduced at this time. Two existing data collection forms (a weekly effort form and a software change request form) had already been in use for some time within the organization, and were already being used by three of the eleven target projects. These two existing forms continued to be collected, but now were required for all eleven target projects. In August 1994, following completion of some of the qualitative analysis and after discussions with a wider circle of maintainers, the weekly effort form was revised to capture effort by release and by change request instead of merely by project. The software activities list also was broadened. The preliminary results of the quantitative data analysis are summarized below.

5.1 Maintenance Effort

The average distribution of maintenance effort by activities is presented in Figure 4. The activities (listed above) have been grouped into four categories (design, implementation, test, other). This figure represents the overall distribution based on total effort expended on the eleven maintenance projects from January through October 1994. It includes both entire release cycles and some partial release cycles. This distribution is dominated by the six busiest projects, which contributed 93% of the hours used in the calculation of Figure 4. The distributions for the individual projects vary significantly from each other and also from this average distribution. When more data are available for complete release cycles, there may be some reduction in the variability of this distribution among projects.

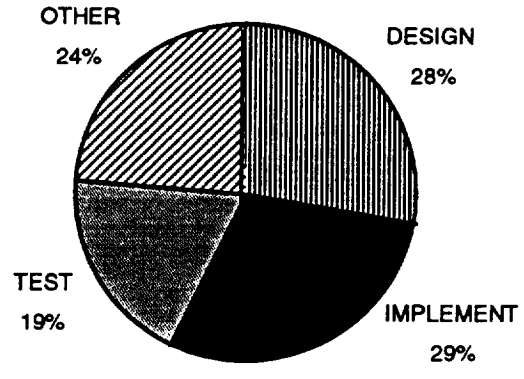


Figure 4. Maintenance Effort Distribution

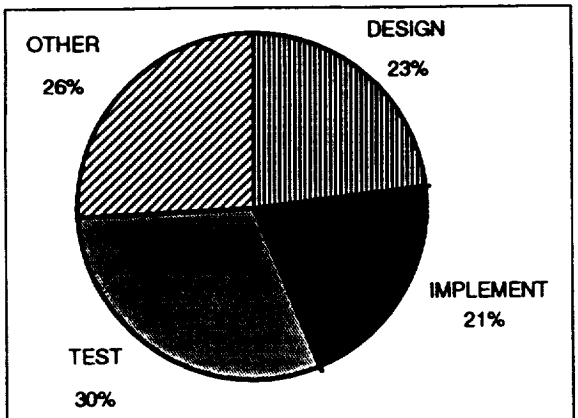


Figure 5. Development Effort Distribution

The distribution of effort during the original development was not available for many of these projects. Figure 5, however, presents the distribution of effort for the original software development of eleven fairly typical projects from this environment.

As illustrated by these two figures, design and code (implement) activity constitute a larger percentage of effort during maintenance than during software development (57% versus 44%). This contrast reinforces the belief that design and implementation are more costly in maintenance than in development. There are many possible reasons for this, for example, the difficulty in isolating errors and the relatively large overhead required to make small code changes. One might expect that this cost increase would be more pronounced for error corrections than for enhancements, because adding major enhancements is more like doing new development work. The data in the next section support this hypothesis, showing greater productivity for enhancements than for error corrections.

5.2 Release Characteristics

When programmers, testers, and managers reported their time spent on maintenance effort each week, they recorded their hours by software activities. Prior to mid-August, when weekly effort collection forms were revised, they also classified their hours by the type of change requests on which they worked (i.e., adaptation, error correction, or enhancement) and *other* hours (e.g., management, meetings). This provided researchers insight into the distribution of types of changes requested and the amount of effort each type requires.

Figure 6 presents the average distribution of effort hours by type of change. These data represent all the effort data for the eleven target maintenance projects from January to mid-August 1994. It includes both entire release cycles and some partial release cycles. This distribution is again dominated by the same six busiest projects, which contributed 93% of the hours used in the calculation in Figure 6. The distributions for the individual projects vary significantly from each other and also from this average distribution. For example, effort spent on enhancements varied from 51% to 89% among the six dominant projects.

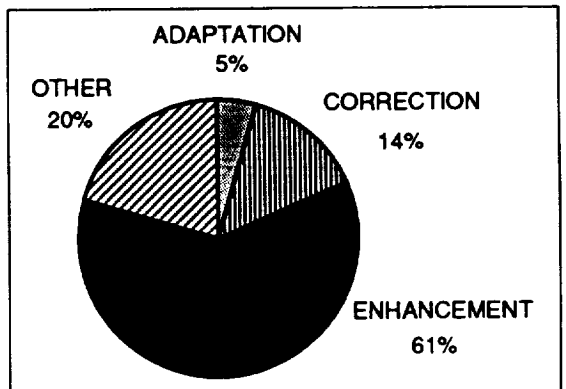


Figure 6. Effort Distribution by Type of Change

Figure 7 presents the distribution of change requests by type. The data are limited to completed releases from the last 2 years for which complete change request data were available. This amounted to nine releases containing 83 change requests (4 adaptations, 37 enhancements, 42 error corrections). Only five of the eleven maintenance projects under study are represented. As more data from complete releases become available, this distribution may change. Again there was much

variability. The percentage of changes that were enhancements in a release varied from 20% to 83%, excluding one release that consisted entirely of error corrections.

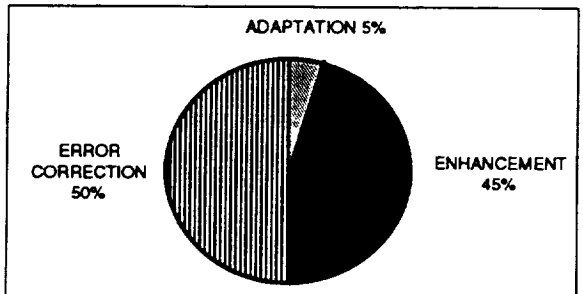


Figure 7. Changes By Type

These last two figures demonstrate that in the FDD enhancements typically are larger than error corrections and require more effort to implement. This is shown by the fact that although the number of enhancements was slightly smaller than the number of error corrections (45% versus 50%), the ratio of effort spent on enhancements to effort spent on error corrections was 4.3:1.

The difference in size is even more dramatic than the difference in effort. The 37 enhancements in these nine releases accounted for 96.6% of the lines of code added, changed, or deleted, whereas the 42 error corrections accounted for only 3.1%, for a ratio of 31:1. By comparing the size ratio (31:1) to the effort ratio (4.3:1), the productivity (lines of code added, changed, or deleted per hour) is about seven times greater for enhancements than it is for error corrections.

5.3 Error Characteristics

The 83 change requests described above represent the *original content* of these nine releases. These are all requests to change the operational version of the software; in this paper, these changes are referred to as *operationally indigenous* changes. During the implementation of each release, however, some errors usually are introduced by the maintenance work. If these errors are caught by the testers, they in turn generate additional change requests which usually become part of the same release delivery. These latter changes are termed *release indigenous* changes. In this study, an attempt was made to separate these two categories of changes. (The effort distribution in Figure 5, however, includes effort on both operationally indigenous and release indigenous change requests. Revised data collection since mid-August will allow effort to be separated by change request.)

The next two figures demonstrate the sources of the errors in these nine releases, both operationally indigenous and release indigenous. The 83 operationally indigenous changes included 42 error corrections (see Figure 8). Note that requirement specification, code, and design each represent a significant portion of the source of errors, 20% to 35% each. These nine releases also included 29 release indigenous change requests, all of which were error corrections (see Figure 9).

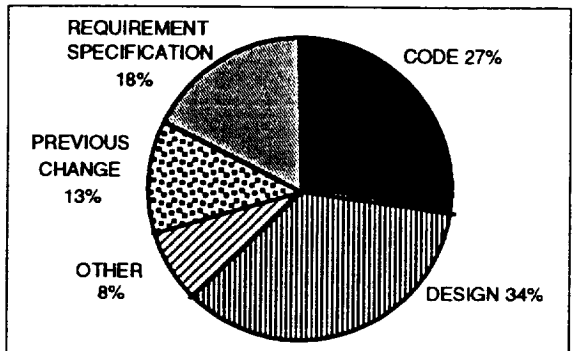


Figure 8. Operational Errors

Note that requirement specification and design represent much smaller portions of the release indigenous errors than of the operationally indigenous errors. Previous change is somewhat higher, and coding is much higher, for release indigenous errors. The distribution of errors found in release testing is similar to the distribution of errors found during acceptance testing of new development projects. This similarity suggests that release testing and development acceptance testing both

uncover similar kinds of errors with similar degrees of success. On the other hand, software operations seem to uncover a different distribution of errors, suggesting that operations are more effective than these testing processes at uncovering certain types of errors, such as design errors, for example. More study is needed to explain why testing and operations should have such different error detection distributions.

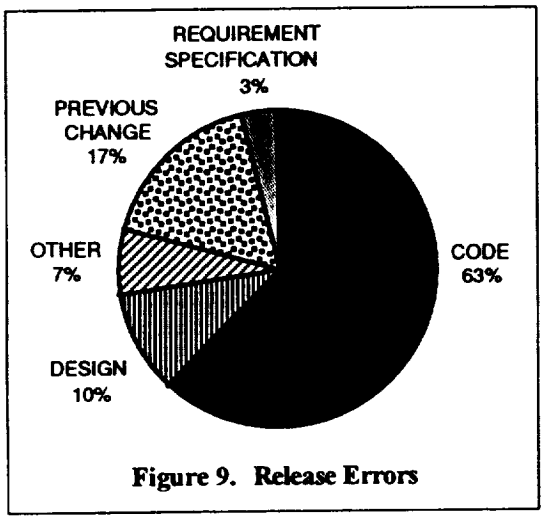


Figure 9. Release Errors

5.4 Error and Change Rates

When the error rate was analyzed for operationally indigenous errors, errors were normalized by both the size of the project (SLOC) and the time period during which they were detected. This adjustment was made for the following reasons: It was expected that, all other things being equal, a larger piece of software would tend to have more errors than a smaller piece of software, so errors/SLOC would be a more meaningful measure of software quality than raw errors. It was also suspected that, all other things being equal, the piece of software that had been exercised operationally for a longer time probably would have more errors uncovered. When comparing error rates for many projects, this dual normalization resulted in more uniform error rates across projects, more so than when either normalization was done separately, or when no normalization was performed at all.

Error rate data were available for ten of the eleven projects in this study, reaching back 2 years for most projects. Analysis of the error rates for these ten projects over the last 2 years (less than 2 years for some of the newer projects) resulted in a mean value of 11 errors per 100 KSLOC per year (minimum 5, maximum 32). Project size ranged from 42 to 263 KSLOC.

Release indigenous errors are those that are introduced by the maintenance process. It was expected that the more code that was modified in a release, the more errors were likely to be introduced. Therefore release indigenous errors were normalized by the modified KSLOC in the original content of the release. *Modified KSLOC* is the sum of KSLOC added, changed, and deleted. For the nine maintenance releases mentioned above, the mean error rate for release indigenous errors was 0.8 errors per modified KSLOC (minimum 0, maximum 6.9). Correcting the release indigenous errors required more lines of code to be added, changed, or deleted before delivering the release. The overall ratio of this additional modified code to the original modified code for the nine was 2.5% [25 additional modified SLOC (minimum 0, maximum 172) per original modified KSLOC].

6. Lessons Learned

This study demonstrated the importance of closely consulting with the software project personnel (here maintainers) when carrying out any software development study. Both the researchers and the maintainers benefited by the close working relationship on this study. The researchers gained a better understanding of the difficulties and peculiarities of the maintenance process; the maintainers gained some insights into the difficulties of the data definition, collection, and analysis process that leads to useful models.

The qualitative analysis that was done for four of the maintenance projects in this study helped ensure that the maintainers were intimately involved in the baselining process. This analysis also helped the researchers to rethink and to begin to redefine the measurement program. For example, weekly personnel effort data is now grouped by release and

by software change, instead of merely by project. Researchers have also redefined and expanded the list of software activities to which maintainers apportion their effort. In addition, the qualitative analysis has suggested the usefulness of reexamining error taxonomies, which the study team hopes to address at a later date.

As the researchers studied the release process, it became evident that there was a need to differentiate between those errors that were *operationally indigenous* and those errors that were *release indigenous*. One obvious reason was that reduction of release indigenous errors is an important improvement goal for maintenance. A second reason is that each of these error sets has something important to say about the maintenance process. In trying to resolve operationally indigenous errors (and adaptations and enhancements), maintainers sometimes introduce release indigenous errors. When such errors are introduced, both the original change request and the change request for the resulting release indigenous error must be examined to learn how effective the maintenance process is and how it might be improved.

Although the definitions given above for these terms imply that the two error sets are distinct, in practice, the actual error populations do not fit the definitions one hundred percent. For example, the set that this study termed the operationally indigenous error set should include only those errors that were introduced during the original development of the software. In reality, this set may also include a few errors that were introduced during maintenance, but which were not identified until the maintenance release became operational. The release indigenous error set should include only errors that were introduced by the maintenance process. In reality, this set may contain some errors that, although caught by release testers, were in fact residing in the operational software and were not new to the maintenance release. Despite these imperfections, there was enough consistency in each set to treat them separately.

In characterizing the size of a release, some measure other than the total number of changes is necessary, because some changes (especially enhancements) tended to be more complex and time consuming than others. For this study, the total modified lines of code (new SLOC + changed SLOC + deleted SLOC) for all changes was used as the measure of release size.

The release characterization demonstrated that, on average, FDD releases are composed of about an equal number of error corrections and enhancements, but that the enhancements require significantly more effort and far more code. Comparing this effort and size data between enhancements and error corrections revealed that the productivity for enhancements was approximately seven times greater than for error corrections. Why this is so, and whether it is good or bad, remains to be seen. The characterization of maintenance errors revealed surprisingly few errors attributed to requirement specifications or to design. This deserves further investigation, especially since the qualitative analysis suggested that requirements deficiencies on software change requests were a problem. The preliminary characterization of error rates resulted in two different ways to normalize errors, one appropriate for operationally indigenous errors and another appropriate for release indigenous errors.

Qualitative analysis suggested that the FDD needs to provide better guidelines for content and format of change requests and release documents. The FDD also needs to enforce stricter adherence to the maintenance process, especially attendance at review meetings. The preliminary quantitative analysis provided many insights into FDD maintenance but also spawned as many new questions. The preliminary effort distributions indicated that design and implementation require more effort in maintenance than they do in new development. Exactly why this is so is not clear at this time.

7. Future Study of Software Maintenance in the SEL

The combination of qualitative and quantitative analysis methods has provided a comprehensive look at the software maintenance process in the FDD. From this researchers have made a good start at baselining this

process. Preliminary quantitative data analysis is based on only nine complete maintenance releases. More releases need to be studied. Also baseline models need to be extended to include an understanding of maintenance cost and cost estimation, plus a better understanding of error rates. Beyond this, future maintenance study activities need to provide a more complete understanding of the testing process and the inspection and certification process. The impact of software development practices on later software maintenance also must be measured.

The FDD has recently embarked on a major effort to port most of its software from IBM mainframes to UNIX workstations. This effort will result in a great many maintenance change requests of the adaptation type. The current study needs to analyze whether and how it should adapt itself to make the most use of the data that this transition will generate.

Once the understanding phase of the current study is completed, the assessing phase will begin. Researchers will design and carry out experiments through which they will be seeking answers to these questions and others:

1. How might we know when a product has outlived its usefulness?
2. What is the "right size" for a maintenance release?
3. Can we predict the most error-prone modifications, and if so how?
4. How can we more accurately estimate the cost of software changes?

This application of the QIP has expanded the SEL's understanding of the maintenance process and product in this environment. Further baselining, experimentation, and research should lead to recommendations for improvements to the maintenance process that can be packaged and instituted in the FDD.

References

1. McGarry, F., G. Page, V. R. Basili, et al., *An Overview of the Software Engineering Laboratory*, SEL-94-005, December 1994.
2. Basili, V. R., "Quantitative Evaluation of Software Engineering Methodology," *Proc. of the First Pan Pacific Computer Conference*, Melbourne, Australia, September 1985 [also available as Technical Report, TR-1519, Department of Computer Science, University of Maryland, College Park, July 1985].
3. Basili, V. R., G. Caldiera, F. McGarry, et al., "The Software Engineering Laboratory – An Operational Software Experience Factory," *International Conference on Software Engineering*, May 1992, pp. 370-381.
4. Briand, L., V. R. Basili, Y. M. Kim, D. Squier, "A Change Analysis Process to Characterize Software Maintenance Projects," *International Conference on Software Maintenance 1994*, Victoria, British Columbia, Canada, September 1994.

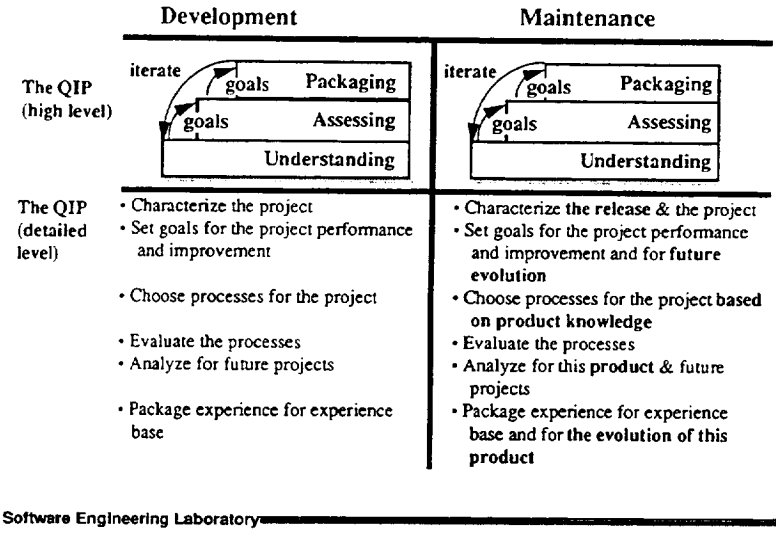
Building an Experience Factory for Maintenance

Jon D. Valett
NASA/GSFC

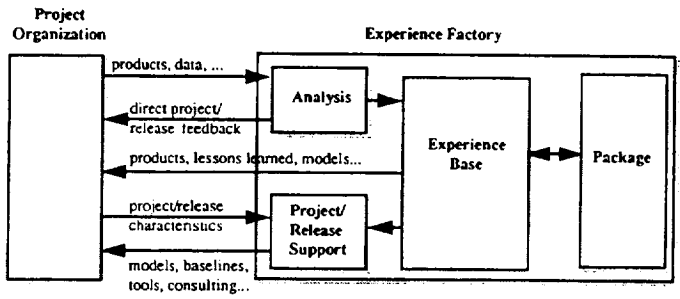
Steve Condon
CSC

Lionel Briand, Yong-Mi Kim, Victor Basili
University of Maryland

An Experience Factory for Maintenance



An Experience Factory for Maintenance (continued)



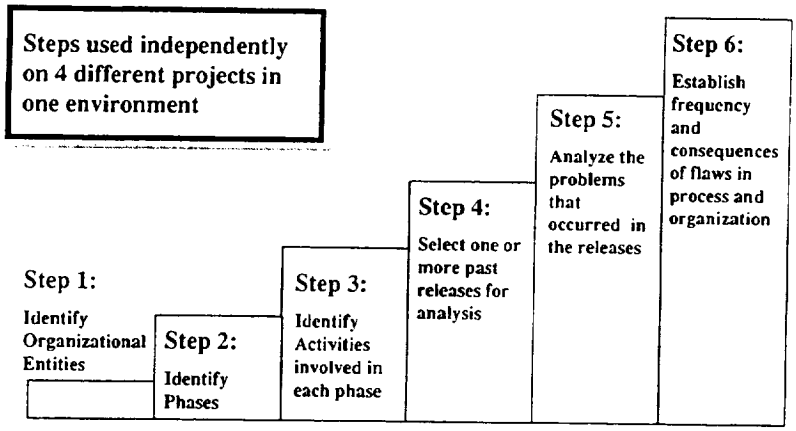
Experience Factory Organization is the same for maintenance, except
- Focus includes releases
- Analysis for release feedback requires quicker response (release life cycle ~6 months)
- Product evolution is emphasized (experience includes past experience on the same project)

Building the Experience Base

- Key First Step is Still Understanding
- Use SEL Development Experience as a Basis for Studying Maintenance
- Set Goals
  - Characterize the maintenance process
  - Characterize the maintenance products
- Use Qualitative and Quantitative Analysis
  - Qualitatively - Follow an organized approach to understanding
    - » Work with maintainers and project leads
    - » Process can vary across projects (process documents aren't always followed)
  - Quantitatively - Establish a measurement program to build baselines
  - Use quantitative data to understand the qualitative and use qualitative data to help define the data to collect

Qualitative and Quantitative Components are Critical to Maintenance Understanding

Qualitative Approach to Understanding*



* Briand, et. al. ICSM '94

Steps 1 - 3: Understand Organization and the Release Process

- Step 1 - Identify organizational entities
  - Identify distinct teams and their roles
  - Characterize information flow between teams

  eg. release approval passes from the configuration control board to the maintenance team
- Step 2 - Identify the phases of the release process

  eg. preliminary release definition...release design review...integration test
- Step 3 - Identify activities involved in each phase
  - Define each phase in terms of inputs, outputs, and activities

  eg. Design phase:

  Input is Release Review Document,

  Output is design, test plans and prototypes,

  Activities are changing design, changing code, unit testing and integration testing

Steps 1-3 Provide:

- Understanding of the process
- Point of comparison amongst projects
- Check of adherence to policies

Steps 4 - 6: Identify Problem Areas

- Step 4 - Choose a recent release for analysis
  - Choose recent releases
  - Choose releases with complete documentation
  - Choose releases where the technical lead is still available for interview
- Step 5 - Analyze causes of problems
  - For each change in a release use interviews and document review to:
    - » Determine the difficulty of the change
    - » Determine the maintenance process flaws
    - » Determine what delays and errors were caused by the process flaws

  eg. One change resulted in 11 errors.

  Due to Incomplete requirements and Unclear definition of responsibilities.

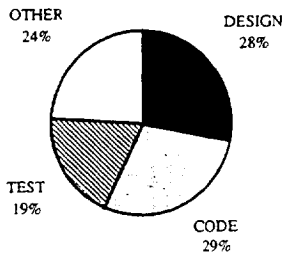
  Up to one month of effort lost.
- Step 6 - Establish frequency and consequences of flaws in the process and organization
  - Provide suggestions for improvement based on Step 5 from multiple projects, *e.g.*
    - » *Standard for content and format of change requirements needed*
    - » *Stricter adherence to process needed*
    - » *Document and review content needs explicit definition*

Quantitative Approach to Understanding

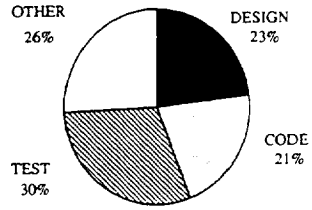
- Measurement program to establish baseline understanding of maintenance process and product
- Based on goal for the maintenance study generate questions such as
  - What is effort distribution during maintenance?
  - What are characteristics of maintenance releases?
  - What are characteristics of maintenance errors?
  - What are error and change rates?
  - etc.
- Measurement data includes
  - Effort by activity
  - Effort by type of maintenance change
  - Error and change data
    - » Time spent
    - » Source of errors
    - » Class of errors
  - Release estimates and actuals
  - Size of software under maintenance

Understanding Maintenance Effort

Maintenance Effort Distribution *



Development Effort Distribution **

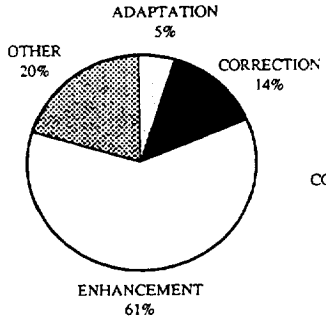


Design and Code are a Larger Percentage of Activity During Maintenance

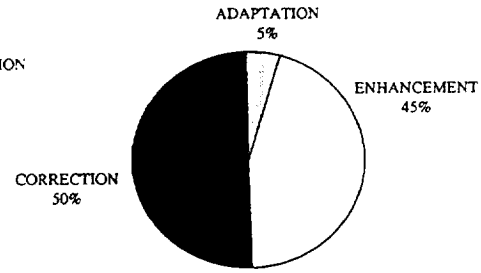
*Based on 11 projects
**Based on 11 different projects

Release Characteristics

Effort Distribution by Type of Change*



Changes by Type**



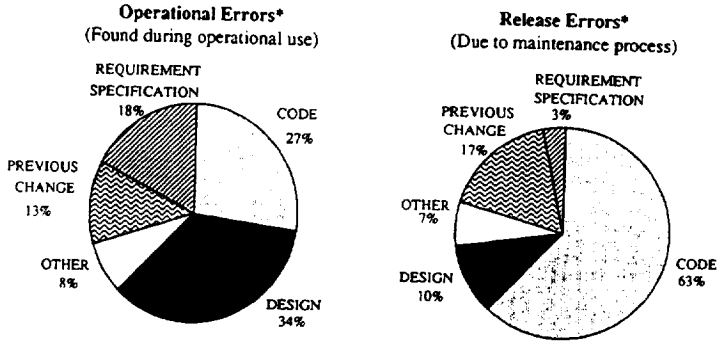
• 97% of code added and modified due to enhancement

Releases are made up of many small changes and large enhancements

* 11 Projects
** 83 Changes on 9 releases

Error Characteristics

Source of Errors



* 9 Releases

Error and Change Rates

- Operational Error Rate
  - 10 Errors / 100 KSLOC / year (5 min., 32 max.)
- Release Error Rate (through acceptance testing)
  - 0.8 Errors / Modified KSLOC (0 min., 6.9 max.)
- Change Rate
  - 3.7% of code modified / release (0.1% min., 11.7% max.)

- Based on 9 releases
- Modified KSLOC = 1000's of New + Modified + Deleted LOC
- Project size ranges from 48 to 227 KSLOC

Lessons Learned

- Include the Maintainers in the Study
  - Valuable to both groups
- Use the Qualitative Analysis to Help Define the Measurement Program
  - We now collect effort by change
  - We redefined our effort activities
  - We need to reexamine our error taxonomies
- Distinguish Between Operational Errors and Errors During Releases
- Define a Measure for Release Size
  - We use New LOC + Deleted LOC + Changed LOC

Studying Software Maintenance in the SEL

- Using Qualitative and Quantitative Understanding in Combination has been Very Successful
- Future Maintenance Study Activities
  - Baselining Activities Need to Continue to
    - » Understand cost and cost estimation
    - » Understand error rate
  - Understanding Testing and Inspections
  - Understanding how Development Impacts Maintenance
  - Understanding the Adaptation Process
  - Experiment with Process Changes
- We would like to be able to
  - know when a product has outlived its usefulness
  - know the "right size" for a release
  - predict the most error prone modifications
  - estimate the cost for changes
  - leverage our experience base to solve these quicker

Closing the Loop on Improvement: Packaging Experience in the Software Engineering Laboratory

Sharon R. Waligora, Linda C. Landis, Jerry T. Doland

Computer Sciences Corporation
10110 Aerospace Road
Lanham-Seabrook, Maryland 20706

Abstract

As part of its award-winning software process improvement program, the Software Engineering Laboratory (SEL) has developed an effective method for packaging organizational best practices based on real project experience into useful handbooks and training courses. This paper shares the SEL's experience over the past 12 years creating and updating software process handbooks and training courses. It provides cost models and guidelines for successful experience packaging derived from SEL experience.

1. Introduction

The Software Engineering Laboratory (SEL) is a partnership among NASA Goddard Space Flight Center (GSFC), Computer Sciences Corporation (CSC), and the University of Maryland; it has received international recognition for its achievement in continuous, measurable improvement in software products and processes. The SEL supports the Flight Dynamics Division (FDD) at GSFC, which builds software systems for satellite ground support and spacecraft attitude control.

The SEL has forged a process improvement approach that identifies the goals of the organization, initiates process improvement initiatives based on those goals, and measures the impact of those initiatives on the products produced. This approach is based on the concept of organizational learning from project experience, similar to the way that successful people learn from their experience and apply new techniques to the way they do their jobs. For example, once an improved process or new technology has been used successfully by a pilot project, it must be shared with other projects to broaden its impact. This expansion of process improvements throughout the organization is accomplished in the SEL through packaging. Packaging is a structured mechanism for capturing the best practices, the most effective technologies, and the lessons of past experience and communicating that information throughout the organization. By making improvements part of the standard way of doing business, packaging closes the process improvement loop.

Recent SEL experience shows the benefits of packaging. In the late 1980s, the SEL began experimenting with several software engineering technologies, including object-oriented design, the Ada language, and the Cleanroom methodology. Around 1990, the SEL updated and improved its methodology guidebooks and developed new training courses. As a part of this update, beneficial parts of each of the experimental technologies were integrated into the methodology along with process improvements derived from best practices that had evolved since the guidebooks were last revised. Key product measurements from the 1990–1993 time period show dramatic across-the-board improvements over baseline measurements taken in the mid- to late 1980s: a three-fold increase in software reuse that resulted in significant cost and schedule savings, and a 75 percent decrease in software development

errors. These improvements can be traced to new techniques—such as the high-reuse process that grew out of the Ada/OOD experimentation and the consistent use of software inspections and code reviews that was introduced by the Cleanroom methodology—which were highlighted and stressed in the new guidebooks and training.

These standards are not just "shelfware;" a survey of the local software engineering staff indicated that users find the guidebooks relevant and easy to use. The survey results revealed that 95% of the software developers use the guidebooks, with software project leaders and managers using them most frequently. In addition, SEL guidebooks have been cited by industry publications, such as *The Software Practitioner*, as excellent examples of practical software engineering standards. The SEL's *Manager's Handbook* has been used as a textbook for software management courses at the University of Maryland, the Johns Hopkins University, and McGill University in Canada. The training courses also have been well received. Course evaluations consistently rate the SEL courses as highly relevant and informative, with 90% of the participants stating that the courses were well worth the time they had invested.

This paper describes the SEL packaging process—our approach to capturing and reusing experience. We discuss the methods used to synthesize experience into a standard software engineering process and to effectively communicate that process to the software engineers. We consider the needs of the audience; sources of information; issues of package scope, content, and format; and offer cost and schedule models for packaging. Finally, we summarize some of the key lessons learned and rules of thumb for packaging experience.

2. Background

2.1 SEL Process Improvement Paradigm

The SEL's process improvement paradigm is shown in Figure 1. The first and most important step is *understanding* how an organization currently does business and what it values. This is done by characterizing the products generated and the process that is used to produce them. In the second step, *assessing*, the organization sets goals for improvement, and experiments with process changes, such as a new technology, that might help achieve its goals. This is done by introducing a process change on pilot projects, assessing its impact on the product, and refining it if necessary before selecting it for use throughout the organization. The final step is *packaging*, where the successful new technologies and procedures are integrated into the organization's standards and training program so that all projects may benefit from the changes.

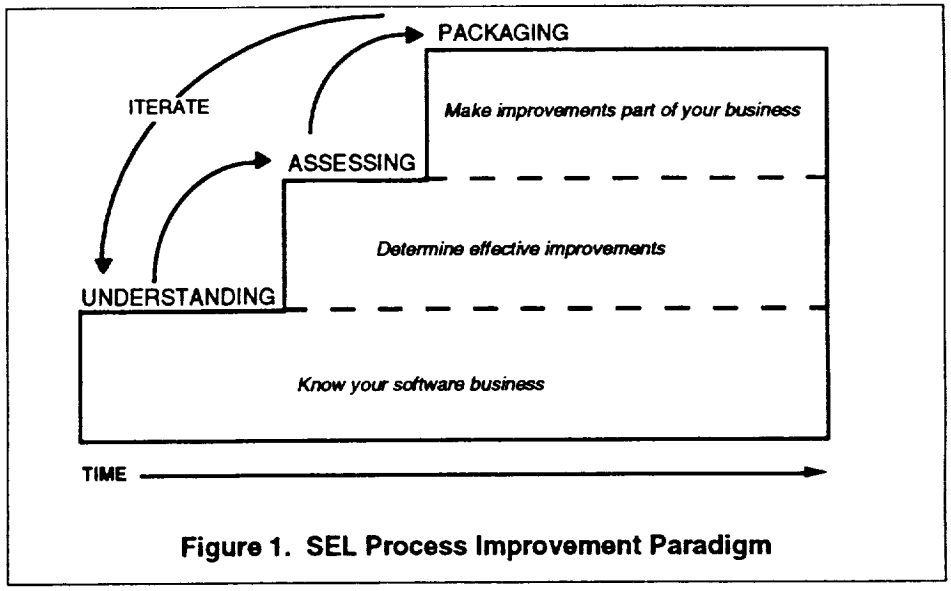


Figure 1. SEL Process Improvement Paradigm

Within the SEL, a group of researchers, analysts, and support personnel (separate from software developers) perform process improvement activities. They collect and analyze software project measurements to produce models and standards for use by the projects. They design and monitor experiments with new technologies and modified procedures to determine their applicability to the local environment and refine/tailor them for optimum use in the FDD. They package research results and local experience in process guidebooks, training courses, and tools.

2.2 Experience Packaging

The SEL relies on its measurement program to provide a view into actual product and process characteristics. Similarly, it uses experiments to gain additional insight into the effect of new or modified techniques, tools, and processes on the products. Based on this information, the SEL identifies and captures the most appropriate practices/technologies in "experience packages." These packages are in the form of standards, tools, and training that give practical guidance on how to apply the new techniques in the context of the local process. This guidance effectively captures the results of the understanding and assessment phases, packages them for "reuse" by subsequent projects, and integrates them into the routine software business. SEL packages are always designed with the local organization's needs in mind, but many have found broader applicability outside the SEL domain.

Packaging is performed by a team that is independent from the development organization, but whose members work closely with development personnel. Packagers talk with developers to learn about improvements made within the projects while using the standard process in a changing environment. They study project documentation and data to verify their findings. Using the current documented software development process as a reference point, packagers determine the evolved state of the practice based on current project experiences and create new baseline models. The packagers also integrate beneficial new methods derived from SEL experiments into the standard software development process. Working in consult with the project personnel who will use the materials, the packagers synthesize all of this information into an updated process. From there, they design the optimal presentation of the information, develop the package, and introduce it to the users through organized deployment and delivery.

2.3 SEL Experience Packages

The SEL has developed a primary set of guidebooks, training, and tools that document and support the evolving local process. In addition to these major packages, the SEL produces technology reports and interim packages. These products support the assessing step of the process improvement paradigm and have shorter life spans and are less extensively distributed. Technology reports record the results of SEL studies with specific technologies and techniques. They contain the recommendations and rationale for including all or parts of the subject technology in the standard local process or for abandoning the technology or process change as inappropriate for this environment. Interim packages fill the gap when a new technology is being assessed, while its applicability in the environment has not been determined or sufficiently refined for widespread local use. Some interim packages have been incorporated into later updates of the standard methodology, and others have been entirely superseded.

2.3.1 Guidebooks

The SEL has produced a set of guidebooks that defines the baseline development standard. The guidebooks communicate the rationale for the methods and offer guidance for applying them, rather than specifying detailed procedures. We have found that this level of detail allows each project the flexibility to define project-specific procedures as needed (based on those used by previous similar projects) to meet the needs of its current environment. Given that detailed procedures change as improvements are introduced and the organization evolves, segregating procedures from the formal documentation mitigates the need for project waivers and continual updates to the standards.

In addition to the baseline standards, several specialized documents have been developed to support tailored applications of the local process,

such as implementing in a particular programming language or using a tailored methodology. We have learned that it is best to document language-specific processes separately from the baseline methodology; this allows them to be modified as frequently as needed to keep pace with rapidly changing technology.

Baseline Standards

- *Manager's Handbook for Software Development*—Contains the models, guidelines, and acceptable processes for managing the development of flight dynamics systems. It provides specific guidance for using planning and performance models to successfully manage software engineering projects.
- *Recommended Approach to Software Development*—Presents guidelines and standards for developing software in the flight dynamics environment. Intended for developers and technical managers of software development projects, it describes the recommended practices for each phase of a software development life cycle, including key activities, products, measures, methods, and tools.
- *Cost and Schedule Estimation Study Report*—Presents planning models for cost and schedule estimation based on local project data. The planning parameters are built into spreadsheet tools for use by project managers and are updated yearly based on ongoing analysis.

Tailored Standards

- *Ada Developers' Supplement to the Recommended Approach*—Presents guidelines for programmers and managers who are developing flight dynamics software in Ada. Intended to be used in conjunction with the *Recommended Approach to Software Development*, it provides additional detail on reuse and object-oriented analysis and design.
- *C Style Guide*—Presents the recommended practices and coding style for programmers using the C language in the flight dynamics environment. The guidelines are based on generally recommended software engineering techniques, industry resources, and local convention. It offers preferred solutions to C programming issues and illustrates through examples of C code.
- *Cleanroom Process Handbook*—Presents guidelines for using the Cleanroom methodology in the flight dynamics environment. It describes the Cleanroom life-cycle model and the specific activities performed in each life-cycle phase. It also addresses pertinent managerial issues and highlights the key differences and similarities of the SEL Cleanroom process and the standard development approach. This handbook started out as an interim package and later became a tailored standard after the methodology matured in the local environment.

Figure 2 shows the development history for several SEL guidebooks. The *Manager's Handbook* and *Recommended Approach* were initially developed in the early to mid-1980s and then updated around 1990. The SEL developed a few interim guidebooks, a *Generalized Object-Oriented Development (GOOD) Guide* and an

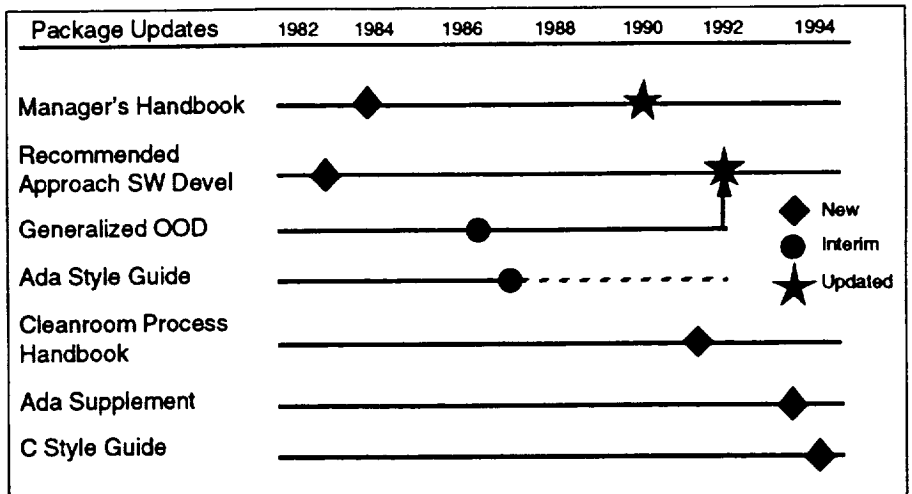


Figure 2. SEL Guidebook Development History

Ada Style Guide to support experiments in the late 1980s. The *GOOD Guide* eventually was absorbed into the next update of the *Recommended Approach* while the *Ada Style Guide* was replaced by an evolved industry standard.

2.3.2 Training Courses

SEL training packages include several core courses and a training plan that documents the goals of the training program, describes course content, and recommends the training sequence for project personnel. The core courses cover the SEL organization, methodology, and process improvement approach, and provide in-depth training in the application area and software development process. These courses are updated as needed to reflect changing process elements within the SEL. All staff (managers, developers, maintainers, testers) are expected to participate in the core set of training classes. Courses include:

- *Orientation to the FDD*—Orients the newcomer to the local environment, application/mission, organization, process improvement approach, and methodology; 6 hours–lecture.
- *Principles of Flight Dynamics*—Bridges the gap between academic mathematics and physics, and their application in flight dynamics software systems; 30 hours–lectures and homework exercises.
- *Recommended Approach to Software Development*—Illustrates the use of and rationale for applying the local software development methodology (based on the guidebook discussed above); 24 hours–lectures and workshops.
- *Task Leader/ATR Training*—Demonstrates how client and contractor project leaders work together within the context of the contract to successfully manage software projects; 12 hours–lectures, workshops, and interactive exercises.

2.3.3 Tools

An important aspect of packaging is the infusion of technology in the form of support tools for use by project personnel. The SEL developed a project management tool called the Software Management Environment (SME) that puts local experience at the fingertips of project managers. The SME provides access to the SEL's database of previous project information and baseline process models. Using the SME, a manager can, for example, compare the growth rate of source programs or the error rate of the current project against the models, or, using data from similar projects in the database, the manager can predict future trends on the current project. This tool has helped institutionalize the SEL process, because project managers can use it to gain insight into their software projects.

3. Packaging Guidebooks and Training

The SEL's current packaging process is based on the fundamental understanding that the local software engineers are the primary users of our products. When developing guidebooks and training courses, the SEL emphasizes user involvement to ensure that the documented process matches what is actually done, that recommendations are based on agreed-upon procedures, and that the end product will be useful to the software engineers.

This approach has evolved over the years, with the SEL learning from some missteps along the way. For example, the first issue of the SEL's baseline standard, the *Recommended Approach to Software Development,* was a classic case of the "typical" approach (described below). Originally conceived and published without much input from local, practicing software engineers, the standard was ill-received and almost immediately recalled for revision. At that point, early SEL "packagers" decided to take a new approach, and just write down exactly how the developers actually produce, test, and maintain software in this environment. Their new document, albeit rough, formed the basis of the current *Recommended Approach.* This guidebook has since been refined based on the experience packaging concepts discussed in this paper. At its core is the concept that the users know best how they do their jobs and what guidance they need to support them in their work.

3.1 Typical Industry Approach

The typical industry approach to defining process often results in the creation of "shelfware," i.e., standards and procedures that aren't used and wind up gathering dust on bookshelves throughout an organization. This commonly used process (shown in Figure 3) begins with managers or quality assurance personnel creating a list of topics to be addressed based on an external (industry) standard. The topics are then divided up and distributed to people who have expertise in the subject areas. These people do their best to draft the standards and procedures for their particular topic in their spare time—because rarely are there resources or time allocated to the effort—while also meeting their regular project responsibilities. The draft standards are subsequently distributed to a small group of reviewers and turned into "legalese" by incorporating everyone's review comments. They are then assembled by a coordinator, published, and distributed to the developers. When the standards are delivered, it may well be the first time that most of the developers will have seen them. They peruse them, often don't understand them or recognize their relevance, and so, they place them on the shelf, and continue following their current process.

3.2 SEL Approach

The SEL packaging process, illustrated in Figure 4, involves the users directly. Two separate groups each play an important role in this process: the *packagers* who document the process and the *software developers* who are the users of the process and the supporting packages. The packagers, who are usually dedicated full time to the effort, are responsible for gathering and distilling process information and then presenting it in a useful form. The experienced developers are one of their key sources for this information. SEL packagers also consider information from the SEL's metrics database and results from SEL experiments when defining the updated process. This information flow is depicted by the outer loop in Figure 4.

The inside loop in Figure 4 represents the iterative method used to refine the software development process and develop the package. Often, as the preliminary step in a packaging effort, project personnel are interviewed or invited to a brainstorming session to gather information and requests for package content. They explain to the packagers how the process is being applied in the current environment; they raise issues and problem areas; and they offer suggested improvements. Based on the identified strengths

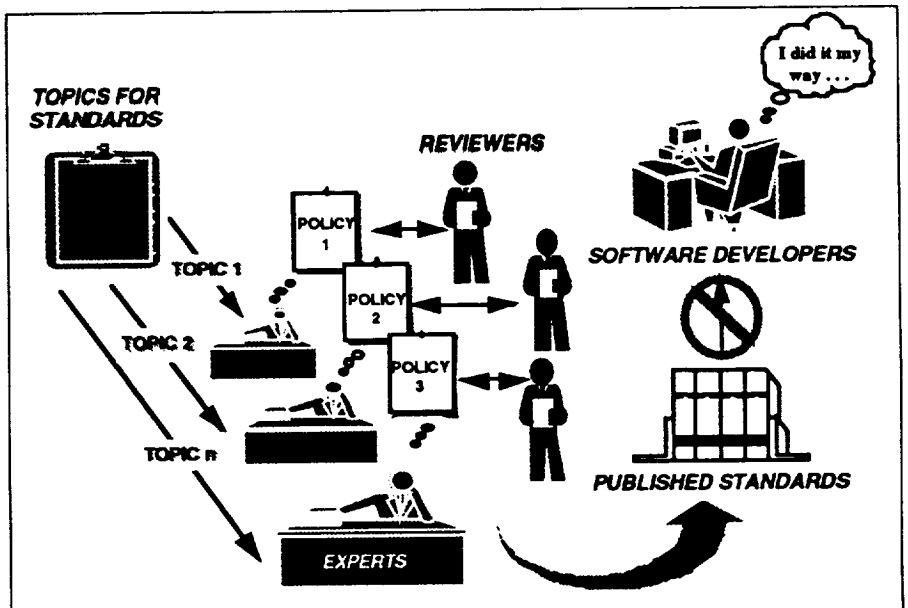


Figure 3. Typical "Shelfware" Development Process

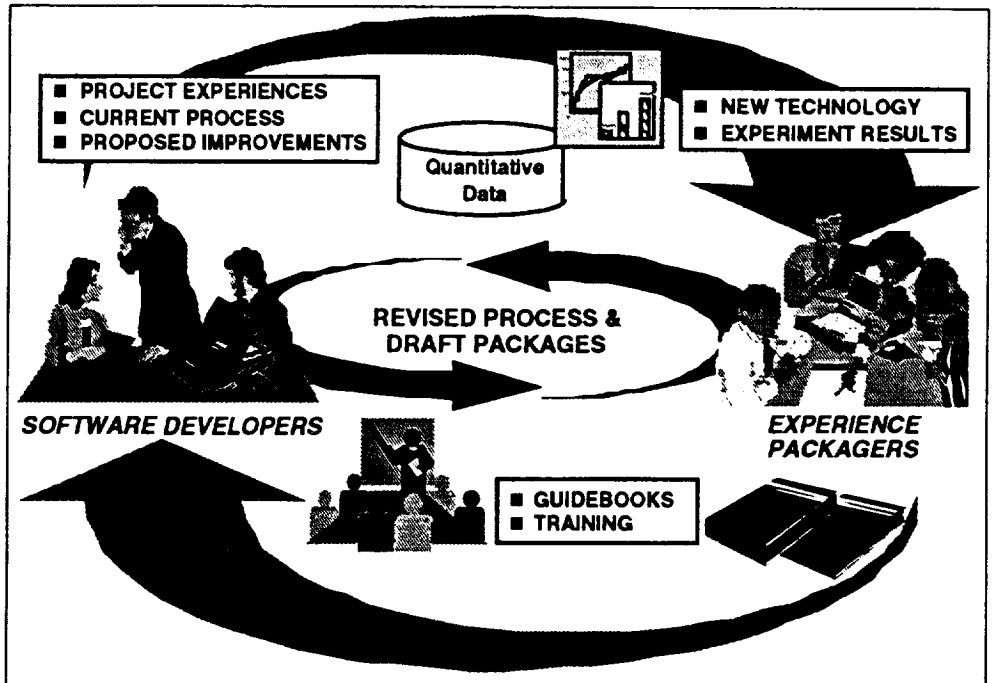


Figure 4. SEL Package Development Process

and weaknesses of current books and courses (if available), packagers design and prototype new packages using key developers as reviewers for both content and usability.

We find that this approach results in accurate, usable guidebooks and effective training courses. The developers feel connected to the products because they were involved in creating them, and they appreciate the fact that they were not burdened with producing them. The quality of the package is top-notch because the team that produced it was dedicated to the effort and skilled in communication, information analysis, and desktop publishing.

3.3 Packaging Activities

The basic activities in the experience packaging process include:

- Information gathering and synthesis
- Package development
- Package deployment

These activities are looked at in detail in the paragraphs that follow.

Information Gathering and Synthesis

Packagers first review any existing version of the standard or guidebook that is to be updated or, in the case of training, the information on which the course will be based. In essence, they apply step 1 of the process improvement paradigm, understanding; they baseline the documentation that currently exists in the environment. Packagers then apply step 2 of the paradigm when they interview experienced developers, maintainers, testers, and managers to assess how closely the actual software engineering practice maps to the documented process. They determine where the process has changed and how it has been tailored for different situations, and they validate this information by analyzing empirical data. Packagers also review SEL study reports and experiment results to identify new techniques that have been recommended for inclusion in the standard process. Finally, all of the input is synthesized to define the new process. Facilitated workshops are then organized to clarify issues and to develop consensus on the process content.

Package Development

Once the content is decided, the focus shifts to designing the right package to communicate the information. Users are interviewed to understand their work habits and approaches to learning. They are asked to cite the strengths and weaknesses of existing courses and guidebooks. This helps packagers choose the most effective communication style based on the users' preferences and to choose the most appropriate course format for their audience and subject matter. Although guidebooks and training courses have the same goal of describing a process and the rationale for applying it, they are fundamentally different communication media. Whereas guidebooks provide standalone text references, successful training courses leverage the combination of written (visual) material and a human instructor in an interactive setting. These two types of packages require different production processes.

Guidebooks

For guidebooks, the packagers begin by developing prototypes of key sections to get early feedback from the users on the "look and feel" of the document. This is an extremely effective way to validate and further refine the packagers' understanding of the right level of detail and to get feedback on different communication styles. The text and layout of the guidebook are then developed iteratively, allowing the users to review both content and format. As the document evolves, key developers serve as expert reviewers. The packagers also solicit comments from a broader group of reviewers in the final stages of development, before deploying the final product. The SEL has found that a typical guidebook may go through 3–4 iterations before it is ready for final review. Throughout the review and feedback process, SEL packagers carefully scrutinize and synthesize all review comments to avoid inserting into the guidebooks "special interest" language (e.g., individual preferences or compliance "loopholes") and to guard against writing in the obtuse style that often results from mass review.

The SEL has discovered that this iterative process results in user-friendly guidebooks that are actually used. SEL guidebooks use graphics to illustrate concepts and lead users to the information they need. For example, the *Manager's Handbook* uses an innovative graphic layout to communicate measurement models, and the *Recommended Approach* includes keywords, notecards, icons, "chapter highlights," and a detailed subject index. Both are designed as references, rather than for one-time reading.

Training Courses

When developing training, the first step is setting goals for the course and identifying the primary audience. This is typically done in a brainstorming session with personnel from the development organization, where packagers gather information about user needs. Packagers then meet with the selected instructor(s) to define the course outline and to choose the best organization and apportioning of the information into individual class sessions. Packagers work with the instructor to determine the appropriate level of interaction for the various classes based on the material to be covered. We have found workshops in which participants can practice new techniques to be essential when teaching new skills. Classroom brainstorming and role playing exercises help students discover new ways of thinking about familiar subjects, and lectures are most useful for conveying new information.

Course developers create a preliminary set of training materials, including lecture slides, project examples and handouts, and workshop exercises. The course is then reviewed for content and continuity. When the content is stable, the slides are livened up with graphics and polished to become a cohesive package that will hold the participants' interest. At this point, a dry-run of the course is held as a final review. This allows the instructor to get comfortable with the material and provides a forum for soliciting final review comments before deployment.

Package Deployment

Deployment is a critical step in the packaging process. If guidebooks are simply dropped on people's desks or training courses are simply announced, the packaging effort is likely to fail. Software developers must read the books and attend the courses for the information transfer to take place. The SEL has found that a publicity campaign is important. The people need to

know that a new guidebook or course is coming, that it is new and different, that it will be useful (we show examples), and that their colleagues (we name them) contributed to it. Those who were involved in the package review already have "bought in," so their marketing help is solicited. Managers are invited to attend dry runs of the training courses and to review guidebooks. This is an excellent way of getting their input and support; managers are in the best position to encourage their people to attend training and to use the guidebooks. Briefings at all-hands meetings and posters also work well to call attention to a new package.

The SEL has found that guidebooks that are closely followed by a related training course are particularly effective for infusing process change. The training course serves to get users "into" the guidebook, demonstrating for them how it can support their work. Training also provides a forum where revised elements of the process can be pointed out and clarified. Accompanying workshops provide a safe environment for getting hands-on experience with new techniques.

3.4 Investment in Packaging

In the SEL, we spend about 10% of the software budget on process improvement activities, of which a relatively small percentage is spent on the packaging process described here. Over the past 5 years, the SEL has spent 1.5% of the total software budget on packaging: 1% on guidebooks and 0.5% on developing training courses.

During that time, we have tracked costs and schedules for most of the packages that we have produced. Our data show that it costs about 24 staff-hours per page to develop guidebooks and 55 staff hours per hour of class time to develop training courses. The effort expended and the relative size of the guidebooks and training courses are shown in Tables 1 and 2, respectively. Please note that these numbers reflect the effort spent by the packagers only; none of the developers' time (which is relatively small) is included here.

Table 1. SEL Guidebook Cost and Schedule Data

Guidebook	Pages	Effort (staff-months)	Schedule (months)
Manager's Handbook	76	13.2	23
Recommended Approach	200	28.6	30
Ada Supplement	33	5.0	10
Software Measurement Guidebook	131	20.6	20
C Style Guide	89	3.7	4.5

Table 2. SEL Training Course Cost and Schedule Data

Course	Class Hours	Effort (staff-months)	Schedule (months)
Orientation	6	*	2
Task Leader/ATR Course	12	4.2	5
Recommended Approach	24	8.8	6
Principles of Flight Dynamics	30	*	7

* Effort data not available

The calendar time required to develop a package is harder to predict. For guidebooks, it appears that the schedule is driven by the scope of the material; the broader the subject, the longer it will take. For training, schedule depends more on the length of the course and the level of detail presented. Our experience indicates that time to develop generally depends on how new and different the material and/or the format is and how difficult it is to capture the experience. For example, capturing process information for the *Recommended Approach* and *Ada Supplement* took much longer than distilling information for the *C Style Guide*, which addressed a single product standard.

4. Lessons Learned

Over the past 12 years, the SEL has tried different approaches to packaging the software process. We have continually improved both our packaging process and our products based on user feedback and measured results. Some guidelines follow for producing successful guidebooks and training courses based on our lessons learned.

Standards Should Reflect Local Experience

Standards should be based on the best practices of *what is actually done* locally rather than *what an outside source says should be done*. Developers and managers tend to ignore standards that have little or no connection with their real world. It's best to introduce the most promising new methods and techniques from ongoing experiments so that the guidebooks will be current when released. Where possible, address the problem areas identified by the developers during the interviews and workshops.

It's important to clearly state what is expected to be done and provide guidance for decision-making and tailoring. Rarely will a methodology be applied exactly as it is specified; therefore tailoring guidance is extremely important. Don't overload the guidebooks with rationale. If the methodology is based on local experience, the rationale will be evident to most users. Training courses provide an opportunity to elaborate on the methodology in an interactive setting, and they are an excellent vehicle for demonstrating its proper application using local examples.

Design Packages for Ease of Use

Our interviews with developers indicate a typical usage pattern for SEL packages: developers initially read a guidebook cover-to-cover or attend a training course to get the whole story and then they primarily use the guidebooks and training materials as references during project execution. Therefore, the packages are designed with this type of use in mind. We have found several key attributes that help usability:

- Keep documents small. It's best if they can fit in a briefcase.
- Make information easy to locate. Use graphics to guide the eye.
- Use clear direct language and local terminology.
- Use graphs and pictures to clarify text.
- Provide a good, hierarchical index.

Treat Developers as Customers

Developers and managers are the primary users of guidebooks and training courses. The products are intended to help them do their jobs better. When soliciting their input to support a packaging effort, we need to treat them as customers:

- Listen to them.
- Solicit their requirements.
- Build useful and usable products for them.
- Keep them involved in package development.
- Don't ask them to do the work. (They are busy building software.)

Dedicate a Small, Highly Skilled Team to the Packaging Effort

The packagers' job is to gather, distill, and communicate information based on a thorough understanding of the environment. This requires additional skills that are not commonly found among software developers. For example, people skills are extremely important. Packagers need to be good listeners and interviewers to elicit information from people. They must be able to analyze and synthesize information and

then organize and present the resulting process effectively. Desktop publishing and technical writing skills are also critical to the quality of the final product.

We have found that a team of three people tends to be optimum for developing packages. The team is composed of

- A lead writer or course developer who has experience in software engineering, but not necessarily in the local domain;
- A domain or subject matter expert—usually a manager or senior developer who has extensive experience in the local domain (for courses, this person may also be the instructor);
- A publication specialist, who has expert presentation, editing, and desktop publishing skills.

Don't Get Bogged Down in Detail; Update Packages Judiciously

Programmers get annoyed and confused when they are constantly receiving updates to the standard process. The amount of maintenance required to keep standards current depends on the level of detail in them. We found that it is best to avoid the detailed "how to" information; instead, build in space for the process to evolve, which is happening all the time in an improving organization. Interim changes, such as updated cost models and new techniques, can be integrated into training courses and tools. For example, the SEL updated the Recommended Approach course twice in 18 months to reflect change in the local process; whereas the guidebook on which the course is based has been updated only once in 6 years since its deployment in its revised form.

Don't include experimental processes in the standards until they have been tried and proven beneficial in the local environment. In addition, keep language-specific standards separate from process information, because they tend to change independently. Let the amount of change in the organization, process, or environment drive when guidebooks and training courses are updated, rather than a fixed time interval.

5. Summary

For process improvements to be effectively infused throughout an organization, they must be packaged in a useful form. Effective standards, guidebooks, and training courses cannot be produced by programmers in their "spare" time. It takes a focused effort from a team of dedicated people with the proper skills to capture, synthesize, and communicate the improved software process. The SEL has demonstrated its commitment to broad-based improvement by investing both time and money in experience packaging. As a result, we have produced the high-quality guidebooks and training courses described in this paper. The investment has paid off—as it will for other organizations committed to managing their software process. Today, software developers in the SEL are building software faster, better, and cheaper using many techniques and methods that were considered experimental only a few years ago.

Acknowledgment

The authors thank Maureen McSharry for her editorial help in producing this paper and the corresponding presentation given at the Nineteenth Annual Software Engineering Workshop.

References

1. McGarry, F., G. Page, V. Basili, et al., *An Overview of the Software Engineering Laboratory,* Software Engineering Laboratory, SEL-94-005, December 1994

2. Landis, L., F. McGarry, S. Waligora, et al., *Manager's Handbook for Software Development (Revision 1),* Software Engineering Laboratory, SEL-84-101, November 1990

3. Landis, L., S. Waligora, F. McGarry, et al., *Recommended Approach to Software Development (Revision 3),* Software Engineering Laboratory, SEL-81-305, June 1992

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

5. Kester, R., and L. Landis, *Ada Developers' Supplement to the Recommended Approach,*

Software Engineering Laboratory, SEL-81-305SP1, November 1993

6. Doland, J., and J. Valett, *C Style Guide*, Software Engineering Laboratory, SEL-94-003, August 1994

7. Green, S., *Software Engineering Laboratory Cleanroom Process Model*, Software Engineering Laboratory, SEL-91-004, November 1991

8. Hendrick, R., D. Kistler, and J. Valett, *Software Management Environment (SME) Concepts and Architecture (Revision 1)*, Software Engineering Laboratory, SEL-89-103, September 1992

9. Doland, J., R. Pajerski, and S. Waligora, *Software Engineering Laboratory Training Plan*, Software Engineering Laboratory, SEL-93-TP1, September 1993

Closing the Loop on Improvement: Packaging Experience

Sharon Waligora
Linda Landis Jerry Doland

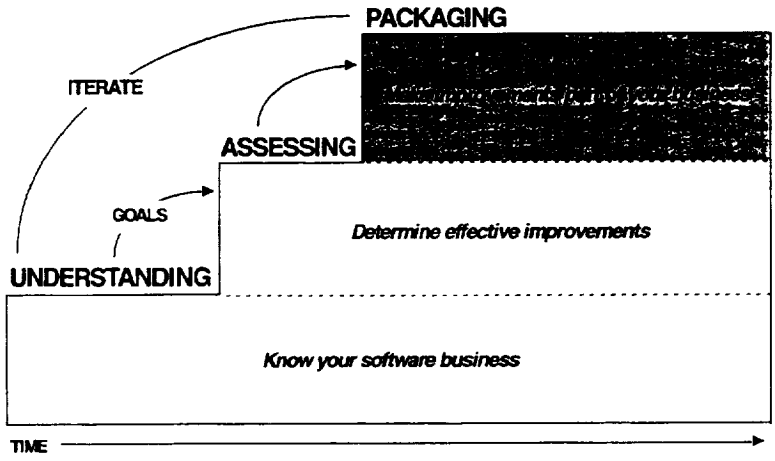
Computer Sciences Corporation

Topics

- What is Experience Packaging?
- The SEL Packaging Process
- Cost and Schedule
- Package Development Guidelines

Where Does Packaging Fit In?

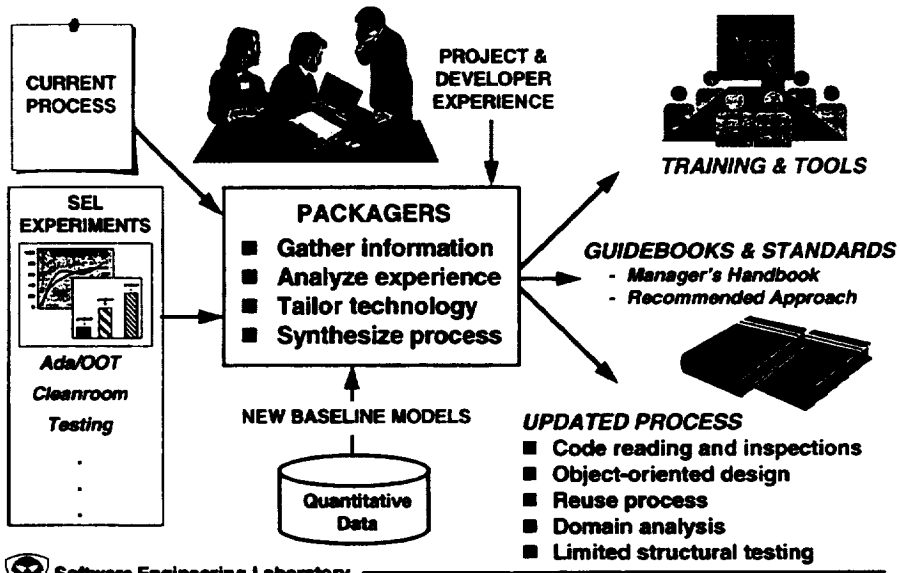
SEL Process Improvement Paradigm



Software Engineering Laboratory

SEW11/94 3

What is Experience Packaging?



Software Engineering Laboratory

SEW11/94 4

SEL Guidebooks, Tools, and Training

Guidebooks

Baseline Packages:
- Manager's Handbook
- Recommended Approach to Software Development
- Guide to Cost Estimation

Tailored Packages:
- Ada Supplement to Recommended Approach
- C Style Guide
- Cleanroom Process Handbook

Training

SEL Training Plan

Courses:
- Orientation
- Principles of Flight Dynamics
- Recommended Approach
- Task Leader/ATR

Tools

Software Management Environment
Automated data collection

Are SEL Packages Used?

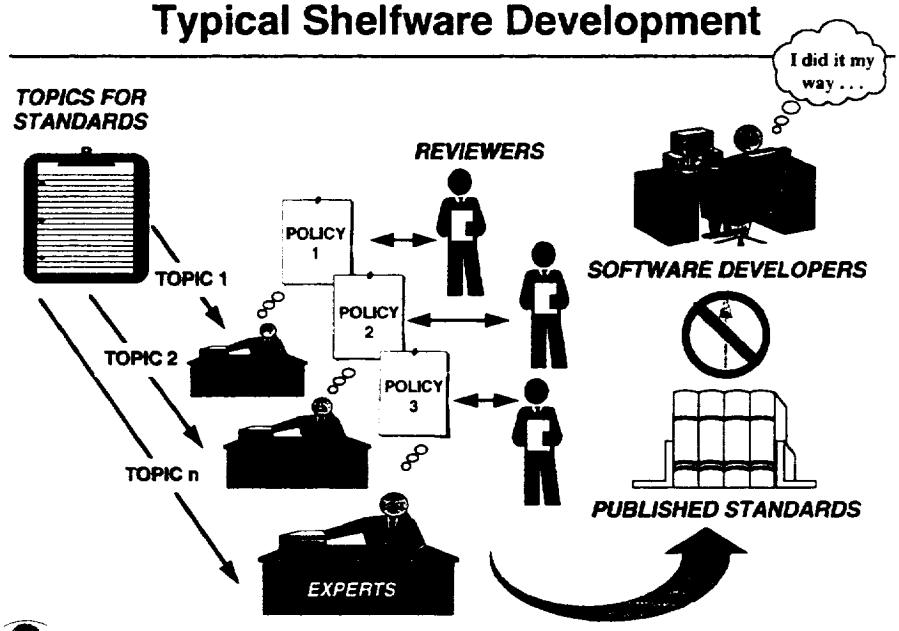
Guidebooks

- **Survey of 110 developers, maintainers, and managers**
  - **89% have used guidebooks**
  - **76% of project leaders and managers use guidebooks regularly**
  - **95% of developers found the guidebooks fairly easy to use and understand**

Training

- **Course evaluations show**
  - **95% of participants found course content directly applicable to their jobs**
  - **Most participants felt training was valuable and worth their time**

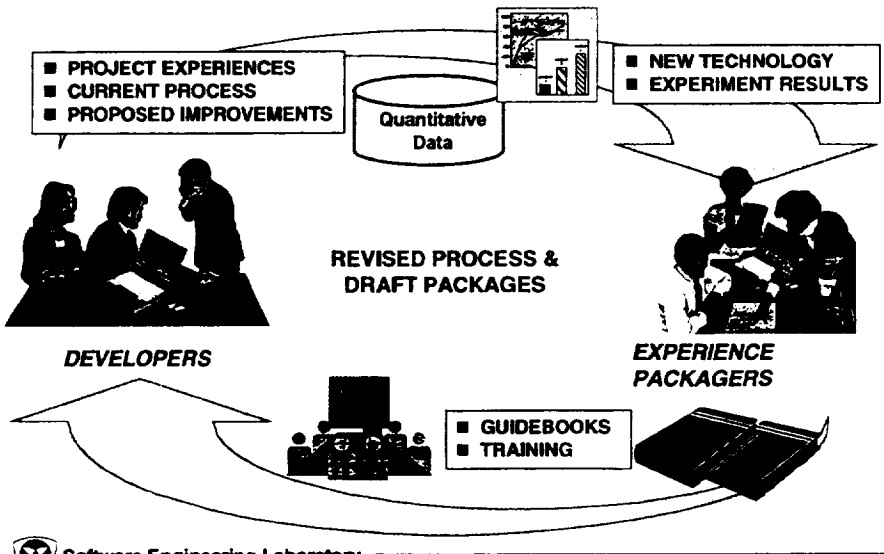
Typical Shelfware Development



Software Engineering Laboratory

SEW11/94 7

SEL Experience Packaging Process



Software Engineering Laboratory

SEW11/94 8

How Much Does a Package Cost?

- Cost models
  - Guidebooks = 24 staff-hours per page
  - Training = 55 staff-hours per class hour
- 1.5% of software budget is spent on packaging
  - Approximately 1% on guidebooks
  - Approximately 0.5 % on training courses

Guidebooks

	staff-months	pages
Manager's Handbook	13.0	76
Recommended Approach to Software Development	28.6	200
Ada Supplement	5.0	33
C Style Guide	3.7	89

Training Courses

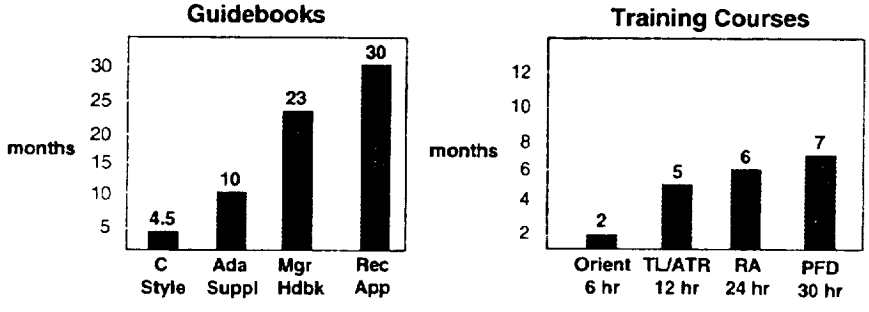
	staff-months	class hours
Orientation Course	*	6
Principles of Flight Dynamics	*	30
Recommended Approach	8.8	24
Task Leader/ATR	4.2	12

* data unavailable

Note: Process improvement is 10% of total SEL software budget

How Long Does it Take to Develop a Package?

- Time to develop is dependent on
  - Novelty of material and format
  - Availability of packagers
  - Ease/difficulty of capturing experience data



Guidebooks Should Reflect Local Experience

- Document the best practices
  - From local domain experiences (what *is*)
  - Not from outside experts (not what *should be*)
- Introduce most promising new methods and technologies from successful experiments
- Address problem areas identified by developers
- Clearly state what process/method is expected
- Provide guidance for decision-making and tailoring
- Do not overload with rationale
- Use training courses to expand, show good examples, and to explain rationale

Design Packages for Ease of Use

- Keep documents small
- Make information easy to locate (index, icons, graphics)
- Present material clearly and directly
- Use local terminology
- Prototype to get early feedback on package "look and feel"

If you can't find information and understand it, you can't use it.

Treat Developers as Customers

- Developers are the users for guidebooks and training
- Experienced developers are the key source of information
- Treat developers as customers
  - Listen to them
  - Solicit requirements
  - Build useful products
  - Keep them involved in package development
  - Don't ask them to do the work

Dedicate a Small, Highly Skilled Team to Packaging Effort

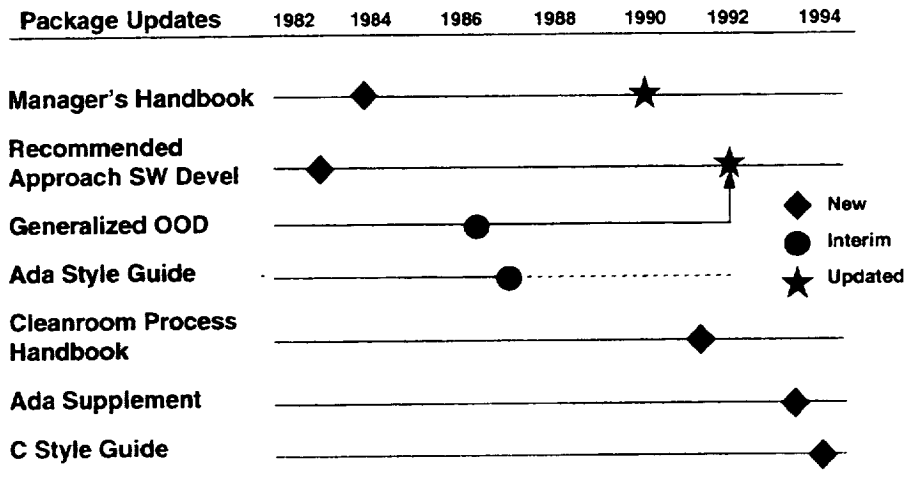
- Packagers gather, distill, and communicate information based on a thorough understanding of the environment
- Packaging team
  - Lead writer or course developer
  - Domain or subject expert
  - Publication specialist

- Packager expertise
  - Software development experience
  - Good interviewing and listening skills
  - Strong interpersonal skills
  - Able to analyze, synthesize, and organize information
  - Presentation skills
  - Technical writing
  - Desktop publishing

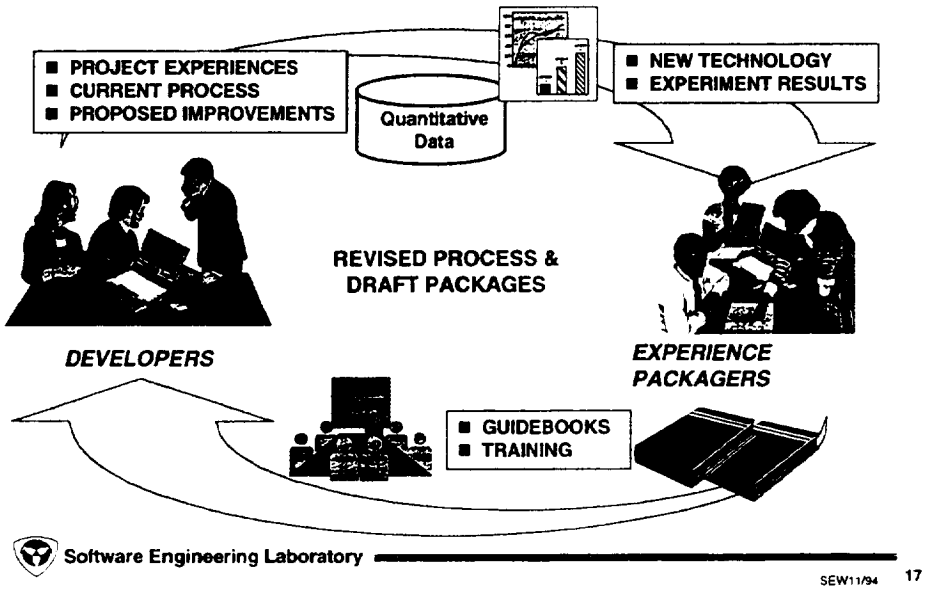
Update Packages Judiciously

- Guidebook maintenance depends on level of detail
  - Avoid detailed "how-to" information
  - Build in space for process to evolve
- Use training courses and tools to integrate interim changes
  - Cost model updates
  - New technique guidelines
- Include experimental processes when proven locally
- Separate language standards/style from process descriptions
- Update guidebooks when the organization, process, or environment changes significantly

SEL Package Development History



SEL Experience Packaging Closes the Loop



Software Engineering Laboratory

SEW11/94 17

Session 2: Process

Process Maturity Progress at Motorola Cellular Systems Division
Alan Willey, Motorola

The Personal Software Process: Downscaling the Factory?
Daniel Roy, Software Engineering Institute

Process Maturity Progress at Motorola Cellular Systems Division

Ron Borgstahl
Mark Criscione
Kim Dobson
Allan Willey

Motorola Cellular Systems Division
Arlington Heights, IL

email: {borgstal | criscion | dobson | willey }@cig.mot.com

Introduction

This year the Cellular Systems Division of Motorola submitted an application to the IEEE Computer Society for a Software Process Achievement Award. We placed second overall, with the Award going to our hosts, the Software Engineering Laboratory. In our application for the award we made public results of more than five years of effort we have been undertaking to improve our software processes.

Like many large software development organizations, we have experienced our share of customers who complain about product defects, failure to meet schedule commitments, and our inability to provide the software functionality they are demanding. By early 1990, the staff had come to recognize that the software processes in place were inadequate to meet our customer needs. Thus, in 1990 we began using the SEI Process Maturity Model (PMM) and Humphrey's *Managing the Software Process*[1] to help us define the requirements for a more mature software process. Our ultimate goals were (and still are) to improve:

- our customer's satisfaction,
- our product quality,
- our on-time delivery record, and
- our productivity.

In April of 1991, a team of SEI-trained assessors, both from SEI and from across Motorola, assessed our organization at Level 1. Then in late 1991 we were presented with a classic example of "requirements creep" when the SEI announced their first draft version Capability Maturity Model (CMM)[2] which was intended to replace the PMM. Careful review lead us to conclude that we had no choice but to adapt this more rigorous and detailed set of process requirements. We found to our delight that the software process architecture we developed, which was implicit in IEEE Std 1074-1991 "Standard for Developing Software Life Cycle Processes,"[3] was robust enough to meet the new CMM requirements. What needed attention were the "process specifications." These would have to be far more detailed to assure conformance to the CMM requirements. We had previously formed working groups to write process specifications for all processes, and now we began to identify the changes needed to meet the new CMM requirements. Next, we prioritized our efforts based on the CMM five-level model.

In June of 1993, after months of implementing this Software Process Improvement (SPI) Plan, we were re-assessed formally (using the PMM) at Level 2. More importantly, as more of our processes have begun to conform to the CMM requirements, we have begun to demonstrate significant measurable improvement in delivered product quality and on-time delivery, delivering more functionality to a more-satisfied customer, as accompanying data will support. Our data gathering activities have lagged behind other process changes, and key process measures were not routinely made before 1992, but we think that it is important to keep in mind that the data presented covering the last six quarters effectively represent results of process improvements underway since early 1991.

To support the Nomination of the SPI team at CSD a set of representative data was prepared. We presented data from a single product software development group representing about three hundred developers in our division. Since the submission of this application we have continued our efforts, and new data continues to demonstrate the benefits. We will review all of the data we have available to us at this time, which represents the time frame from the first quarter of 1992 to the end of the second quarter of 1994. Data from all projects completed by this product group and released to customers in that time frame are included. Six charts will be presented.

Figure 1

This figure shows our progress made in achieving

compliance with the requirements of the six Level 2 Key Process Areas (KPAs) named in the SEI CMM, Requirements Management (RM), Project Planning (PP), Project Tracking (PT), Subcontractor Management (SM), Quality Assurance (QA), and Configuration Management (CM).

An internally-developed procedure is used to assess compliance, and each development group conducts quarterly internal self-assessments.[4] The assessment procedure focuses on key practices described in the CMM, and compliance is contingent upon evidence of the presence of each key practice. The "percent compliance" described in this Figure is therefore the mean percent compliance of all of the key practices in each KPA which are evident to the assessment team. Outside team members from other development organizations and from the software quality assurance organization participate in these assessments to assure more-uniform and rigorous scoring.

The first round of these assessments was held in the third quarter of 1992, and the results of that assessment are compared to the current scores. The entire development organization was assessed at Level 2 using the PMM in June of 1993, but this development group had not yet achieved complete compliance with all of the requirements of the CMM at that time. However, since then significant progress has been made, and full implementation of all the key practices described in each KPA is now evident.

Figure 2

With the completion of our formal Self-assessment in June 1993, when we were rated at Level 2, the entire organization has moved forward with an initial assessment of our status with respect to the key practices found in Level 3 KPAs using our self-assessment procedure. The initial scores of this development group are presented in this chart. The initial conclusion one might draw from this chart is that the group is far from compliant with the requirements for Level 3. In view of our initial scores on the Level 2 Key Process Areas, however, we are confident that the group can be expected to make rapid progress toward compliance. Combined with the information presented in Figure 1, we can see that the group is in full compliance with Level 2 KPAs, and working on improvements on the Level 3 KPAs.

Figure 3

A customer survey is conducted regularly by an independent market research firm using a "Motorola Confidential Proprietary" survey questionnaire. In constructing this survey questionnaire "Key Drivers" have been identified which represent our effort to measure what our customers think is important. Each satisfaction survey measures our performance on these Key Drivers. Figure 3 compares our percent improvement in this product group for the Key Drivers which are concerned with software, in comparison to our performance in 1991. Since the survey contents and results are confidential, we have represented our progress by means of an index, with year-end 1991 results being "1," and year-end 1992 and 1993, and year-to-date 1994 being shown relative to that index quantity.

Figure 4

To explain Figure 4, some specific definitions are required.

Customer Found Defects are those post-release defects which are found by the customer. This does not include post-release defects found by Motorola internally, or defects of which customers have been notified before these customers find them.

Each customer found defect is recorded based upon the release in which it is found. A "window of opportunity" to find defects exists for each successive release. For a particular release, the first opportunity to find and report defects occurs at the time the first customer installs it. Defects in that release can be found by customers up to the time the last customer using that release retires it. Most releases are in use about 12 to 18 months. When a release is made in a particular quarter, and defects are reported against that release, the number of Customer Found Defects for all releases in that quarter is incremented. Over time, if additional defects against that release are reported, the quarterly total of defects for releases in that quarter is incremented. As releases are retired, since defects

can no longer be reported further against them, the total defect count becomes fixed. Our experience, like most software developers, is that most Customer Found Defects ever found are reported in the first quarter of use.

Delta KAELOC is the size of the added, deleted, and modified source code expressed in thousands of Assembly-Equivalent Lines of Code.This number is calculated based on a factor specific to each programming language used using the table provided by Capers Jones of SPR, Inc.

Total KAELOC is the total size of the released software expressed in thousands of Assembly-Equivalent Lines of Code. This number is calculated based on a factor specific to each programming language used.

Figure 4 demonstrates that in this time period the number of customer found defects has continued to decline, and that our most-recent releases are approaching 6 sigma quality.

Figure 5

Delay in delivery of promised software releases is a key contributor to customer dissatisfaction. In all of our product groups, release dates are forecast at the time of "project initiation" when the release project plan is approved and development begins. Figure 5 records for each release in a quarter how long after the forecasted release date the actual release occurred. Coincidentally, there has been one release per quarter for this product for the last two years.

Figure 5 shows a step-function improvement occurred in on-time deliveries between the releases in the second and third quarters of 1992. This came about primarily through better management controls in project planning and project tracking. Demonstrating that we are still a Level 2 organization, one release was delayed significantly in the second quarter of 1993 because of a delay in delivery of a vendor's code, and because some key staff members were temporarily reassigned to another project. In the fourth quarter of 1993 another release was delayed because of extended negotiations with a key customer on feature content for the release. This experience clearly highlights why both subcontractor management, project tracking, and requirements are key contributors to customer satisfaction. A note of explanation about the seeming lack of data for the first quarter of 1993. In fact, this release was exactly on time, thus the delay was zero months.

Figure 6

More and more functionality is being demanded by our customers, and with each new release we place more functionality into the customer's hands. Figure 6 demonstrates the extent to which the amount of new code (Delta KAELOC, as defined in the note to Figure 4) is growing at each release. In data not presented here we have measured that our productivity in terms of the number of lines of code produced by each software engineer has more than doubled in this time. Thus, while we have added staff, the staff has continued to increase the amount of code being delivered. The decline in the total number of new lines of code evident in 1994 results from the fact that this product development group is in the midst of a major product upgrade this year and only small, point releases have been made this year while most work continues to focus on the planned major upgrade to occur in the first quarter of 1995.

Returning to a topic mentioned in the note to Figure 4 we want to reiterate that even though we have increased the number of lines of code delivered with each new release by seven-fold, we are still seeing a significant decline in the number of customer-found defects in these releases. Stated simply, we are releasing more functionality to our customers, with higher productivity, and with fewer defects.

Summary

We believe that the key success elements are related to our recognition that Software Process Improvement (SPI) can and should be organized, planned, managed, and measured as if it were a project to develop a new process, analogous to a software product. In summary, we believe that our process improvements have come as the result of these key elements:

- use of a rigorous, detailed requirements set (CMM),
- use of a robust, yet flexible architecture (IEEE 1074),

- use of a SPI project, resourced and managed like other work, to produce the specifications and implement them, and
- development of both internal and external goals, with metrics to support them.

We have achieved significant, measurable results as a result of these efforts, and we want to share these findings with a broad industry audience. Our efforts may be viewed as unique in the sense that our business is entirely commercial and we have no customer pressure to adopt any particular paradigm for improvement, yet we selected the SEI Process Maturity Model and have successfully used the requirements of this Model to drive software process improvements. In a sense, we have validated this Model for change, and used it to substantially change our development processes and the customer's view of our product.

References

1. Humphrey, Watts S. *Managing the Software Process*, Addison-Wesley Publishing Company, Reading, MA, 1989.

2. Paulk, Mark C., et al. *Capability Maturity Model for Software, Version 1.1*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, February, 1993.

3. IEEE Std. 1074-1991 *IEEE Standard for Developing Software Life Cycle Processes*, Approved September, 1991. The Institute of Electrical and Electronics Engineers, Inc.

4. Daskalantonakis, Michael "Achieving Higher SEI Levels" in *IEEE Software*, July, 1994, p. 17.

MOTOROLA
Cellular Infrastructure Group

Presentation to:
Nineteenth Annual
Software Engineering Workshop

Allan Willey
Member, Technical Staff
Cellular Infrastructure Group
November 30, 1994

MOTOROLA
Cellular Infrastructure Group

Topics

- Introduction
- Our Experiences
- Results
- Summary
- Lessons Learned

MOTOROLA
Cellular Infrastructure Group

Congratulations to the SEL!

- Winner of the IEEE Computer Society Software Process Achievement Award for 1994
- Motorola's Cellular Systems Division (CSD) was "First Runner Up"
- We are the "Avis" of Process Achievement this year, and "trying harder."

19th SEW
November 30, 1994

ALW – 2

MOTOROLA
Cellular Infrastructure Group

Motorola Cellular Systems Division (CSD)

- Approximately 1,000 in the R & D Division
- Four locations:
  - Arlington Heights, IL, USA
  - Cork, Ireland
  - Tel Aviv, Israel
  - Ft. Worth, TX (the fourth country)
- Data presented here is for the EMX 2500 Switch Software Development Group (~300 staff)

19th SEW
November 30, 1994

ALW – 3

MOTOROLA
Cellular Infrastructure Group

CSD Key Events

- Motorola has a corporate software engineering goal to achieve SEI Level 3 by YE'95
- CSD had first SEI Self-assessment in Nov.'90
  - Level 1 (are you surprised?)
- Second Self-assessment, June '93
  - Level 2 (phew! Made it)
- Third Self-assessment scheduled *next week*

19th SEW
November 30, 1994

ALW-4

MOTOROLA
Cellular Infrastructure Group

CSD Key Strategy Decisions

1. Use SEI 5-level Model for "Requirements"
2. Use IEEE 1074 for the "Design"
3. Implement a "Process Improvement" Project

19th SEW
November 30, 1994

ALW-5

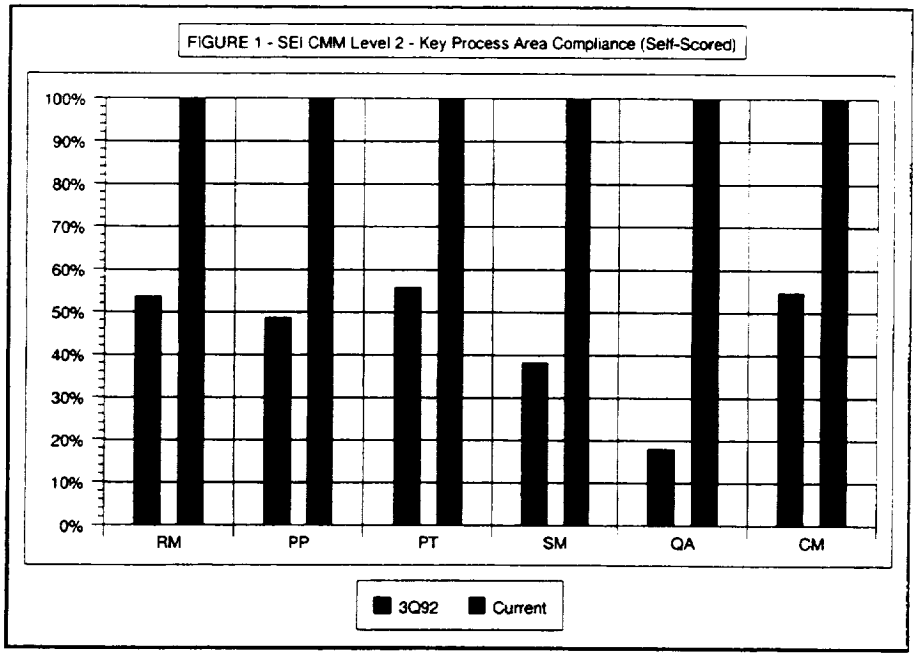
MOTOROLA
Cellular Infrastructure Group

Summary of Results

- Progressive improvements in "Process Maturity"
- Continuous improvements in quality, productivity, on-time delivery, and customer satisfaction
- "Quantum leap" in the quality of work life

19th SEW
November 30, 1994

ALW – 6



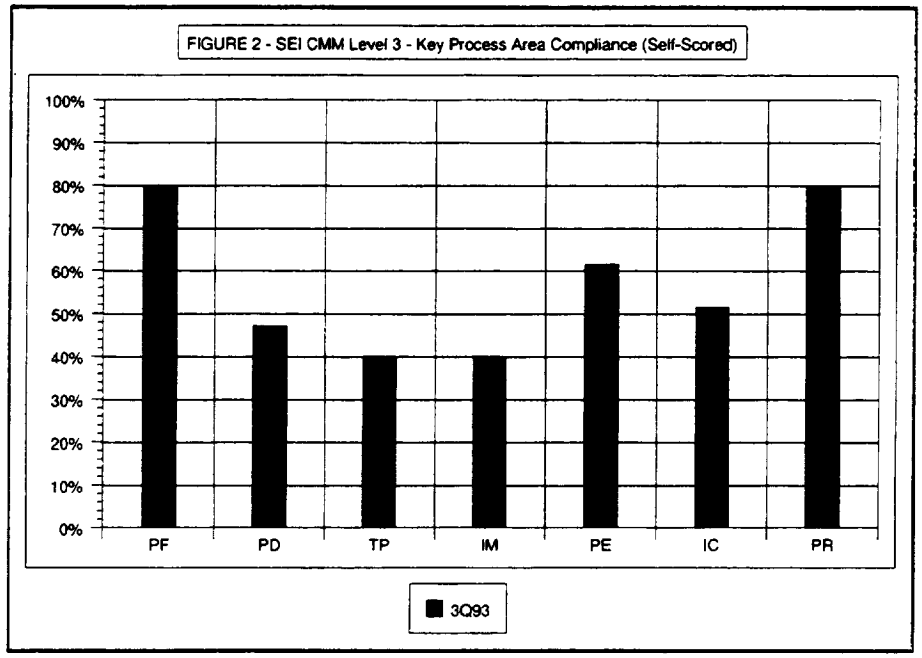


FIGURE 2 - SEI CMM Level 3 - Key Process Area Compliance (Self-Scored)
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
PF
PD
TP
IM
PE
IC
PR
3Q93

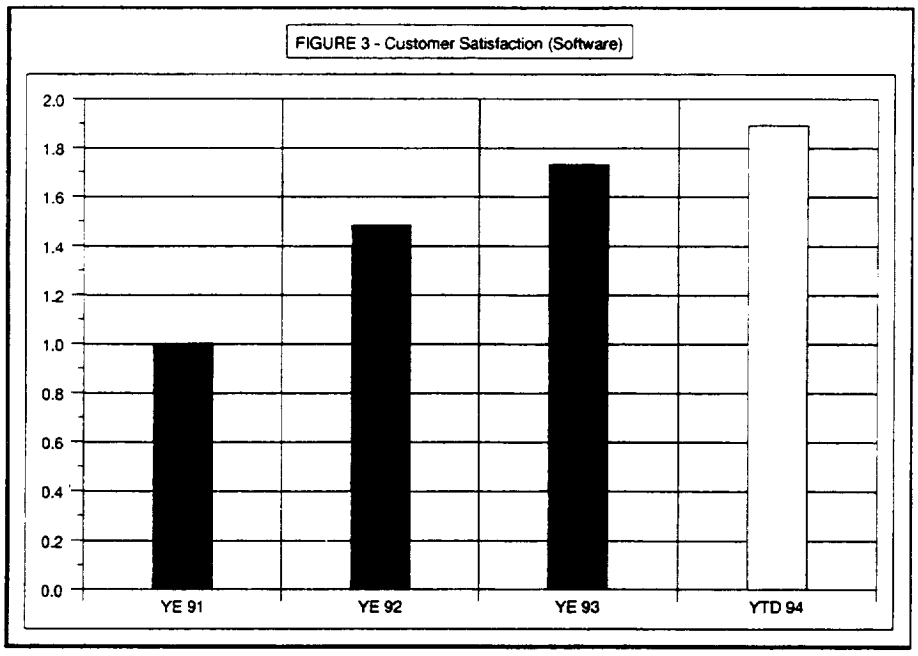


FIGURE 3 - Customer Satisfaction (Software)
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
YE 91
YE 92
YE 93
YTD 94

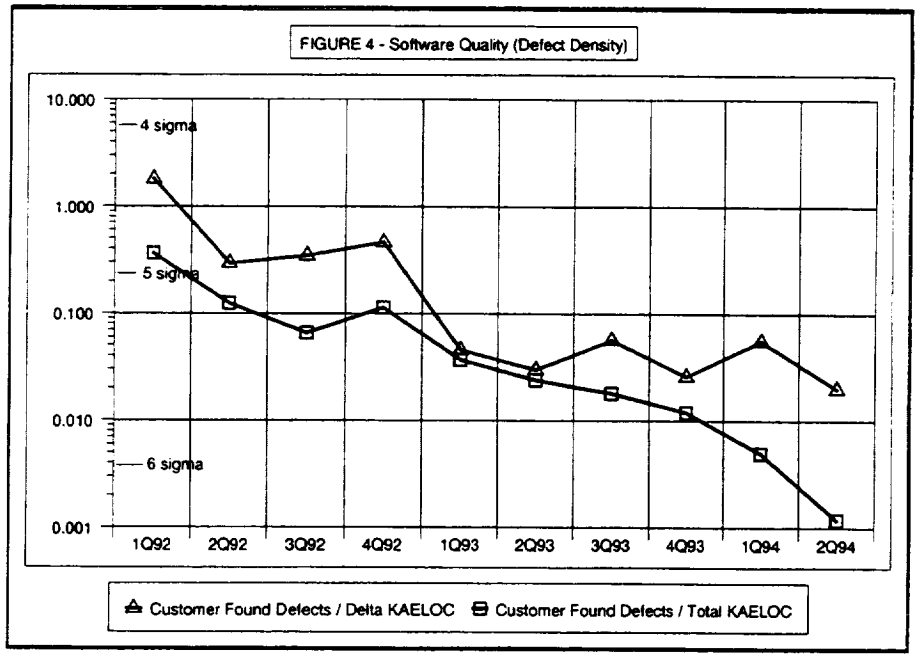


FIGURE 4 - Software Quality (Defect Density)
10.000
1.000
0.100
0.010
0.001
4 sigma
5 sigma
6 sigma
1Q92
2Q92
3Q92
4Q92
1Q93
2Q93
3Q93
4Q93
1Q94
2Q94
Customer Found Defects / Delta KAELOC
Customer Found Defects / Total KAELOC

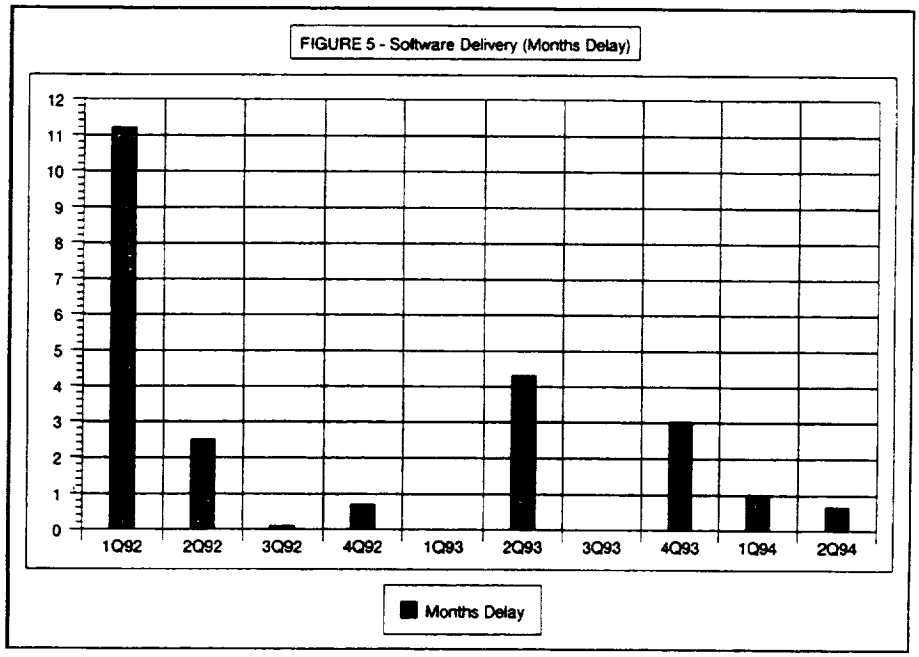
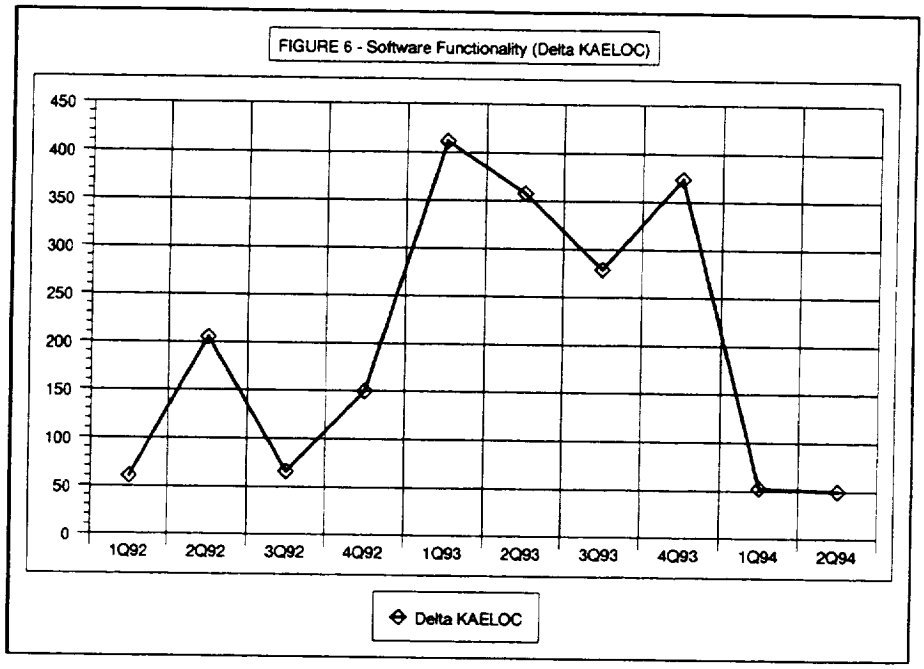


FIGURE 5 - Software Delivery (Months Delay)
12
11
10
9
8
7
6
5
4
3
2
1
0
1Q92
2Q92
3Q92
4Q92
1Q93
2Q93
3Q93
4Q93
1Q94
2Q94
Months Delay



MOTOROLA
Cellular Infrastructure Group

Lessons Learned

- "Plan your work"--in this case Process Improvement
- "Work your plan"--in this case the Process Improvement Project Plan
- This Project has:
  - Requirements Specifications
  - Design Architecture
  - Implementation Phases
  - Verification and &Validation Phases

19th SEW
November 30, 1994

ALW - 7

The Personal Software Process: Downscaling the factory?

Daniel M. Roy
Software Technology, Process and People (STPP)
20 Forest Rd. Bradford Woods, PA 15015
(412) 934 0943 E-mail: dmr@sei.cmu.edu
(Visiting scientist, SEI)

Abstract: It is argued that the next wave of software process improvement (SPI) activities will be based on a People-centered paradigm. The most promising such paradigm, Watts Humphrey's Personal Software Process (PSP) is summarized and its advantages are listed. The concepts of the PSP are shown to also fit a down-scaled version of Basili's experience factory. The author's data and lessons learned while practicing the PSP are presented along with personal experience, observations and advice from the perspective of a consultant and teacher for the Personal Software Process.

1 Toward a People-centered SPI paradigm

The Capability Maturity Model (CMM) and CMM-based SPI paradigms have had a profound impact on the organizational practices within the software industry [Herbsleb-94]. Other SPI paradigms such as the experience factory have been demonstrating the value of experiment based software improvement for over 15 years [IEEE-94]. In spite of these progress, we technologists, process advocates and other change agents still have to fight an entrenched and pernicious resistance.

To better ascertain what to do about this, we must understand where we have been and where we want to go next. As Basili puts it in [Basili-89]:

> *'We have evolved from focusing on the project, e.g. schedule and resource allocation concerns, to focusing on the product, e.g. reliability and maintenance concerns, to focusing on the process, e.g. improved methods and process models'*

However, addressing the practitioner's resistance from healthy skepticism to outright obscurantism is not a technical problem; it is a human concern. Perhaps accelerated progress requires that we now continue the evolution by focusing on the People, e.g. individual education and practices based on individual self improvement.

Major relatively new concepts such as the CMM or the experience factory are both intellectually satisfying and daunting to practice at the individual level. As a programmer, I may well understand the importance of the practices of the subcontract management KPA while at the same time failing to relate to any of them in my individual work. As a reuse technologist, I may be totally convinced that my company should operate as an experience factory while at the same time having no idea how to incorporate the concepts in my day to day practice.

Conversely, I may be highly skeptical of 'their' SCEs, 'their' pilot project, and God knows what other latest fad. I will remain unconvinced until 'they' show me that it will really work for me. I may have heard good things about clean room, I may even have watched a convincing presentation at the Software Engineering Workshop about it. If I have never personally experienced it, it will remain alien to me, something even vaguely frightening that I will keep resisting. In the word of a most famous (and anonymous) Chinese proverb:

> *'I hear and I forget, I see and I remember, I do and I understand'*

A more personal and more practical approach to software improvement where the individual practitioner learn by doing, may be needed to accelerate the transition of better engineering practices throughout our organizations.

2 The Personal Software Process

As students, we typically practice on toy problems in programming language classes. Our ad hoc processes are sufficient to produce moderate size programs quickly and get a passing grade. As programmers we quickly discover that these student practices do not scale up but what can we do?. The product must be out the door if we want to work on the next one. There is very little time to experiment with something unproven.

The personal software process was developed by Watts Humphrey to indoctrinate students (in university and industry alike) in the use of large scale methods based on the CMM. To quote Watts in [Humphrey-95], the PSP...

> *'... scales down industrial software practices to fit the needs of small scale program development. It then walks you through a progressive sequence of software processes that provide a sound foundation for large-scale software development'*

Using fairly simple and well proven engineering principles, the PSP student plans his work, enacts a well defined process, building the product while gathering data, and performs a post mortem that seeds the next improvement cycle. This personal approach to software improvement offers the following advantages:

- By having to adhere to more disciplined practices, students learn a lot about process, engineering, and software improvement. Most becomemotivatedto learn even more about their field
- By gathering their own private data, students quickly build a significant experience base which allow them to set new goals, perform the next experiment and check the results against the goals
- Since the data is personal and private, PSP practitioners need no convincing from anyone about the value of a process step or a technology. They know whether it works for them or not based on their own quantitative results
- Armed with their own productivity and quality statistics, practitioners of the PSP are better able to make commitments they can meet. They can also better resist unreasonable commitment pressures

The PSP course leads the student to the gradual application of software engineering discipline through a set of 10 assignments:

1. Average and standard deviation using linked list
2. Physical line counter
3. Object LOC counter (build on 2)
4. Linear regression using linked list (build on 1)
5. Standard distribution (integration by the method of Simpson)
6. Linear correlation (build on 5)

7. Confidence intervals (build on 5 & 6)
8. Sorting a set of numbers in ascending order
9. Performing statistical fit tests on the above data
10. Computing muti-linear regression coefficients (by solving a system of linear equations)

These simple exercises were found to have the following advantages:

- Simplicity without being trivial.
- Fostering reuse and good object oriented development practices
- Gradually building a small PSP support toolset

The PSP data shown on the transparencies was collected during Watts Humphrey's Spring 94 course for the Master of Software Engineering at CMU.

3 Personal data, experience, and lessons learned

Several PSP reports have to be written as part of the course detailing:

- Evolution of size and time estimates accuracy
- Pareto charts and checklist for defects
- Defect injection and removal trends
- Cost per defect type and injection/removal phases
- Process development process for PSP reports
- Detailed process analysis such as A/F ratio
- Lessons learned
- Future steps

The large number of graphs could not be reproduced here or even shown during the talk. Watts Humphrey's data analysis diskette (which can be obtained with the book [Humphrey-95]) includes an optimum set of Excel templates and macros for PSP data analysis. I found it very useful to track my progress and accelerate the routine of the post mortem analysis.

A central part of my talk dealt with the application of the concepts of experience factory to my PSP results. The experience gathered can be summarized as follow:

- The accuracy of my time and size estimates improved from +-40% to +-20% over the 10 assignments of the PSP.
- The PSP linear regression model helped me increase the accuracy of my size estimates. The multi linear coefficients computed by program 10 offer great potential to similarly increase the accuracy of my time estimates.
- The percentage of development time spent compiling decreased from 15% to 5%.
- My productivity during the development phase remained at 20 LOC/hr.
- I made a humiliating number of syntax errors with a language I know well until I truly inspected my code BEFORE compiling it.
- My error injection rate decreased from 180/kLOC to 30/kLOC and from 4 defect/hr to less than 1 defect/hr.
- From assignment 4 on, the sum of my code reuse and code developed for reuse stayed at about 80%.
- Defect fix cost varied from 1 min/defect to 8 min/defect depending on phase injected/removed.
- The process development process I enacted to develop a report development process for my PSP experience reports was an overkill. But I learned a lot trying that hard.

Building on this experience, I have applied the GQM paradigm to the definition of my next process improvement steps:

- Reduce my error injection rate to less than 20 defects/kLOC
- Improve my error detection processes
  - Keep design and code inspection yields above 50%
  - Keep formal pre-compile inspection yield above 80%
  - Strive for zero compile error
  - Improve my testing process to a yield over 50%
- Keep containing costs
  - Keep personal and informal review rates above 200 LOC/hr
  - Keep formal inspection rates above 100 LOC/hr
- Increase reuse
  - Either assemble 80% of the software out of reusable components
  - Or make reusable components out of at least 50% of the new code
- Formalize the experience gathered with the PSP by applying experience factory concepts

4 Teaching the PSP

SEI has already conducted one 'Train the trainer' course In Pittsburgh from October to December 1994. I taught the 2 lectures on design in that occasion. Besides the usual lessons learned from our own lectures, I think all instructors agreed that:

- The PSP is not your usual "teach and run" course
- Serious commitment is necessary from both student and sponsor
- A qualified instructor is necessary to get long term results

The PSP is about behavioral change. It is not a typical lecture course. It is a 200 hr intensive educational experience. The lectures are but the tip of the iceberg. The instructor must spend a significant amount of time tutoring the student in the correct implementation of the organization's process. The students don't just sit there either, they write working programs. These programs have to be reviewed and corrected. The process must be analyzed and feedback must be given. The PSP is more like a complete training program (in the sense of the CMM level 2 KPA) and typically spans 20 weeks. Strong commitments are necessary:

- from the student to honestly work the exercises, improve his process, and to finish the course
- from the sponsor to allow the time necessary for the lectures and for part of the implementation of the programs (typically shared 50/50 between sponsor and student)

Best resuts are seen when the sponsor treats the PSP assignment as any other (assuming correct project tracking and oversight practices). This means that the student's assignments are integral part of the workday and are part of his deliverables.

The PSP also requires a dedicated and qualified instructor with demonstrated programming and software management experience. Based on historical data, the effort necessary to correctly teach the PSP is roughly:

- Lecture preparation: 2-4hrs/lecture
- Tutoring: 5-10 hrs/student
- Program & process analysis: 2-10 hrs/student

Anything less has a great chance of failing to make a lasting difference in the disciplined, quality-driven individual practices demonstrated in the PSP course.

5 Conclusions

The PSP gives me the opportunity to improve the quality of the software I produce by offering a framework for objective measurement and improvement of my practices. However, the accuracy and the consistency of the data gathering process is paramount. Watts made this point very clear throughout the course. Nevertheless, it took me quite a while to truly understand why. I believe that a strict data inspection process should be enacted and particularly strongly enforced at the beginning of a PSP course to ensure that all students start on the right foot. I also believe that the postmortem phase should be expanded to include the systematic analysis and archiving of lessons learned with the assignment at hand. I have modified my own PSP accordingly.

I believe that the PSP is not only about scaling down the CMM. It can also be seen as a scaled down experience factory. It is because the PSP encompasses such an elegant synthesis of large scale methods that it will power the next wave of software practice improvement.

By practicing the PSP, I have learned a great deal about enacting, improving and even developing personal processes. I have carried the very simple principles of the PSP and the process development methodology described in chapter 13 of [Humphrey-95] to other processes:

- The organization of my work day
- A consulting personal process
- A process to perform Rate Monotonic Analysis
- A family of processes to write papers and reports.

These have been very exciting first steps.

6 Bibliography

[Basili-94] Victor Basili et. al., 'The Experimental Paradigm in Software Engineering', Experimental Software Engineering issues, Springer-Verlag, 1994.

[Herbsleb-94] James D. Herbsleb and David Zubrow, 'Software Process Improvement: An Analysis of Assessment Data and Outcomes', Technical report CMU/SEI-94-TR7, September 1994.

[IEEE-94] IEEE, 'IEEE Computer Society Award for Software Process Achievement, Nomination of 1994 Award Winner', Information bulletin, May 1994.

[Park-92] Robert E. Park, 'Software Size Measurement: A Framework for Counting Source Statements', Technical report CMU/SEI-92-TR-20, September 1992.

[Humphrey-95] Watts S. Humphrey, 'A discipline for Software Engineering', Addison Wesley, January 1995.

Carnegie Mellon University
Software Engineering Institute

The PSP: Downscaling the factory?

Daniel M. Roy, SEL workshop, December 1994

Software Engineering Institute
Carnegie Mellon University
Pittsburgh PA 15213

Sponsored by the U.S. Department of Defense

Carnegie Mellon University
Software Engineering Institute

Agenda[1]

The Personal Software Process (PSP)

Some preliminary results

SEL experience factory

Scaling down the models: The experience workshop

1. Preliminary work offered for informal review.

The Personal Software Process

Programming language class practices do not scale up

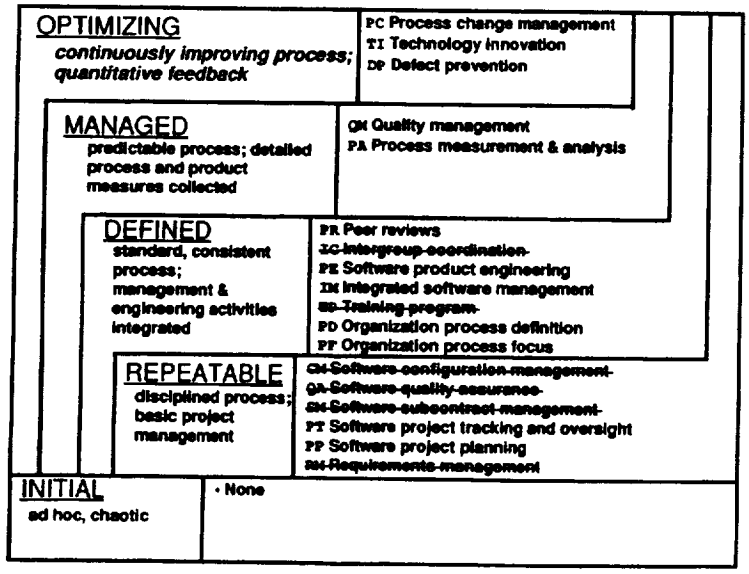
Corporate wide efforts encounter increasing resistance on the way down.

The PSP:

> *"Scales down industrial software practices to fit the needs of small scale program development. It then walks you through a progressive sequence of software processes that provide a sound foundation for large-scale software development."*[1]

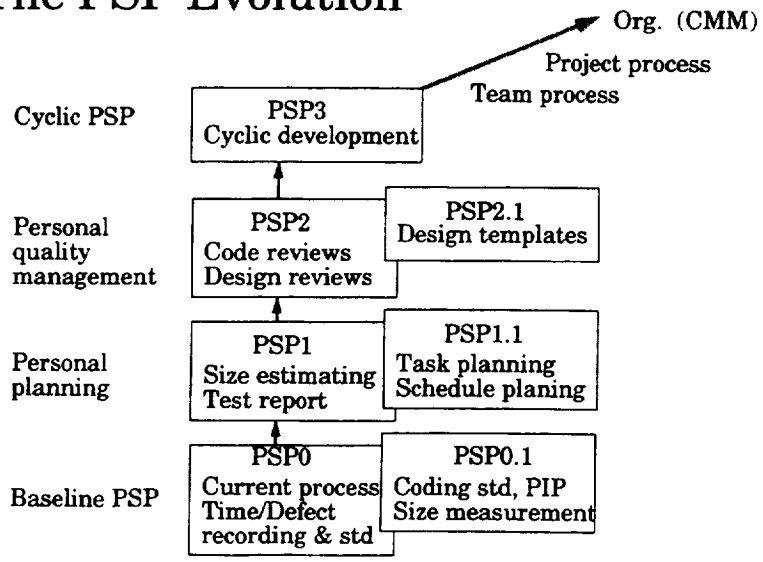
1. Watts Humphrey, "A discipline of software engineering", Addison Wesley, December 1994.

KPAs scaled down for the PSP



Carnegie Mellon University
Software Engineering Institute

The PSP Evolution[1]

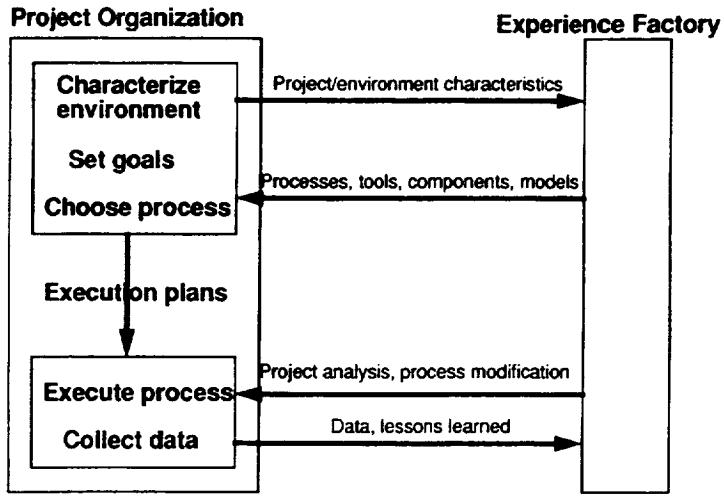


3

1. Watts Humphrey, "A discipline of software engineering", Addison Wesley, December 1994.

Carnegie Mellon University
Software Engineering Institute

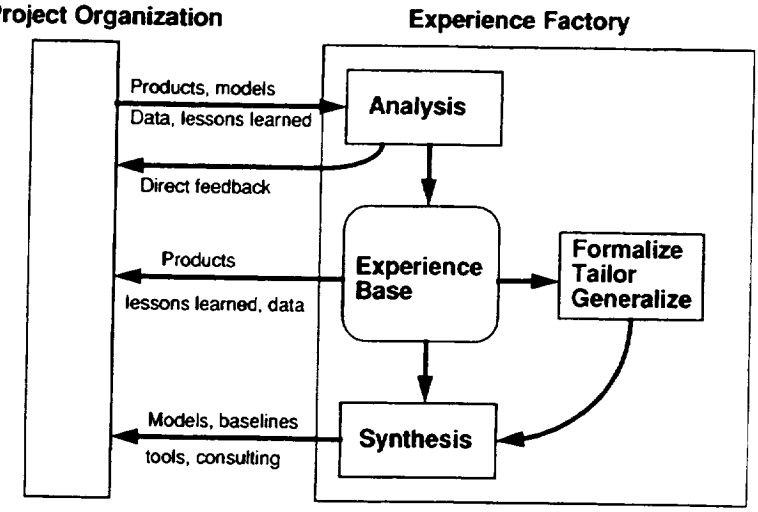
The Experience Factory context[1]



4

1. From "The Experimental Paradigm in Software Engineering, Experimental Software Engineering Issues", Rombach, Basili, Selby, Springer-Verlag

The Experience Factory structure[1]



5

1. From "The Experimental Paradigm in Software Engineering, Experimental Software Engineering Issues", Rombach, Basili, Selby, Springer-Verlag

The PSP assignments as experiments

Goal:

- **Actual staff-hours will be within 20% of estimates 80% of the time.**

Questions:

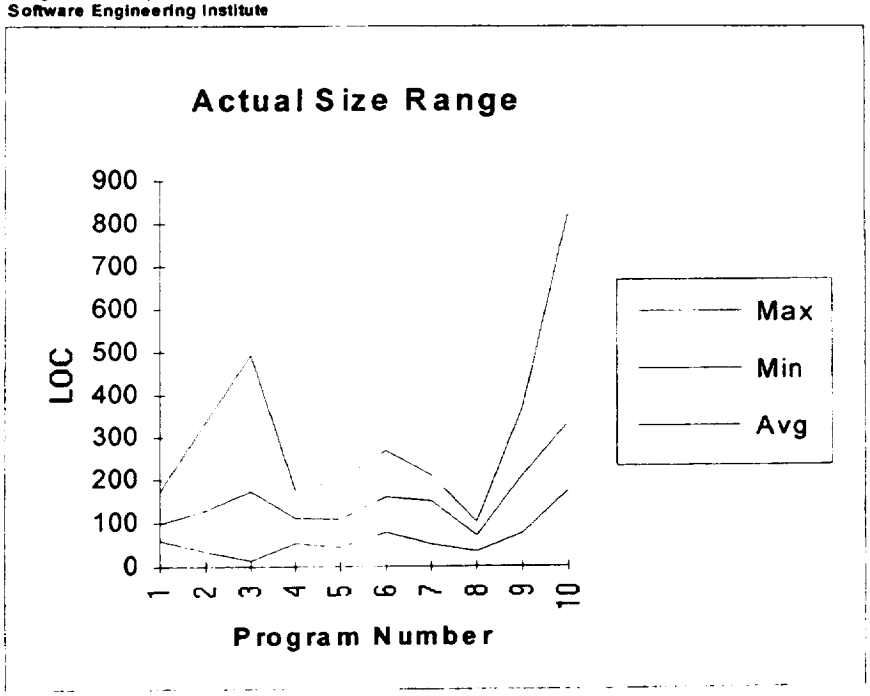
- **How do I predict my effort now?**
- **How do I measure the actual effort?**
- **How do I track actual against estimates?**
- **What is the dispersion now?**
- **If I had a data base of these, I could get statistics**

Metrics:

- **Estimate in mn before, measure actuals during**
- **Compute linear regression and confidence intervals from the data base. Accuracy is given by stats.**

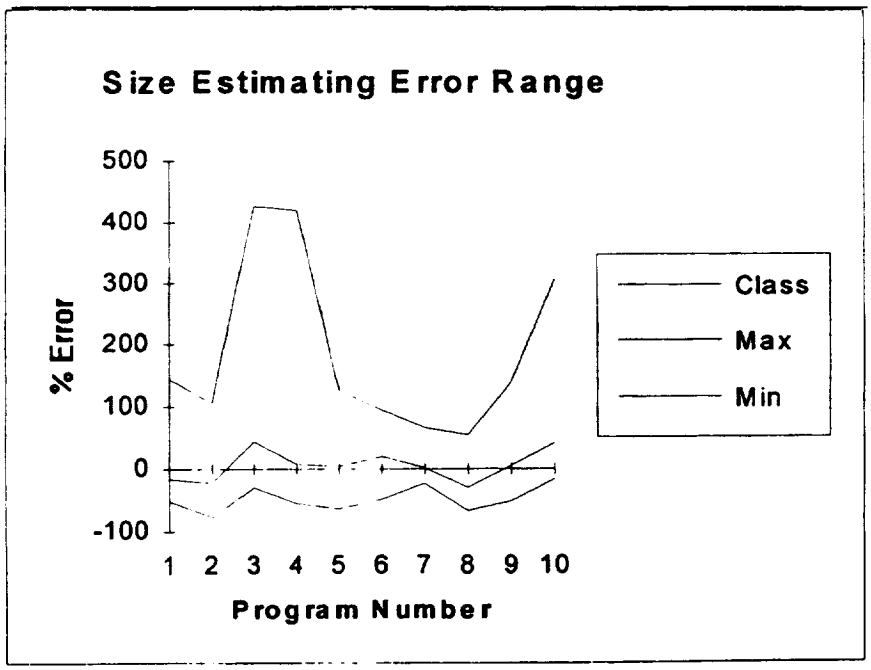
6

Carnegie Mellon University
Software Engineering Institute



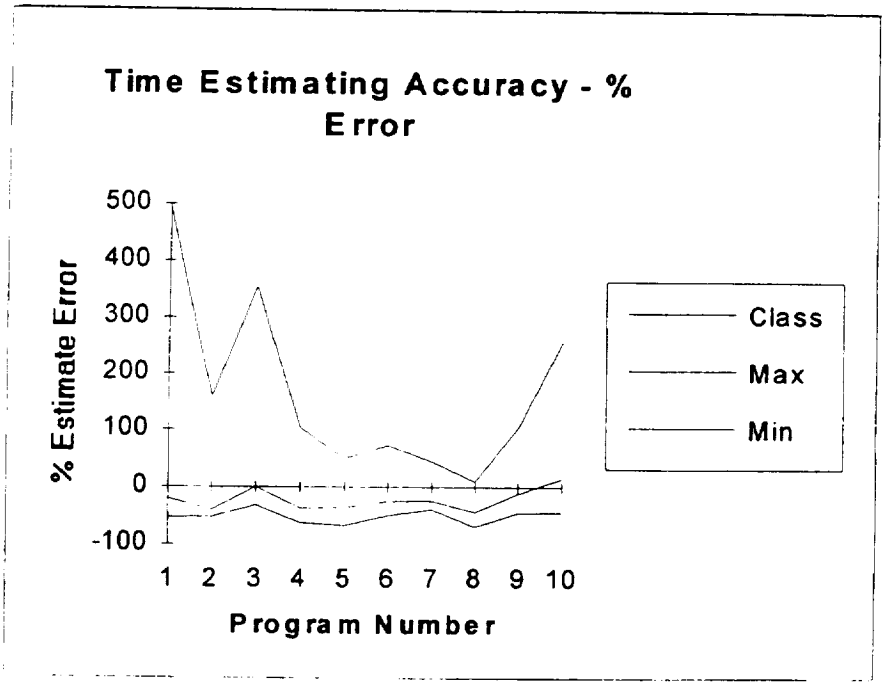
9

Carnegie Mellon University
Software Engineering Institute



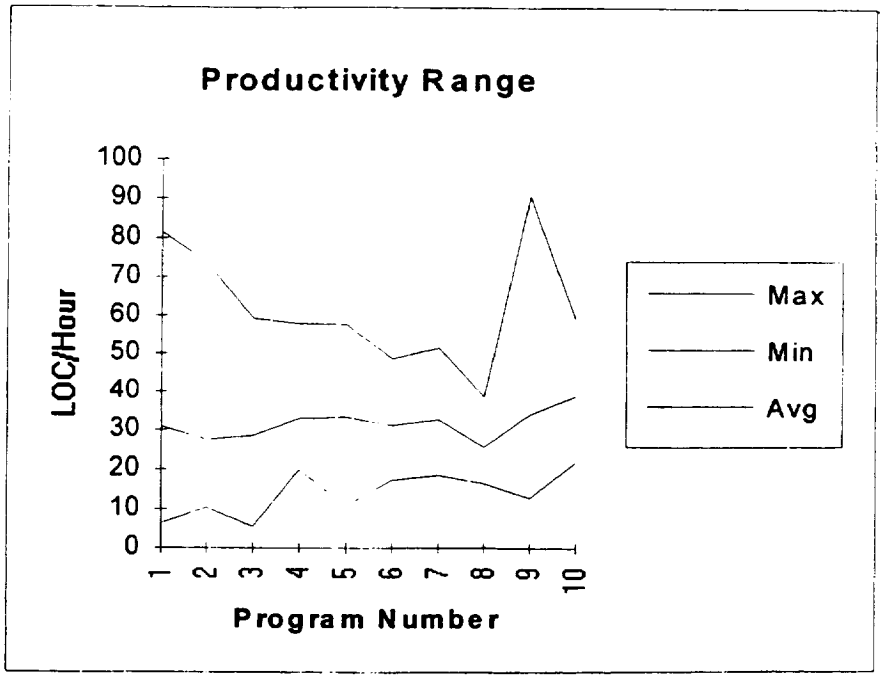
10

Carnegie Mellon University
Software Engineering Institute



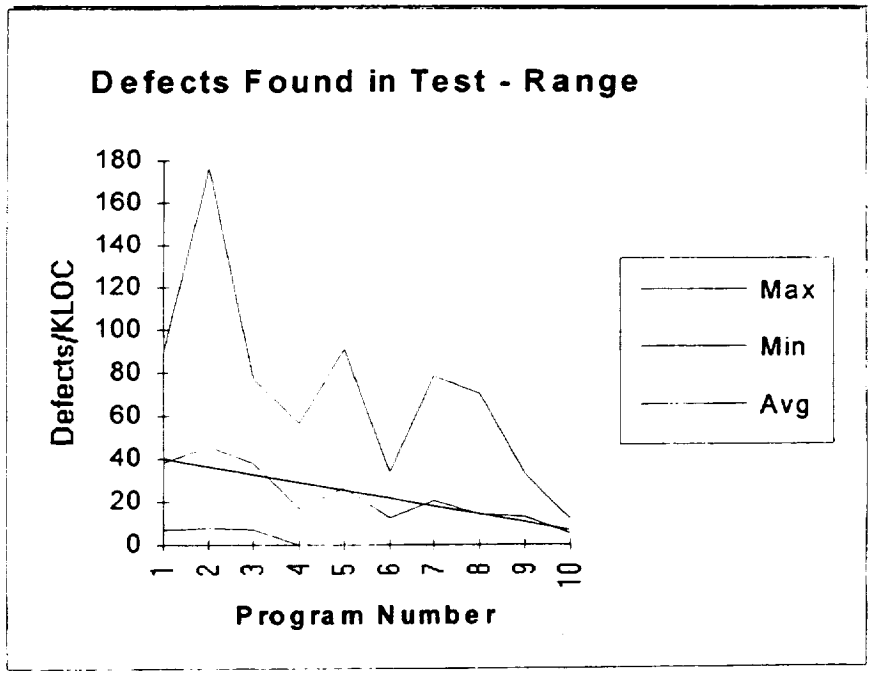
6

Carnegie Mellon University
Software Engineering Institute



21

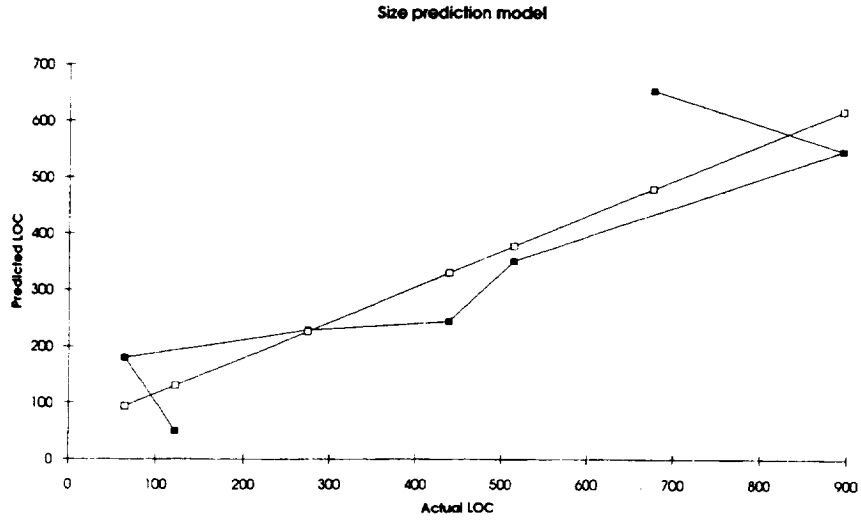
Carnegie Mellon University
Software Engineering Institute



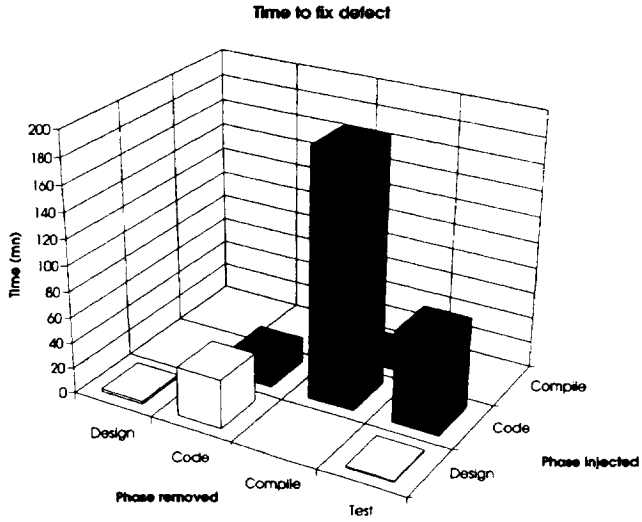
20

Carnegie Mellon University
Software Engineering Institute

Size prediction model (dmr data)

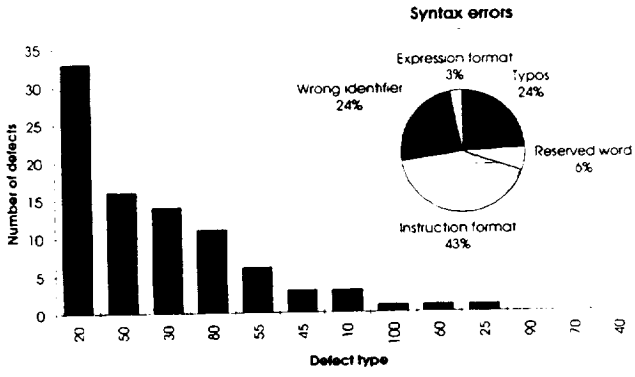


Cost of error (dmr data)



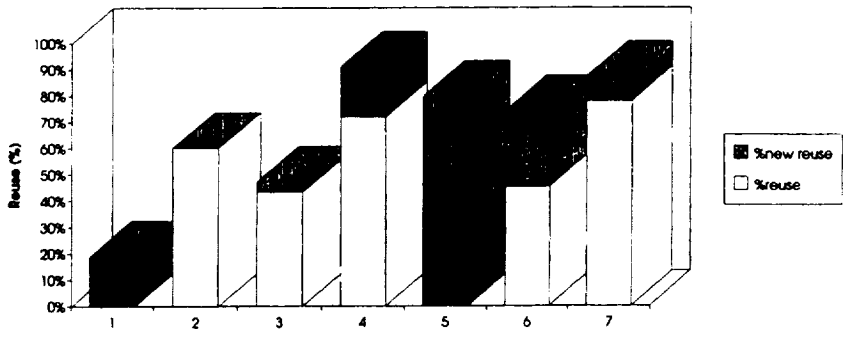
9

Defects analysis (dmr data)



10

Reuse trends (dmr data)



11

Ada PSP: Some experience artifacts

"I hear and I forget, I see and I remember, I do and I understand"[1]

A lot of very useful process data:

- **predicted and actual time per phase**
- **error classes and distribution**
- **linear regression models for size and cost estimates**
- **trend analysis graphs on all of the above**
- **post mortems and reports as experience base**
- **a deeper understanding of PSI that carries beyond software development**

A lot of new goals and ideas to try next

12

1. Anonymous Chinese proverb

Some of my next goals:

Reduce my total defect injection rate to less than 20 per KLOC.

Optimize my set of inspection processes to reduce their cost to less than 1 inspection staff-mn per SLOC while keeping yield above 80%

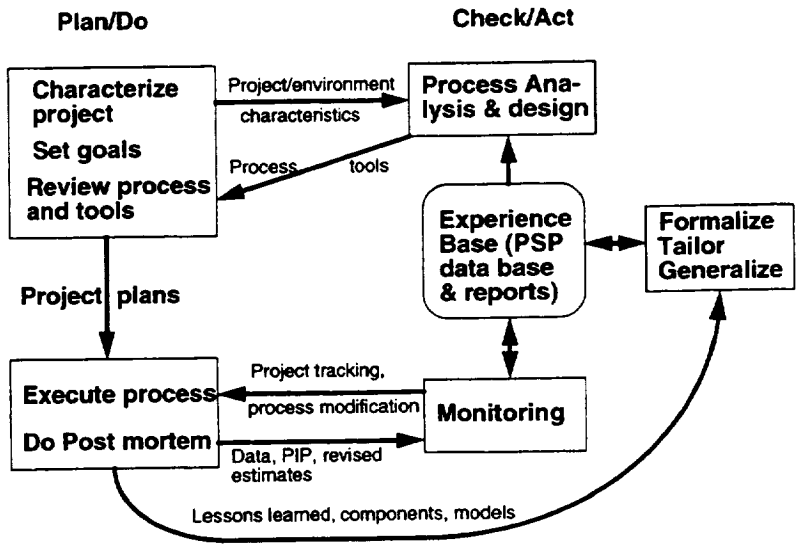
Either build with reuse (at least 80% of total SLOC) or build for reuse (at least 50% of the new code is reusable)

Revisit the PSP in the light of the CMM and ISO 9000-3

Recast the PSP in the experience factory mold

13

PSP: The experience workshop



14

Carnegie Mellon University
Software Engineering Institute

Conclusion

The PSP represents an elegant synthesis of proven concepts (CMM, experience factory) scaled down to the individual level.

Preliminary PSP results are encouraging. Team data is needed.

Until now:

> *"We have evolved from focusing on the project, e.g. schedule and resource allocation concerns, to focusing on the product, e.g. reliability and maintenance concerns, to focusing on the process, e.g. improving methods and process models"*[1]

Future progress may well hinge on focusing on the People.

---

15 1. From "Software Development: A Paradigm for the Future", Victor R. Basili, Proc. 13th Int'l Computer Software & Applications Conf. Orlando FL, Sep 89

Carnegie Mellon University
Software Engineering Institute

PSP Status

The PSP was developed by Watts Humphrey

Several industrial organizations are now introducing PSP methods (DEC, HP, TI) with encouraging results

SEI is offering train the trainer courses

Several universities are teaching the PSP (CMU, U. of Mass., Howard U., Embry-Riddle U., McGill, and others)

The textbook "A Discipline for Software Engineering" and support diskette are available from Addison Wesley.

16

Questions

For more information or off-line discussion contact:

Daniel Roy
20 Forest Rd
Bradford Woods PA 15015
(412) 934 0943
dmr@sei.cmu.edu

16

Session 3: Certification

Applying Program Comprehension Techniques to Improve Software Inspections
Stan Rifkin, Master Systems Inc.

An Experiment to Assess the Cost-Benefits of Code Inspections in Large-Scale Software Development
Harvey Siy, University of Maryland

A Process Improvement Model for Software Verification and Validation
Jack Callahan, NASA Independent Software Verification and Validation Facility

Applying Program Comprehension Techniques to Improve Software Inspections

Stan Rifkin
Master Systems Inc.
P.O. Box 8208
McLean, VA 22106
sr@seas.gwu.edu

Lionel Deimel
1408 Navahoe Dr.
Pittsburgh, PA 15228
bmtm05a@prodigy.com

Abstract: Software inspections are widely regarded as a cost-effective mechanism for removing defects in software, though performing them does not always reduce the number of customer-discovered defects. We present a case study in which an attempt was made to reduce such defects through inspection training that introduced program comprehension ideas. The training was designed to address the problem of understanding the artifact being reviewed, as well as other perceived deficiencies of the inspection process itself. Measures, both formal and informal, suggest that explicit training in program understanding may improve inspection effectiveness.

The software technical review is a widely -recommended mechanism for software defect removal. Such reviews go by many names—inspections, Fagan-style inspections, code reviews, peer reviews, formal reviews—and exhibit significant variations among organizations [Fagan, Freedman, Gilb]. All such review methods rely on the self-evident notion that software professionals are likely to find defects in software if they actually *look* at the products they produce. A software technical review is a meeting—along with its preparation—in which a group of software professionals (peers) does exactly that. Types of reviews are distinguished from one another by the rules governing how that examination takes place and how it relates to the overall software development or maintenance process. Impressive claims are made for the efficacy of reviews [Humphrey].

What follows is a case study in which developers were given, along with traditional (and non-traditional) instruction, explicit instruction in program comprehension concepts and techniques. The case study suggests that software engineers often have poor strategies for understanding the artifacts they are called upon to review and that providing training in comprehension skills can improve their performance significantly.

A Training Opportunity

One of the authors (Rifkin) was engaged by a manufacturing firm that we will call Widget, Inc.[1] Widget management, having read the literature on software inspections, had expected the introduction of this practice to produce a significant decline in customer-discovered defects. The anticipated decline had not occurred, however, either in the number or percentage of defects identified by customers.

---

[1] The firm wishes to remain anonymous and does not want to divulge raw data on defects, which it considers proprietary. The data in this paper are presented in a manner intended to respect those wishes.

Previous engagements had investigated the common experience that, while the percentage of defects discovered by testing prior to product release declines precipitously after the introduction of inspections, customer-discovered defects show no significant decrease. This is not to say that inspections are not useful or cost-effective. In large measure, however, they seem to identify defects that might otherwise be found using a more expensive method—testing—rather than reduce the overall number of defects in released software.

We had hypothesized that introducing inspections often had had little effect on reducing customer-identified defects because, although reviewers were being thoroughly trained in the group aspects of the inspection process, they were being given little guidance as to how to precisely carry out their preparatory study of work products in the privacy of their own offices. It was generally assumed that reviewers knew how to look for defects, any data to the contrary notwithstanding. This hypothesis had led to the development of a training program on those previous engagements that was intended to be more comprehensive, and this enhanced training was brought to Widget. It incorporated an introduction to program comprehension based on the Deimel and Naveda report from the Software Engineering Institute, "Reading Computer Programs: Instructor's Guide and Exercises" [Deimel90].

Widget, Inc.

Widget is a large-scale manufacturing company. One particular section produces software for engineering computations. There used to be two groups in this section, which we will call Group 2 and Group 3. Each group comprised about 30-35 software professionals who regularly performed inspections. Group 2 had been trained in performing inspections by Michael Fagan [Fagan], and Group 3 had received training from Tom Gilb [Gilb]. Group 2 had received training about five years prior to our engagement, and Group 3 had received training about three years prior. The two groups had developed a number of large FORTRAN programs, and their current duties predominantly involved maintaining and enhancing those programs. Another unit, which we will call Group 1, was about 18 months old. It, too, comprised 30-35 professionals, nearly all of whom had worked previously in one of the two other groups. Group 1 maintained and enhanced a suite of computer-aided design and computer-aided manufacturing programs written in FORTRAN, C, and several script languages. The source code of some of the programs had been purchased. Staff turnover in all three groups was low.

The customers (users) of the software for which the section was responsible were Widget engineers. Although these engineers were organized into a number of separate units, they constituted a substantially homogeneous customer base for all three development groups. Each major customer unit has one or two representatives responsible for collecting issues (including bugs and desired features) and negotiating their resolution with the developers.

Some Group 1 members had received inspections training from Fagan and some from Gilb. This difference in backgrounds and the perceived incompatibility of the Fagan and Gilb methods had inhibited their use of inspections. Group 1 management sought to routinize inspections through training that fostered a common understanding of inspections. After some discussion with that management, however, reduction of customer-discovered defects became the dominant goal of the proposed engagement. It was necessary to define a single inspection process for Group 1, of course; moreover the members of Group 1 were already "sold" on inspections and did not need specific encouragement to perform them.

The Training Workshop

The normal Master Systems 1½ day inspections training workshop was presented at Widget for the members of Group 1, with half the group attending each of two offerings. The workshop followed this syllabus:

Day 1 (full -day)

- DEFINITION OF INSPECTIONS, EXPECTED BENEFITS: Description of the "common" software inspection process and its documented benefits.
- INTRODUCTION TO THE INSPECTION PROCESS: Details of the usual steps before, during, and after an inspection defect collection meeting.
- INTRODUCTION TO READING COMPREHENSION: Discussion of how we come to understand what we read and how that process can be made more effective.
- DEVELOPMENT OF THE INSPECTION PROCESS: What are the requirements for inspections? What is a process that will fulfill those requirements? Two types of work products are chosen to be inspected.

In Between (outside work done by participants)

- CONTINUED DEVELOPMENT OF THE INSPECTION PROCESS: Participants, having each been assigned to one of three groups, meet either to complete a full description of the inspection process or to develop checklists for each of the two work product types.
- SELECTION AND STUDY OF ARTIFACTS: The groups responsible for composing checklists select existing artifacts for practice inspections. Each workshop participant reviews one of these privately, in preparation for the inspections on Day 2.

Day 2 (half day)

- PRACTICE INSPECTIONS: Inspections of the selected artifacts allow participants to practice taking the four rôles of producer, moderator, recorder, and reviewer using the selected artifact.
- DEBRIEF: Discussion of what has been learned and how it can be applied on the job.

Days 1 and 2 were a week apart. Approximately two hours of the instruction time on Day 1 were devoted to understanding programs. This material was to be applied during the In Between time, when the artifacts selected were studied privately by each participant for approximately two hours.

Much of the material on program comprehension was taken from or suggested by the report by Deimel and Naveda. (The report makes a case for the importance of teaching program reading skills, reviews the relevant literature, discusses how program reading can be taught, and illustrates teaching suggestions using a substantial Ada program. It contains an extensive, annotated bibliography.) The workshop introduced a simple model of program comprehension, discussed comprehension goals for reading, and gave participants both general and specific strategies for understanding programs. Instead of using Deimel's and Naveda's case study, actual artifacts from Widget were used to illustrate comprehension issues, concerns, and principles.

An example of the material in the comprehension unit is a brief discussion of how we come to understand what we read. We assume there exists an independent reality, the *real world*. We are interested in a small portion of that reality that is our particular application area. We think of the *application* as an abstraction of the real world. Our job as systems developers is to translate the features of that abstraction into the *computer* domain. There are thus two translations to be dealt with, the first from the real world into application terms, and the second from the application domain into computer terms. We come to understand these different domains (real world, application, and computer) by constructing models of them, and then we test those models by having a dialogue [Schön] with them in light of what we seek to accomplish (that is, compute). Reading and understanding a program is a complex process of translating, interpreting, and hypothesis testing among these (and possibly intermediate) domains.

In addition to the introduction of program comprehension material, there are three aspects of our form of inspection instruction that are distinctive that differ from "traditional" instruction, and may therefore have had some influence on the effectiveness of instruction and the conduct of inspections. First, we develop the process of inspection during the course, from the requirements and design elicited there. We do not arrive with a prepared process.

Second, the participants develop their own checklists based on ones available in the public domain that we supply. The participants usually develop two sets of checklists, one for each type of artifact they decide is most important for them to inspect. Code and requirements are the typical choices. Again, we do not arrive with the final, "best" checklists.

Third, the workshop participants select the artifacts to be inspected, one artifact of each type. Our advice is to select the oldest, most reliable artifacts that can be found. That way, finding defects using the new inspections process impresses even the most skeptical participants.

Results

Because the training of Group 1 grew, in part, out of dissatisfaction with the number of defects still found by customers, it was natural to examine customer defect reports for evidence of improvement. This was easily done, as written defect reports were received daily and were handled in the same, standard manner for all three groups. Reported defects were classified as "critical," "serious," or "other." Critical defects were those that either crashed the system or prevented the application from proceeding. Serious defects resulted in the production of wrong answers. All less severe defects were classified as "other."[2]

Of course, the software engineers trained in our two workshops took some time to begin applying the material presented. Moreover, only after inspected materials were released and in the field for a time did they begin to generate customer defect reports. From a detailed analysis of defect reports, it was determined that reports applying to software released by Group 1 made the transition from being predominately about pre-workshop modules to referring to post-workshop-inspected modules approximately eight weeks after the training was completed. After this time, post-workshop-inspected modules continued to predominate in the defect report stream for Group 1. About 40 days after this time, defect reports were nearly exclusively about software inspected after the training.

The transition between defect reports of pre- and post-workshop work products was short because most customer-discovered defects relate to fixes or enhancements requested by the customers themselves. Newly delivered code is checked immediately upon delivery by the customers or their representatives, who want to make sure it works correctly.

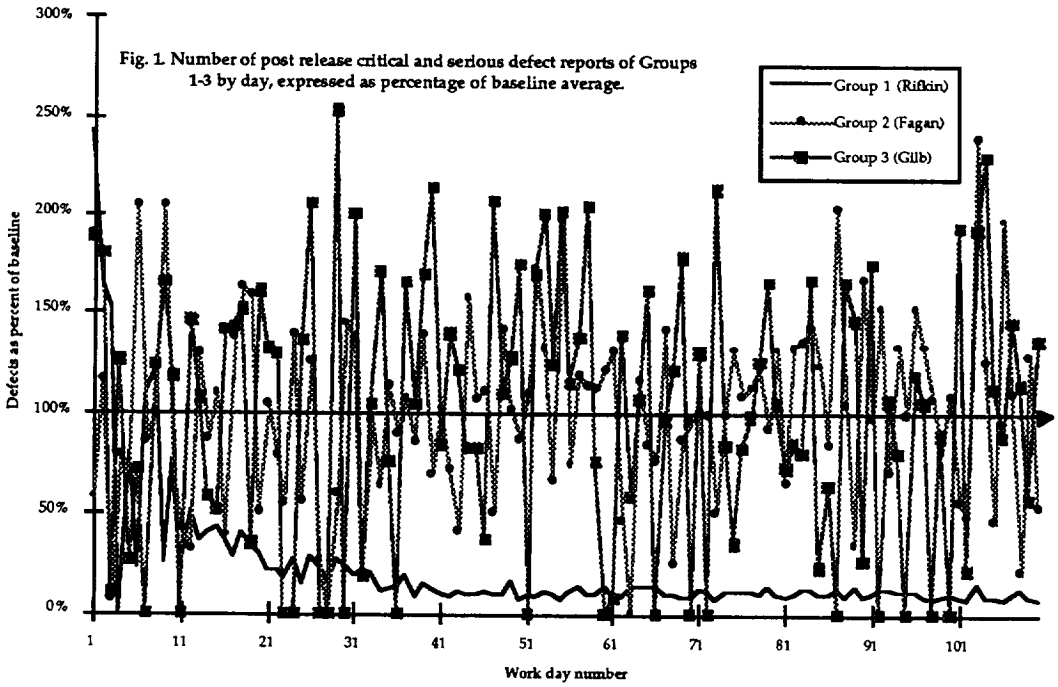
In order to establish a baseline to characterize error reports before our training workshops could exert any influence on behavior, we examined defect reports before and after the last workshop, counting critical and serious defects only. According to our analysis, there was no change in the pattern of Group 1 defects until about 10 working days[3] after the perceived inspection process

---

[2] Each of the groups also classified the type of error, though each used a different scheme. Groups 2 and 3 created their own, different defect categories, and Group 1 was trained in orthogonal defect classification [Chillarege]. The incompatibility of these defect taxonomies precluded drawing meaningful inferences about the differences in the types of defects detected.

[3] The data presented cover regular work days and exclude weekends and holidays, on which customer representatives do not normally work. Note that the modules most heavily used at any given time depend on the point in the product-development life cycle at which customers are working. We did not try to account for effects that might have been attributable to changing usage patterns, in part because, across the

changeover point referred to above. Groups 2 and 3 showed essentially steady-state behavior during this entire period, as one would expect. We therefore used the 10 days before the pattern of reported Group 1 defects began to change as our baseline period. Reports of critical and serious defects for which each of the three groups was responsible were counted during this period, and the average number of defects per day for each group was computed. Rather than presenting numbers of defects, we have expressed the data values as a percentage of the baseline average for each group. This seemed a fair way to measure pre-workshop (baseline) performance because (1) the groups were performing comparable tasks, (2) the groups had similar customer-identified defect rates, and (3) all groups inspected some of their work products, but not all.



The actual number of critical and serious defect reports received daily for each of the three groups was plotted for 110 days, beginning on the first day of the 10-day baseline period. These data are shown in Figure 1. We could have gone back much further than 10 days, but there would have been no change in the patterns seen. Plots by defect type (critical, serious, other) reveal the same pattern as the plots shown.

As might be expected, the data for Groups 2 and 3 vary around 100%, roughly between 0 and 2.5 times the average number of reports in the baseline period per day. The Group 1 data, on the other hand, are distinctive, after the first 10 days.

The customer-reported defects come directly from reports submitted by customers. Figure 1 shows the (normalized to 100%) number of defects recorded on such reports each day. Although the data do include multiple reports of the same defects, there are, in fact, few such duplications. The users are closely-knit and generally decide together to submit defect reports. Group 3 disputed the validity of several reports (that is, its members believed that no defect was indicated), and these are not represented; on days on which all of the Group 3 defects were disputed there is

---

three groups, there is considerable parallelism among the dozen or so products undergoing user development.

a zero count.[4] Group 1, on the other hand, decided, as a matter of policy, that any customer-reported defect *is* a defect, *ipso facto*.

It would have been useful to have been able to collect and compare defect densities, error injection rates, productivity, and other statistical measures of cross-group differences and similarities. No such measures were available, at least in part because none of the groups use an automated configuration management system, which could track easily the actual changes in code. Also, the lack of software configuration management made it impractical for us to ascertain the rate of errors introduced while trying to fix bugs, which can be quite large. We observe, though, that Groups 2 and 3 have been in existence longer than Group 1 and therefore may be more "mature" in some sense.

Analysis

Figure 1 suggests dramatic improvement in the post-workshop performance of Group 1. During the first 10 days, all three groups display the same up-and-down behavior of the number of defects attributable to their work. (There is no reason to expect that the number of reports should be constant from day-to-day.) In terms of absolute numbers, Group 1 was in the middle of the pack, as it had been for the previous 18 months. Then, after the products that Group 1 produced and inspected using the workshop methods begin to be released, there is a clear decrease in the number of post-release defects, those discovered by users. As can be seen from the scale of Figure 1, the rate drops to about 10% of the baseline average. In other words, there was a 90% reduction in the number of post-release defects per day discovered by users.

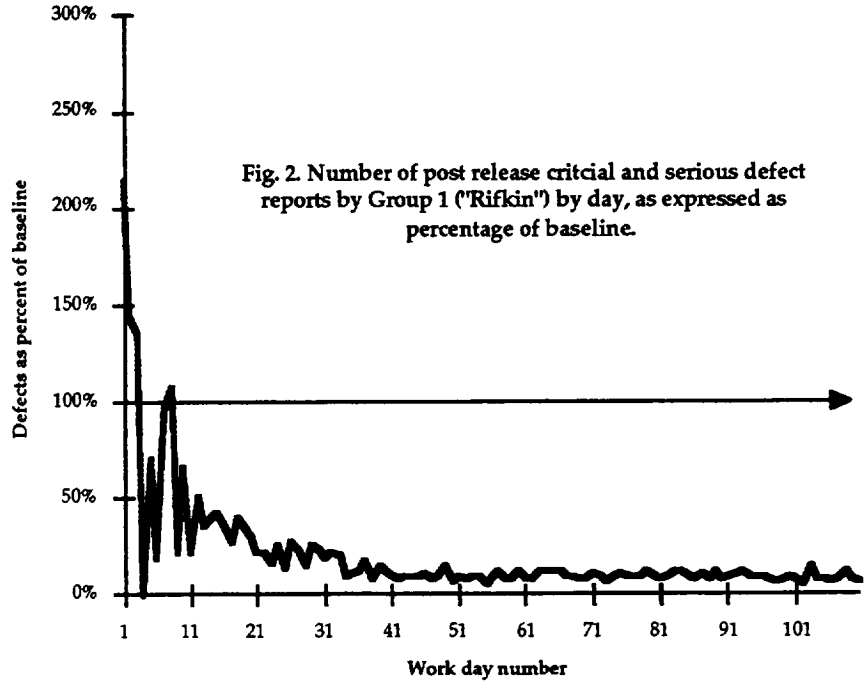


Fig. 2. Number of post release critcial and serious defect reports by Group 1 ("Rifkin") by day, as expressed as percentage of baseline.

[4] A zero count occurs when the development group does not agree that the user has found an error. In other words, there were no errors found for that day, even though some may have been reported.

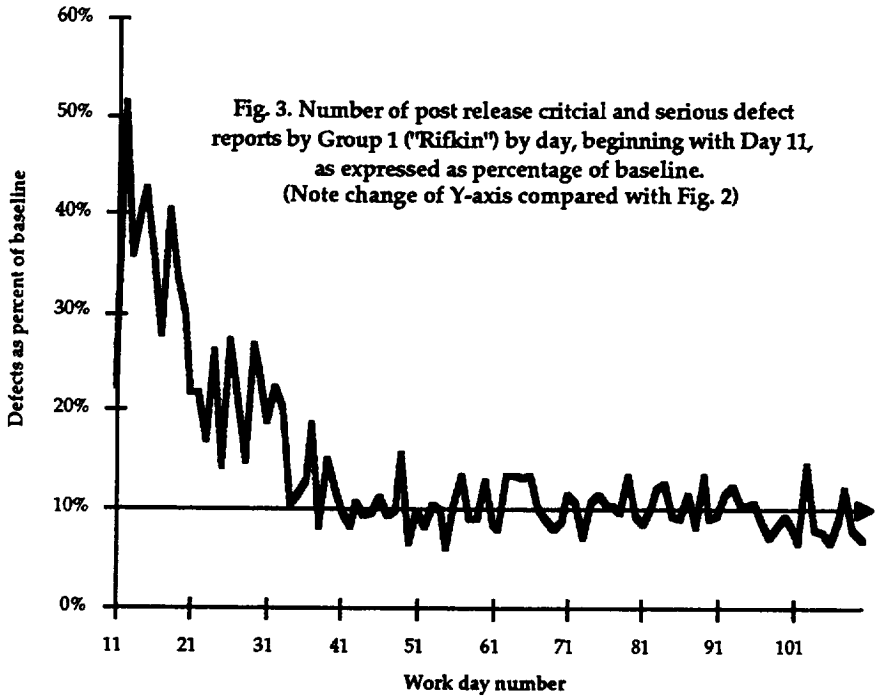


Fig. 3. Number of post release critcial and serious defect reports by Group 1 ("Rifkin") by day, beginning with Day 11, as expressed as percentage of baseline. (Note change of Y-axis compared with Fig. 2)

Figures 2-5 show individual curves for the three Groups. Figure 2 shows Group 1's up-and-down behavior during the first 10 days of this study, more characteristic of Groups 2 and 3. Then there is a steady drop in the number of defects reported by users. Figure 3 illustrates this decrease more clearly because of a vertical scale change resulting from showing only the data from the eleventh day onward. Figure 4 shows Group 2's post-release defect discovery history, and Figure 5, Group 3's. Groups 2 and 3 serve as control groups here—they were doing nothing differently—so there is no reason to expect their defect rates to show changes. Group 3 has a larger variance than Group 2, and also has many more zero counts.

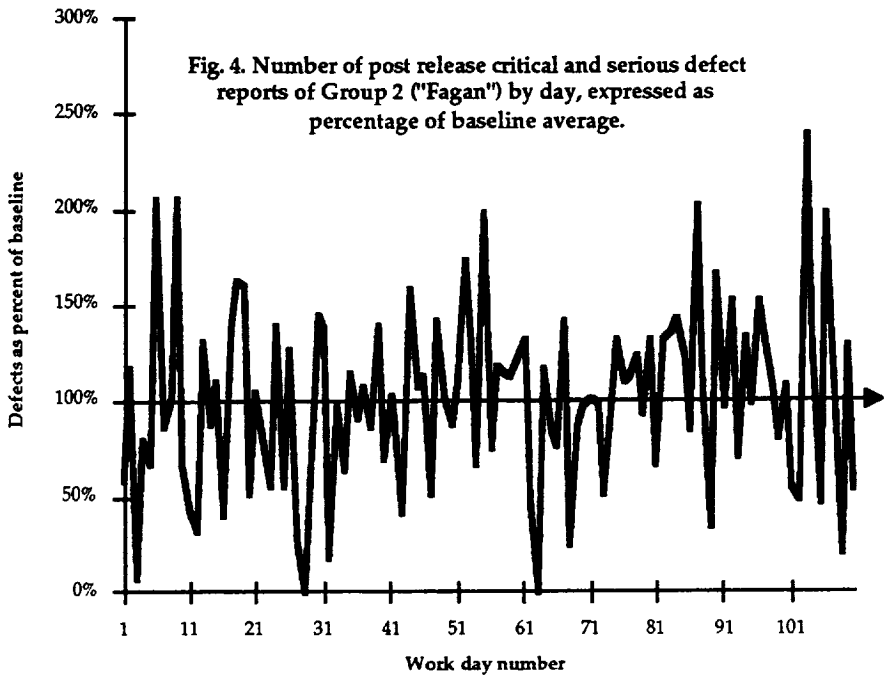


Fig. 4. Number of post release critical and serious defect reports of Group 2 ("Fagan") by day, expressed as percentage of baseline average.

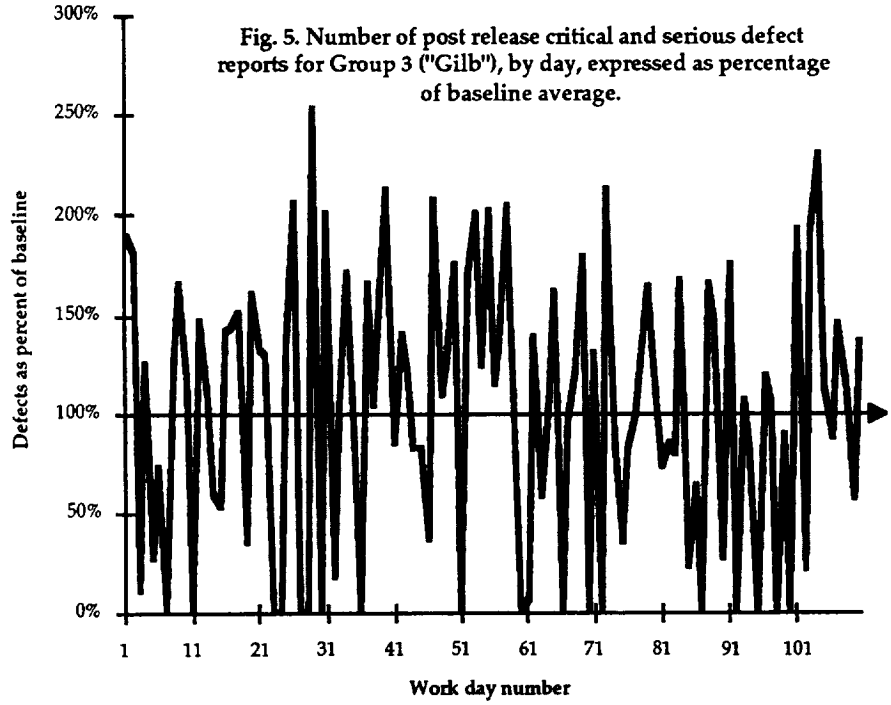
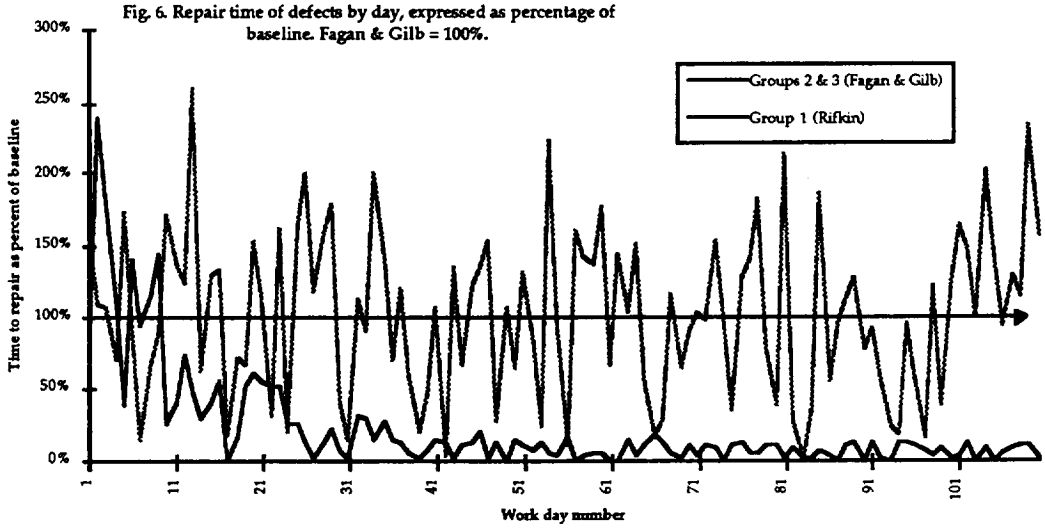


Fig. 5. Number of post release critical and serious defect reports for Group 3 ("Gilb"), by day, expressed as percentage of baseline average.

Using the data available, we investigated two questions:

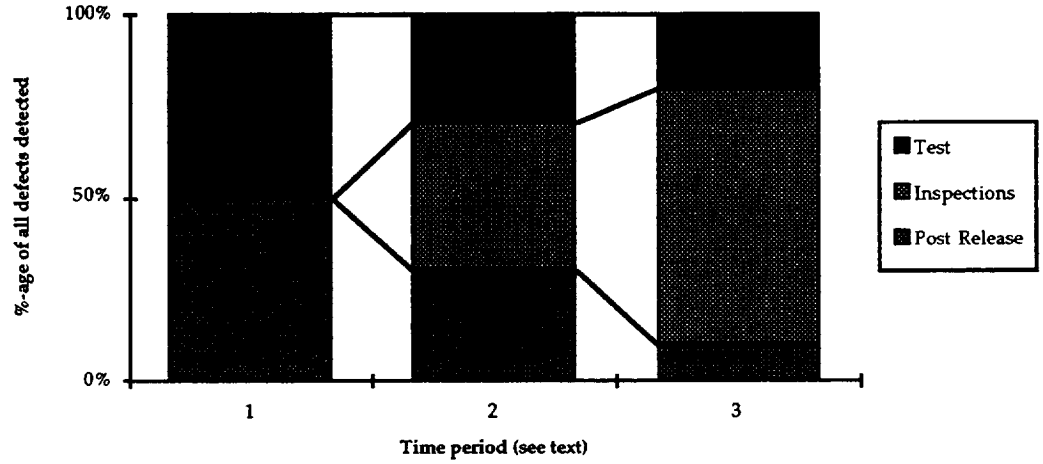
1. How does the decrease in the number of defects discovered post-release by users relate to the cost to repair those defects? In other words, do users discover the really difficult and expensive-to-fix defects, or do inspections catch them? We used effort, that is, time, to indicate cost. Repair data came directly from the defect reports. All groups report the time they spend repairing each defect. Figure 6 shows our findings: there is a significant reduction in the per-defect cost to repair user-discovered, post-release defects from Group 1, but not from Groups 2 and 3. We infer from this that Group 1 is either identifying expensive-to-repair defects before release or learning to program better in the first place. No special pattern is apparent in the data for Groups 2 and 3.

Fig. 6. Repair time of defects by day, expressed as percentage of baseline. Fagan & Gilb = 100%.



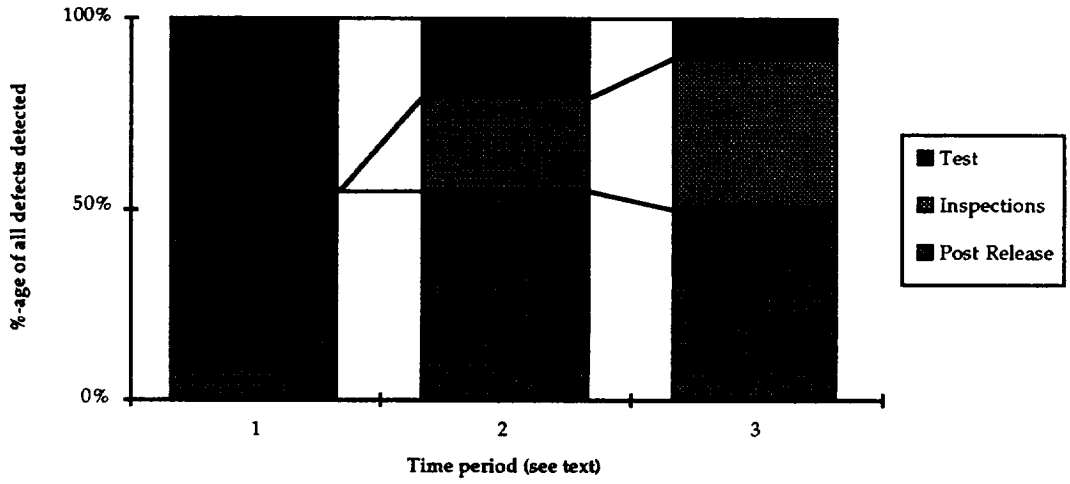
2. Does some other activity account for the difference in post-release defect discovery? We compared over time the relative effectiveness of testing, inspections, and post-release discovery in Figures 7-9. Times 1, 2, and 3 in these figures represent times just before inspection training, a few months after training, and a year or two after training, respectively.

Fig. 7. Percentage of Group 1 ("Rifkin") defects detected by mechanism over time.



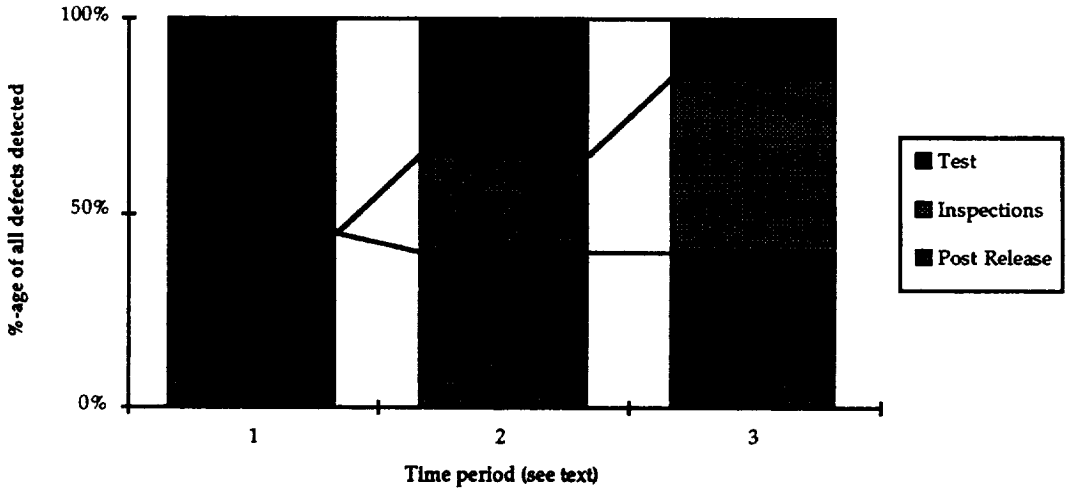
It is generally agreed that there are two ways to identify defects pre-release: reviews and testing. As noted at the beginning of this paper, inspection is a form of software review. The literature on the benefits of inspections commonly notes that the percentage of pre-release defects caught by inspections (and without testing) evolves from 0% before implementing inspections, to 70-80% after inspections are fully implemented; the remainder of pre-release defects being identified through testing [Gilb]. That was also Widget's experience, as seen from the figures. The authors are unaware of any literature about the impact of inspections specifically on post-release defects.

Fig. 8. Percentage of Group 2 ("Fagan") defects detected by mechanism over time.



Figures 8 and 9 indicate that Groups 2 and 3 did not experience a decrease in the percentage of defects discovered post-release by users, but, according to Figure 7, Group 1 did. In fact, according to the figures, the decrease in Group 1's post-release defect discovery was due, in large part, to inspections.

Fig 9. Percentage of Group 3 ("Gilb") defects detected by mechanism over time.



Implications

Did Group 1 improve simply because we paid attention to it—the so-called Hawthorne effect? We cannot say, but we have reason to doubt it. Like Groups 2 and 3, Group 1 knew it was being trained. It did not know it was being studied, however, as all the data collection and analysis were done after the fact from routine paperwork. Moreover, the Hawthorne effect presumably wears off after a time, and we saw no such effect. Some authors even argue that there never was a Hawthorne effect, that it was an artifact of the underlying Hawthorne site experiment and analysis [Jones].

The Widget experience suggests a number of inspections-related lessons or, at the very least, some ideas to be further explored. To begin with, it suggests that we should not be complacent about having discovered the ultimate form of group software review. Some writings, on inspections particularly, suggest fixed necessary and sufficient conditions required for effective reviews [Fagan]. Yet the nature of the defect classification used and the degree to which reviewers "own" their own process—other distinctive features of the training given members of Group 1—may play a significant rôle in making reviews useful. The primary lesson to be learned about inspections, however, is that, in the past, we may have paid too much attention to the global software review process and too little attention to the conduct of an individual and perhaps weighty process, namely the actual review of the software product.

What became obvious from the Widget experience was that individual software professionals have widely differing, sometimes poorly conceived, comprehension strategies . We often heard from workshop participants that, for the first time ever, they were able to say with some certainty that they did or did not understand what they were reviewing.

Comprehension skills *can* be improved with training. (Ideally, comprehension skills should be taught much earlier in their careers of software professionals [Deimel85].) Better comprehension skills among reviewers will likely facilitate development of a shared vision of what software products should look like in order to be understood, a vision that should feed back into the software process planning in a more effective way than merely following checklists. In fact, one author (Rifkin) uses this realization by clients as a milestone to assure that they understand the critical importance of comprehension: you cannot inspect what you cannot understand. Thus arises a new entry criterion for inspections: inspectability—can I comprehend what you have given me to review?

The apparent effectiveness of the inspection workshop is remarkable in light of the relatively superficial treatment given to program comprehension ideas. We theorize, however, that the material presented gave attendees a new way to think about programs and about what it means to examine them. This re-orientation may have been sufficiently powerful in its own right that the lack of supporting details was not a serious impediment to the development of improved program comprehension skills. Along with the introduction to program comprehension, we make the point repeatedly during training that this is just the beginning of a lifelong process of learning of how to understand what you read. The extensive bibliography of Deimel and Naveda suggests as much.

This study points to the importance of comprehension research in stark financial terms, as the comprehension training seems to have led to the identification of significant software defects not caught using a more simple-minded approach to software inspection. This research should continue, and the effect of program comprehension training on the identification of software defects should be examined in greater detail. It would be interesting, for example, to see the effect of providing *only* comprehension training to a group already performing inspections. (What would happen if Group 2 or 3 were given a 2-3 hour comprehension workshop?)

If indeed comprehension training improves performance during inspections, another interesting question is what material is most effective to present and what material can be used later to insure continuously improving inspection results.

Acknowledgments

We are indebted to our colleagues for their comments and feedback: Bill Brykczynski, Marilyn Bush, Bob Grady, Frank McGarry, K. David Neal, Ron Radice, and Ed Weller.

References

[Chillarege] R. Chillarege, R., *et al.*, "Orthogonal Defect Classification-A Concept for In-process Measurements," *IEEE Trans. Softw. Eng. 18*, 11, (November 1992) 943-956.

[Deimel85] Deimel, L. E. "The Uses of Program Reading," *ACM SIGCSE Bulletin 17*, 2 (June 1985) 5-14.

[Deimel90] Deimel, L. E., and J. F. Naveda. *Reading Computer Programs: Instructor's Guide and Exercises.* Educational Materials CMU/SEI-90-EM-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1990. Available electronically from the SEI via anonymous ftp from ftp.sei.cmu.edu as files em-3.ps and em-3code.txt in /pub/education.

[Fagan] Fagan, M. E. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems J. 15*, 3 (1976), 182-211;. Also Fagan, M. E,. "Advances in Software Inspections." *IEEE Software SE-12*, 7 (July 1986) 744-751;. Also Strauss, S., and R. Ebenau. *Software Inspection Process*,. New York: McGraw-Hill, 1994.

[Freedman] Freedman, D. P., and G. M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews*, 3rd Ed. New York: Little, Brown, 1982.

[Gilb] Gilb, T. *Principles of Software Engineering Management.* Wokingham, England: Addison-Wesley, 1988, Chapter 12;. Also Gilb, T., and Graham, D. *Software Inspection*. Reading, Mass.: Addison-Wesley, 1993.

[Humphrey] Humphrey, W. S. *Managing the Software Process.* Reading, Mass: Addison-Wesley, 1989, Section 15.4.3*ff.*

[Jones] Jones, S. R. G., "Was There a Hawthorne Effect?" *American J. Sociology 98*, 3 (November 1992) 451-468.

[Schön] Schön, D. A., *The Reflective Practitioner: How Professionals Think in Action*. New York: Basic Books, 1983.

An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development:

A Preliminary Report

A. Porter*
H. Siy*
Computer Science Department
University of Maryland
College Park, Maryland 20742
aporter@cs.umd.edu
harvey@cs.umd.edu

C. A. Toman
L. G. Votta
Software Production Research Department
AT&T Bell Laboratories
Naperville, Illinois 60566
cat@intgp1.att.com
votta@research.att.com

September 6, 1994

Abstract

This experiment (currently in progress) is designed to measure costs and benefits of different code inspection methods. It is being performed with a real development team writing software for a commercial product. The dependent variables for each code unit's inspection are the elapsed time and the number of defects detected. We manipulate the method of inspection by randomly assigning reviewers, varying the number of reviewers and the number of teams, and when using more than one team, randomly assigning author repair and non-repair of detected defects between code inspections.

After collecting and analyzing the first 17% of the data, we have discovered several interesting facts about reviewers, about the defects recorded during reviewer preparation and during the inspection collection meeting, and about the repairs that are eventually made. (1) Only 17% of the defects that reviewers record in their preparations are true defects that are later repaired. (2) Defects recorded at the inspection meetings fall into three categories: 18% false positives requiring no author repair, 57% soft maintenance where the author makes changes only for readability or code standard enforcement, and 25% true defects requiring repair. (3) The median elapsed calendar time for code inspections is 10 working days – 8 working days before the collection meeting and 2 after. (4) In the collection meetings, 31% of the defects discovered by reviewers during preparation are suppressed. (5) Finally, 33% of the true defects recorded are discovered at the collection meetings and not during any reviewer's preparation.

The results to date suggest that inspections with two sessions (two different teams) of two reviewers per session (2sX2p) are the most effective. These two-session inspections may be performed with author repair or with no author repair between the two sessions. We are finding that the two-session-two-person-with-repair (2sX2pR) inspections are the most expensive, taking 15 working days of calendar time from the time the code is ready for review until author repair is complete, whereas two-session-two-person-with-no-repair (2sX2pN) inspections take only 10 working days, but find about 10% fewer defects.

** This work is supported in part by the National Aeronautics and Space Administration under grant NSG-5123. Mr. Siy was also partly supported by AT&T's Summer Employment Program.*

1 Introduction

For almost twenty years, software inspections have been promoted as a cost-effective way to improve software quality. Their expense is often justified by observing that the longer a defect remains in a system, the more expensive it is to repair, and therefore the future cost of fixing defects is greater than the present cost of finding them.

However, this reasoning is naive because inspection costs are significantly higher than many people realize. In practice, large projects perform hundreds of inspections, each requiring five or more participants. Holding such a large number of meetings can cause delays which may significantly lengthen the development interval (calendar time to completion).[1] Since long development intervals risk substantial economic penalties, the hidden cost of the current inspection process must be considered.

We hypothesize that different inspection approaches involve different tradeoffs between minimum interval and maximum effectiveness. But until now there have been no empirical studies to evaluate these tradeoffs. We have conducted such a study, and our results indicate that the choice of approach significantly affects the cost-effectiveness of the inspection.

Below, we review the relevant research literature, describe the various inspection approaches we examined, and present our experimental design, analysis, and conclusions.

1.1 Literature Review

To eliminate defects, many organizations use an iterative, three-step inspection procedure: Preparation, Collection, Repair[11] . First, a team of reviewers reads the artifact, detecting as many defects as possible. Next, these newly discovered defects are collected, usually at a team meeting. They are then sent to the artifact's author for repair. Under some conditions the entire process may be repeated one or more times.

Many articles have been written about inspections. Most, however, are case studies describing their successful use [8, 9, 20, 17, 24, 12, 1] . Few, critically analyze inspections or rigorously evaluate alternative approaches. We believe that additional critical studies are necessary because the cost-effectiveness of inspections may well depend on such variables as team size, number of inspection sessions, and the ratio of individual contributions versus group efforts.

Team Size: Inspections are usually carried out by a team of four to six reviewers. Buck[2] provides data (from an uncontrolled experiment) that showed no difference in the effectiveness of three, four, and five-person teams. However, no studies have measured the effect of team size on inspection interval.

Single-Session vs. Multiple-Session Inspections: Traditionally, inspections are carried out in a single session. Additional sessions occur only if the original artifact or the inspection itself is believed to be seriously flawed. But some authors have argued that multiple session inspections might be more effective.

Tsai et al.[18] developed the N-fold inspection process, in which N teams each carry out independent inspections of the entire artifact. The results of each inspection are collated by a single moderator, who removes duplicate defect reports. N-fold inspections will find more defects than regular inspections as long as the teams don't completely duplicate each other's work. However, they are far more expensive than a single team inspection.

Parnas and Weiss' active design reviews (ADR)[14] and Knight and Myers' phased inspections (PI)[13] are also multiple-session inspection procedures. Each inspection is divided into several mini-inspections or "phases". ADR phases are independent, while PI phases are executed sequentially and all known defects are repaired after each phase. Usually each phase is carried out by one or more reviewers concentrating on a single type of defect.

The authors believe that multiple-session inspections will be much more effective than single-session inspections, but they do not show this empirically, nor do they consider any effects on inspection interval.

Group-centered vs. Individual-centered Inspections: It is widely believed that most defects are first identified during the collection meeting as a result of group interaction[7] . Consequently, most research has focused on streamlining the collection meeting by determining who should attend, what roles they should play, how long the meeting should last, etc.

[1] As developer's calendars fill up, it becomes increasingly difficult to schedule meetings. This pushes meeting dates farther and farther into the future, increasing the development interval.

On the other hand, several recent studies have concluded that most defects are actually found by individuals prior to the collection meeting. Humphrey [10] claims that the percentage of defects first discovered at the collection meeting ("meeting gain rate") averages about 25%. In an industrial case study of 50 design inspections, Votta [22] found far lower meeting gain rates (about 5%). Porter et. al[16] conducted a controlled experiment in which graduate students in computer science inspected several requirements specifications. Their results show meeting gain rates consistent with Votta's. They also show that these gains are offset by "meeting losses" (defects first discovered during preparation but never reported at the collection meeting). Again, since this issue clearly affects both the research and practice of inspections, concrete studies are needed.

1.2 Hypotheses

Inspection approaches are usually evaluated according to the number of defects they find. As a result, some information is available about the effectiveness of different approaches, but very little about their costs. We believe that cost is as important as effectiveness, and we hypothesize that different approaches have significantly different tradeoffs between development interval and detection effectiveness. Specifically, we hypothesize that

- inspections with large teams have longer inspection intervals, but find no more defects than smaller teams;
- collection meetings do not significantly increase detection effectiveness;
- multiple-session inspections are more effective than single-session inspections, but significantly increase inspection interval.

2 The Experiment

To evaluate these hypotheses we designed and are conducting a controlled experiment. Our purpose is to compare the tradeoffs between minimum interval and maximum effectiveness of several inspection approaches.

2.1 Experimental Setting

We are currently running this experiment at AT&T on a project that is developing a compiler and environment to support developers of the AT&T's 5ESS® telephone switching system. The finished system is expected to contain 30K lines of C++ code, of which about 6K is reused.

All of the team's six members are experienced developers, and all have received training on inspections. The project began coding during June, 1994, and will perform about 100 code inspections by the end of the year.

2.2 Operational Model

To test our hypotheses we must measure both the interval and the effectiveness of every inspection. We began by constructing models for calculating inspection interval and estimating the number of defects in a code unit. These models are depicted in Figure 1.

2.2.1 Modeling the Inspection Interval

The inspection process begins when a code unit is ready for inspection and ends when the author finishes repairing the defects found in the code. The elapsed time between these events is called the inspection interval.

The length of this interval depends on the time spent working (preparing, attending collection meetings, and repairing defects) and the time spent waiting (time during which the inspection does not progress due to process dependencies, higher priority work, scheduling conflicts, etc).

In order to measure inspection interval and its various subintervals, we devised an inspection time model based on visible inspection events [23] . Whenever one of these events occurs it is timestamped and the event's participants are recorded. (In most cases this information is manually recorded on the forms described in Section 2.4.1.) These events occur, for example, when code is ready for inspection, or when a reviewer is finished with his or her preparation. This information is entered into a database, and inspection intervals are reconstructed by performing queries against the database.

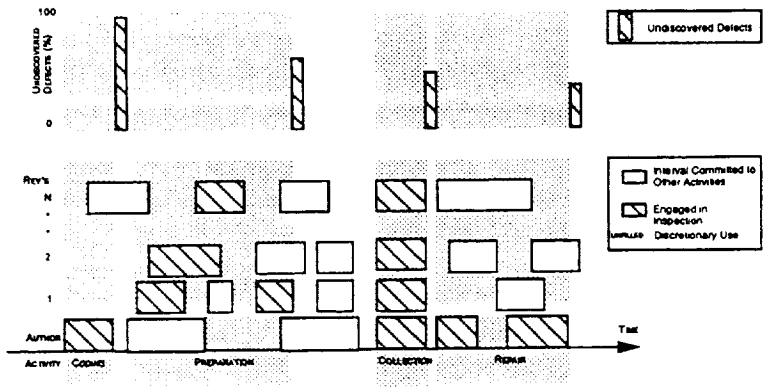


Figure 1: This figure depicts a simple model of the inspection process. The figure's lower panel summarizes the inspection's time usage. Specifically, it shows the inspection's participants (an author and several reviewers), the activities they perform (coding, preparation, collection, repair, and other), the interval devoted to each activity (denoted by the shaded areas), and the total inspection interval (end of coding to completion of repair). It also shows that in a large organization, inspections must compete with other processes for limited time and resources. The upper portion of the figure shows when and to what extent inspections remove defects from the code.

2.2.2 Modeling the Defect Detection Ratio

One important measure of an inspection's effectiveness is its defect detection ratio – the number of defects found during the inspection divided by the total number of defects in the code. Because we never know exactly how many defects an artifact contains, it is impossible to make this measurement directly, and therefore we are forced to approximate it.

We will use the following approaches to approximate the defect detection ratio.

- **Observed detection ratio:** We assume that total defect density is constant for all code units and that we can compare the number of defects found per KNCSL. This is always available, but very imprecise.
- **Complete estimation of detection ratio:** We track the code through testing and field deployment, recording new defects as they are found. This is more precise, but is not available until well after the project is completed.

2.3 Experimental Design

2.3.1 Variables

The experiment manipulates three independent variables:

1. the team size (one, two, or four members, in addition to the author),
2. the number of inspection sessions (one session or two sessions),
3. the coordination between passes (in two-session inspections the author may or may not repair known defects between sessions).

The treatment distributions are shown in Table 1.

For each inspection we measured four dependent variables:

1. inspection intervals,
2. estimated defect detection ratio,
3. the percentage of defects first identified at the collection meeting (meeting gain rate),
4. the percentage of potential defects reported by an individual, but not recorded at the collection meeting (meeting suppression rate).

We also capture repair statistics for every defect.

	Number of Sessions			Totals
	1	2		
Team Size		With Repair	Without Repair	
1	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{3}$
2	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{3}$
4	$\frac{1}{3}$	0	0	$\frac{1}{3}$
Totals	$\frac{5}{9}$	$\frac{2}{9}$	$\frac{2}{9}$	1

Table 1: This table gives the percentage of inspections allocated to each setting of the independent variables. Note: Since we cannot apply capture-recapture estimates to the data from the one-session-one-person or two-session-one-person-with-repair inspections, this data will be held out of the capture-recapture analysis.

2.3.2 Design

This experiment uses a $2^2 \times 3$ partial factorial design to compare the interval and effectiveness of inspections with different team sizes, number of inspection sessions, and coordination strategies. We chose a partial factorial design because some treatment combinations were considered too expensive (e.g., two-session-four-person inspections with and with no repair).

2.3.3 Threats to Internal Validity

Threats to internal validity are influences that can affect the dependent variable without the researcher's knowledge. We considered three such influences: (1) selection effects, (2) maturation effects, and (3) instrumentation effects.

Selection effects are due to natural variation in human performance. For example, if one-person inspections are done only by highly experienced people, then their greater than average skill can be mistaken for a difference in the effectiveness of the treatments. We limited this effect by randomly assigning team members for each inspection. This way individual differences are spread across all treatments.

Maturation effects result from the participants' skills improving with experience. Again we randomly assigned the treatment for each inspection to spread any performance improvements across all treatments.

Instrumentation effects are caused by code to be inspected, by differences in the data collection forms, or by other experimental materials. In this study, one set of data collection forms was used for all treatments. Since we could not control code quality or code size, we randomly assigned the treatment for each inspection.

2.3.4 Threats to External Validity

Threats to external validity are conditions that limit our ability to generalize the results of our experiment to industrial practice. We considered three sources of such threats: (1) experimental scale, (2) subject generalizability, and (3) subject representativeness.

Experimental scale becomes a threat when the experimental setting or the materials are not representative of industrial practice. We avoided this threat by conducting the experiment on a live software project.

A threat to subject generalizability may exist when the subject population is not drawn from the industrial population. This is not a concern here because our subjects are software professionals.

Threats regarding subject representativeness arise when the subject population is not representative of the industrial population. This may endanger our study because our subjects are members of a development team and so are not a random sample of the entire development population.

2.3.5 Analysis Strategy

Once the data are collected we will analyze the combined effect of the independent variables on the dependent variables to evaluate our principal hypothesis. Once the significant explanatory variables are discovered and their magnitude estimated, we will examine subsets of the data to study our specific hypotheses.

2.4 Experimental Instrumentation

We designed several instruments for this experiment: preparation and meeting forms, author repair forms, and participant reference cards.

2.4.1 Data Collection Forms

We designed two data collection forms, one for preparation and another for the collection meeting.

The meeting form is filled in at the collection meeting. When completed, it gives the time during which the meeting was held, and a page number, a line number, and an ID for each defect.

The preparation form is filled in during both preparation and collection. During preparation, the reviewer records the times during which he or she reviewed, and the page and line number of each issue ("suspected" defect). During the collection meeting the team will decide which of the reviewer's issues are, in fact, real defects. At this time, real defects are recorded on the meeting form and given an ID. The reviewer then links this ID to his or her preparation form.

2.4.2 Author Repair Forms

The author repair form captures information about each defect identified during the inspection. This information includes Defect Disposition (no change required, repaired, deferred); Repair Effort ($\leq 1hr$, $\leq 4hr$, $\leq 8hr$, or $> 8hr$), Repair Locality (whether the repair was isolated to the inspected code unit), Repair Responsibility (whether the repair required other developers to change their code), Related Defect Flag (whether the repair triggered the detection of new defects), and Defect Characteristics (whether the defect required any change in the code, was changed to improve readability or to conform to coding standards, was changed to correct violations of requirements or design, or was changed to improve efficiency).

This information is used to discard certain defect reports from the analysis – i.e., those regarding defects that required no changes to fix them or concerned coding style rather than incorrect functionality.

2.4.3 Participant Reference Cards

Each participant received a set of reference cards containing a concise description of the experimental procedures and the responsibilities of the authors and reviewers.

2.5 Conducting the Experiment

To support the experiment, Mr. Harvey Siy, a doctoral student working with Dr. Porter at the University of Maryland, joined the development team in the role of inspection quality engineer (IQE). The IQE is responsible for tracking the experiment's progress, capturing and validating data, and observing all inspections. The IQE also attends the development team's meetings, but has no development responsibilities.

When a code unit is ready for inspection, its author sends an inspection request to the IQE. The IQE then randomly assigns a treatment (based on the treatment distributions given in Table 1) and randomly draws a review team from the reviewer pool.[2] These names are then given to the author, who schedules the collection meeting.

Once the meeting is scheduled, the IQE puts together the team's inspection packets.[3] The IQE attends the collection meeting to ensure that all the procedures have been correctly followed. After the collection meeting he gives the preparation forms to the author, who then repairs the defects, fills out the author repair form, and returns all forms to the IQE. After the forms are returned, the IQE interviews the author to validate the data.

3 Data and Analysis

Four sets of data are important for this study: the team defect summaries, the individual defect summaries, the interval summaries, and the author repair summaries. This information is captured on the preparation, meeting, and repair forms.

[2] We do not allow any single reviewer to be assigned to both teams in a two-session inspection.

[3] The inspection packet contains the code to be inspected, all required data collection forms and instructions, and a notice giving the time and location of the collection meeting.

The team defect summary forms show all the defects discovered by each team. This form is filled out by the author during the collection meeting and is used to assess the effectiveness of each treatment. It is also used to measure the added benefits of a second inspection session by comparing the meeting reports from both halves of two-session inspections with no repair.

The individual defect summary forms show whether or not a reviewer discovered a particular defect. This form is filled out during preparation to record all suspected defects. The data is gathered from the preparation form and is compiled during the collection meeting when reviewers cross-reference their suspected defects with those that are recorded on the meeting form. This information, together with the team summaries, is used to calculate the capture-recapture estimates and to measure the benefits of collection meetings.

The interval summaries describe the amount of calendar time that was needed to complete the inspection process. This information is used to compare the average inspection interval and the distribution of subintervals for each treatment.

The author repair summaries characterize all the defects and provide information about the effort required to repair them.

As of this writing, only 17% of the planned inspections have been completed. Consequently, we do not yet have enough data to definitively evaluate our hypotheses. However, we can look at the apparent trends in our preliminary data, explore the implications of this data for our hypotheses, and discuss how the resolution of these hypotheses at the completion of the experiment will help us answer several open research questions.

3.1 Data Reduction

Data reduction is the manipulation of data after its collection. We have reduced our data in order to (1) remove data that is not pertinent to our study, and to (2) adjust for systematic measurement errors.

3.1.1 Reducing the Defect Data

The preparation and meeting forms capture the set of issues that were raised during each inspection. In practice, many of these issues, even if they went unrepaired, would not lead to incorrect system behavior, and they are therefore of no interest to our analysis.

Based on information in the repair form and interviews with each author, we classified the issues into one of three categories:

- False Positives (issues for which no changes were made),
- Soft Maintenance (issues for which changes were made only to improve readability or enforce coding standards),
- True Defects (issues for which changes were made to fix requirements or design violations, or to improve system efficiency).

Although defect classifications are usually made during the collection meeting, we feel that authors understand the issues better after they have attempted to repair them, and are then better able to make more reliable classifications.

The distribution of defect classifications for each treatment appears in Figure 2. Across all inspections, 18% of the issues are False Positives, 57% involve Soft Maintenance, and 25% are True Defects.

We consider only True Defects in our analysis of estimated defect detection ratio (a dependent variable).[4]

3.1.2 Reducing the Interval Data

The preparation, meeting, and repair forms show the dates on which important inspection events occur. This data is used to construct the inspection intervals (usually considered to be the calendar period between the submission of an inspection request and the completion of all repairs).

We made two reductions to this data.

First, we observed that some authors did not repair defects immediately following the collection meeting. Instead, they preferred to concentrate on other development activities, and fix the defects later, during slow work

[4] We observed that most of the soft maintenance issues are caused by conflicts between the coding style or conventions used by different reviewers. In and of themselves, these are not true defects. We feel these issues might be more efficiently handled outside of the inspection process with automated tools or standards.

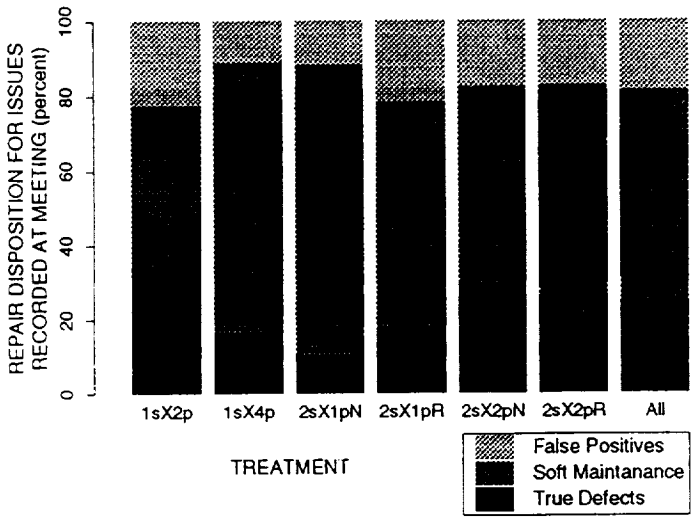


Figure 2: **Disposition of Issues Recorded at the Collection Meeting.** For each treatment, the stacked barchart shows the percentage of the issues recorded at collection meetings that turn out to be false positives, soft maintenance, or true defects. Across all treatments, only 25% of the issues are true defects.

	Number of Sessions			Totals
	1	2		
Team Size		With Repair	Without Repair	
1	0	3	2	5
2	5	2	4	11
4	1	0	0	1
Totals	6	5	6	17

Table 2: This table shows the number of observations we currently have for each treatment.

periods. To remove these cases from the analysis, we redefined the inspection interval to be the calendar period between the submission of an inspection request and the completion of the collection meeting.

When these reductions are made, two-session inspections have two inspection subintervals – one for each session. We equate the interval for such inspections with the longer of these two subintervals, since both of them begin at the same time.

Next, we removed all nonworking days from the interval. Nonworking days are defined as (1) weekend days during which no inspection activities occur, or (2) days during which the author is on vacation and no reviewer performs any inspection activities.

We use the length of these reduced intervals in our analysis of the inspection interval.

Figure 3 is a boxplot[5] showing the number of working days from the issuance of the inspection request to the collection meeting, from the collection meeting to the completion of repair, and the total. The total inspection interval has a median of 10 working days, 8 before and 2 after the collection meeting.

3.2 Overview of Data

Table 2 shows the number of observations to date for each treatment. Figure 4 is a contrast plot showing the interval and effectiveness of all inspections and for every setting of each independent variable. This information is

[5]In this paper we have made extensive use of boxplots to represent prominent features of a distribution. Each set of data is represented by a box, the height of which corresponds to the spread of the central 50% of the data, with the upper and lower ends of the box marking the upper and lower quartiles. The data median is denoted by a bold point within the box. The lengths of the vertical dashed lines relative to the box indicate how stretched the tails of the distribution are; they extend to the standard range of the data, defined as 1.5 times the inter-quartile range. The detached points are "outliers" lying beyond this range.[4]

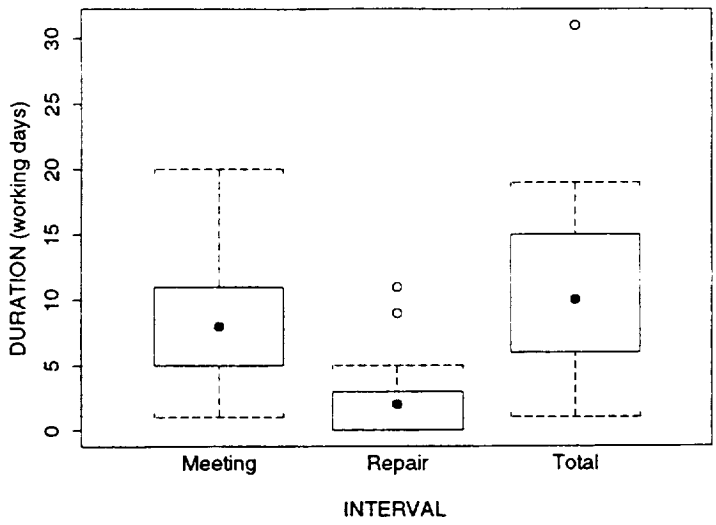


Figure 3: **Premeeting Inspection Interval.** These boxplots show all the interval data divided into two parts: time before the meeting and time after the meeting. The total inspection time has a median of 10 days, 80% of which is before the meeting.

used to determine the amount of the variation in the dependent variables that is explained by each independent variable. We also show another variable, total number of reviewers (the number of reviewers per session multiplied by the number of sessions). This variable provides information about the relative influence of team size vs. number of sessions.

3.3 Analysis of Interval Data

Inspection interval is an important measure of cost. Figure 5 shows the inspection interval (premeeting only) by treatment and for all treatments.

We draw several observations from this data. First, the interval of two-session-one-person inspections is no longer than the interval of one-session-two-person inspections. Second, 2sX2pN inspections also have no longer interval than 1sX2p inspections.

The cost of serializing two inspection sessions is suggested by comparing 2sX2pN inspections with 2sX2pR inspections. The 2sX2pR inspections have a 53% longer interval than the 2sX2pN inspections. This indicates that any multiple session inspections that require repair after each session will significantly increase the inspection interval.

The additional cost of multiple inspection sessions can be seen by comparing 1sX2p inspections with 2sX2pN inspections. The 2sX2pN interval is only slightly longer than the 1sX2p interval. However, since the author must be involved in each session, the interval is likely to grow as the number of sessions increases.

3.4 Analysis of Effectiveness Data

The benefit of inspections is that they find defects. This benefit will vary with the different inspection treatments. Figure 6 shows the observed defect density for all inspections and for each treatment separately.

Several interesting trends appear in the preliminary data. First, 1sX2p inspections are as effective as 1sX4p inspections. Second, 2sX2p inspections appear to be more effective than any one-session inspection, but 2sX1p inspections are *not* more effective than 1sX2p inspections. Finally, 2sX2pR inspections are more effective than 2sX2pN inspections.

The effectiveness of different team sizes is suggested by comparing 1sX2p, 1sX4p, and 2sx1pN inspections. The low effectiveness of 1sX4p inspections may indicate that current inspection teams are too large; however, with only a single 1sX4p inspection drawing any conclusions would be premature.

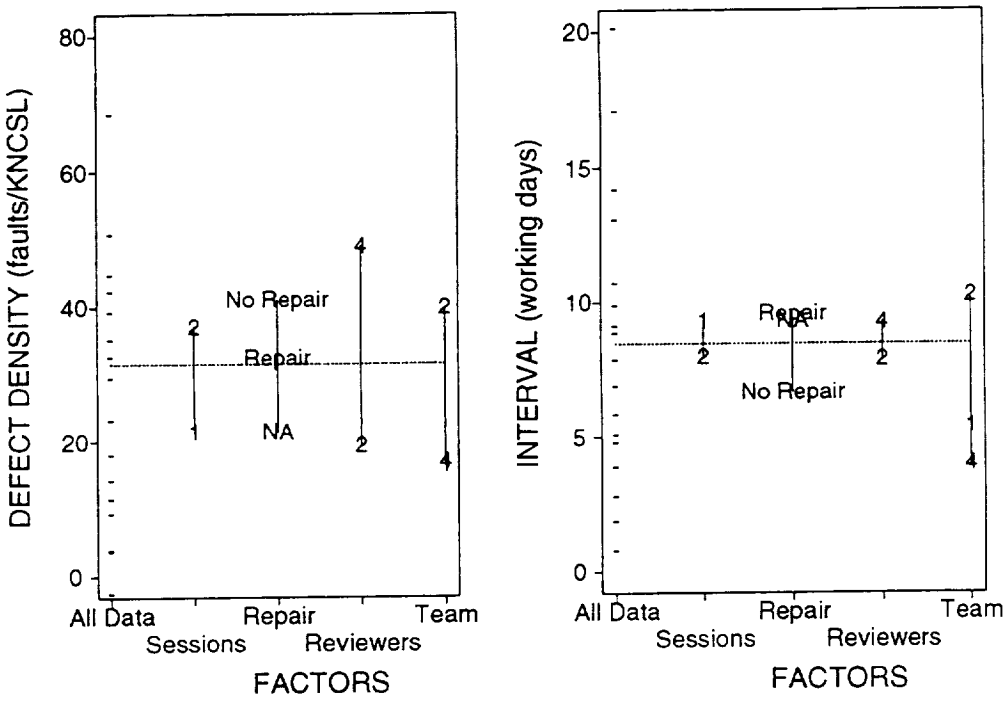


Figure 4: **Effectiveness and Interval by Independent Variables.** The dashes in the far left column of the first plot show the defect detection rates for all inspections. The dotted horizontal line marks the average defect detection rate. The other four columns indicate factors that may influence this dependent variable. The plot demonstrates the ability of each factor to explain variations in the dependent variable. For the Sessions factor, the vertical locations of the symbols "1" and "2" are determined by averaging the defect detection rates for all code inspection units having 1 or 2 sessions. The right plot shows similar information for inspection interval.

The additional effectiveness of multiple sessions is suggested by comparing 1sX4p and 2sX2p and 2sX1p inspections. This data indicates that it is more effective to use two teams of two persons each than to use a single team of four persons. However, it appears that two teams of one person are *not* more effective than a single team of two persons. Many multiple session methods rely on the assumption that several one person teams can be more effective than a single large team. However, our results suggest that the performance of individual reviewers must be increased if multiple session methods are to be effective.

The additional effectiveness due to serializing multiple sessions is suggested by comparing 2sX2pR against 2sX2pN inspections. While the data shows that 2sX2pR inspections are the more effective, the difference in effectiveness between 2sX2pR and 2sX2pN inspections is small, about 2 defects per 300 NCSL.

3.5 Meeting Effects

During prepreparation, reviewers analyze the code units to discover defects. After all reviewers are finished preparing, a collection meeting is held. These meetings are believed to serve at least two important functions: (1) suppressing unimportant or incorrect defect reports, and (2) finding new defects. These meetings have a significant effect on inspection performance.

Analysis of Preparation Reports. One input to the collection meeting is the list of defects found by each reviewer during his or her preparation. Figure 7 shows the percentage of defects reported by each reviewer that are eventually determined to be true defects. Across all inspections, only 25% of all reports turn out to true defects. This figure appears to be independent of inspection treatment.

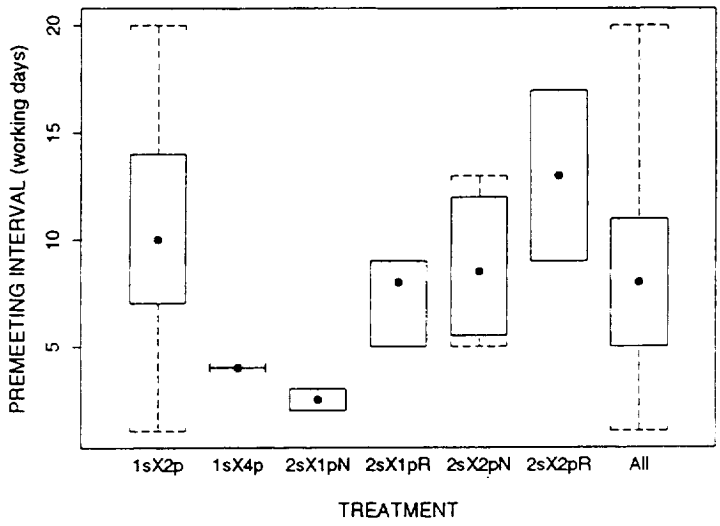


Figure 5: **Premeeting Interval by Treatment.** This plot shows the observed interval for each inspection treatment.

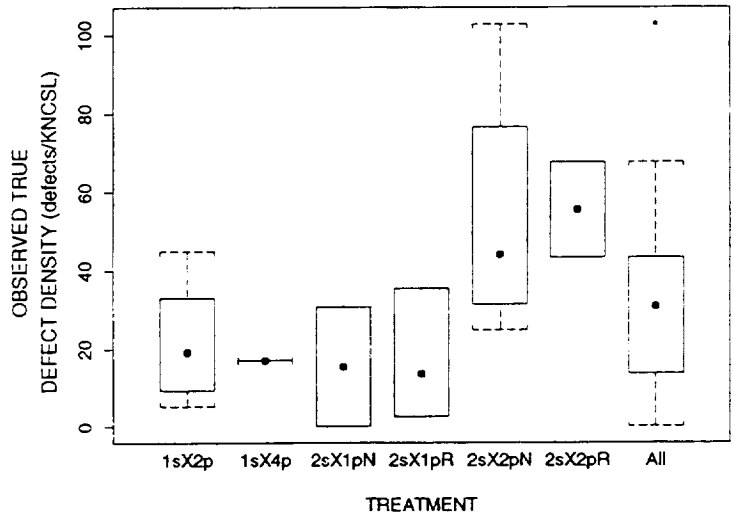


Figure 6: **Observed Defect Density by Treatment.** This plot shows the observed defect density for each inspection treatment. Across all inspections, 32 defects were found per KNCSL.

Analysis of Suppression. It is generally assumed that collection meetings suppress unimportant or incorrect defect reports, and that without these meetings, authors would have to process many spurious reports during repair.

Figure 8 shows the suppression rates for all inspections. One trend in the preliminary data is that four-person inspections consistently suppress more reports than any other treatment. (One-session-two-person inspections show considerable variability.)

Analysis of Meeting Gains Another function of the collection meeting is to find new defects in addition to those discovered by the individual reviewers. Defects that are are first discovered at the collection meeting are

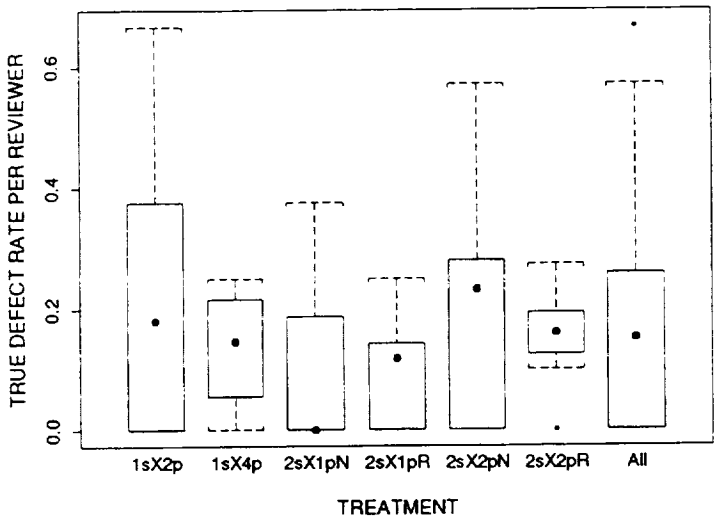


Figure 7: **True Defect Rate per Reviewer Prepartion Report by Treatment.** This boxplot shows the rate at which defects found during preparation are eventually considered to be true defects. Across all treatments, only 17% of the reports turn out to be true defects.

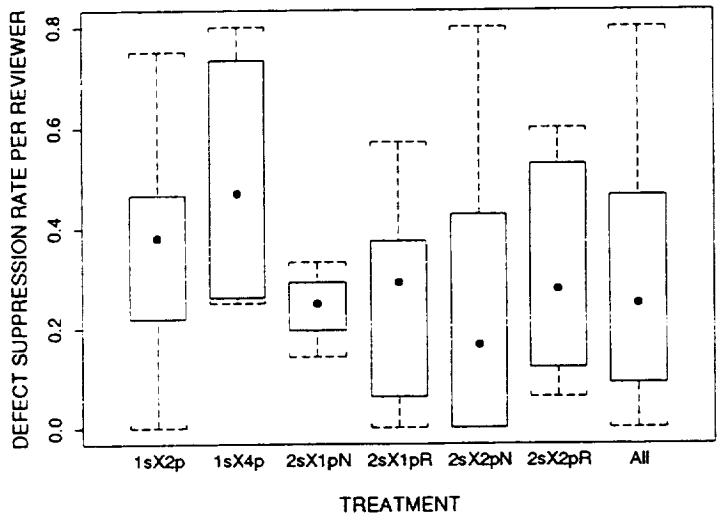


Figure 8: **Meeting Suppresion Rate by Treatment.** These boxplots show the suppression rate for each reviewer by treatment. The suppression rate for a reviewer is defined as the number of defects detected during preparation but not included in the collection meeting defect report, divided by the total number of defects recorded by the reviewer in his/her preparation. Across all inspections, 31% of the preparation reports are suppressed.

called meeting gains.

Figure 9 shows the meeting gain rates for all inspections. Across all inspections, 33% of all defects discovered are meeting gains. The data suggests that 1sX4p inspections have the lowest gain rates.

The effect of team size is suggested by comparing 1sX4p inspections to all others. Although most of the treatments show similar gain rates, one-session-four-person inspections appear to be the lowest. This may indicate

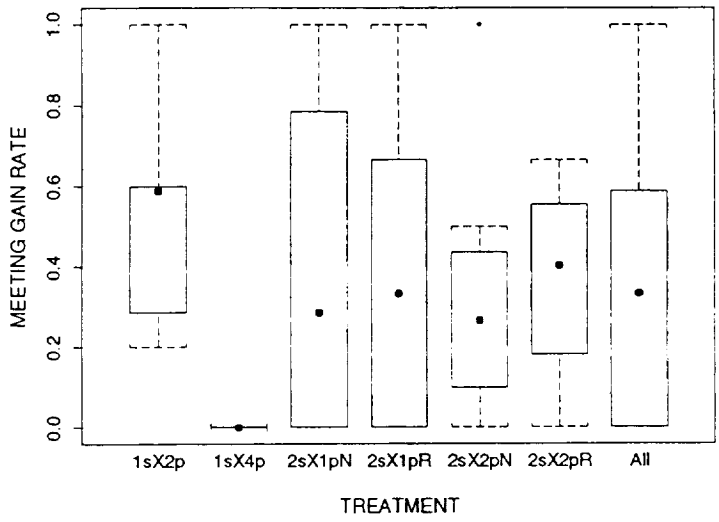


Figure 9: **Meeting Gain Rate by Treatment.** These boxplots shows the meeting gain rates for all inspections and for each treatment. The median rate was 33%.

that larger team size can be detrimental to effectiveness.

The data indicate that almost half of the defects reported during preparation turn out to be false positives. This suggests that much of the preparation effort is unproductive and that the development of improved preparation techniques may significantly increase overall effectiveness.

Our observed gain rates are much higher than those reported by Votta[22] . Explanations include three possible causes:

- Votta's study was concerned with design documents rather than code;
- the average team size for a design review is larger than for code inspections;
- design reviewers may prepare much more thoroughly since design defects have wider impact than code defects.

4 Conclusions and Future Work

We are in the midst of a long term software inspection experiment based on all of the code units in a real 5ESS software development product. We are assessing several inspections methods by randomly assigning different team sizes, combinations of reviewers, numbers of inspection sessions, and author repair activities to each code unit. To date we have completed 17 of the planned 100 inspections. We expect to finish the remaining 83 inspections by the end of 1994.

Preliminary results of our empirical study of the effectiveness of various software inspection methods challenge certain long-held beliefs about the most efficient way to conduct inspections.

Judging from the percentage of defects discovered, we are finding that two-session-two-person (2sX2p) inspections appear to be the most effective. The difference in effectiveness between 2sX2p inspections and other treatments also show that number of sessions and number of reviewers per session are important factors affecting efficiency.

Two-session-two-person-with-repair (2sX2pR) inspections seem to be slightly more effective than two-session-two-person-with-no-repair (2sX2pN) inspections. However, repairing defects between sessions costs 5 extra working days of inspection interval.

We believe that when all the inspection data has been collected and analyzed, the answers to the following questions about software inspections will emerge:

1. Are some inspection methods significantly more effective than others?
2. What is the most efficient number of reviewers per inspection?
3. How many review sessions per inspection will give the best results?
4. Are multiple session inspections more cost beneficial than single session inspections?
5. Should author repair be done between review sessions?

Finally, we feel it is important that others attempt to replicate our work, and we are preparing materials to facilitate this. Although we have rigorously defined our experiment and tried to remove the external threats to validity, it is only through replication that we can be sure all of them have been addressed.

Acknowledgments

We would like to recognize the efforts of the experimental participants – an excellent job is being done by all. Our special thanks to Nancy Staudenmayer for her many helpful comments on the experimental design. Our thanks to Dave Weiss and Mary Zajac who did much to ensure we had all the necessary resources and to Clive Loader and Scott VanderWiel for their valuable technical comments. Finally, Art Caso's editing is greatly appreciated.

References

[1] Barry Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, January 1984.

[2] F. O. Buck. Indicators of quality inspections. Technical Report 21.802, IBM Systems Products Division, Kingston, NY, September 1981.

[3] K P Burnham and W S Overton. Estimation of the size of a closed population when capture probabilities vary among animals. *Biometrika*, 65:625–633, 1978.

[4] John M. Chambers, William S. Cleveland, Beat Kleiner, and Paul A. Tukey. *Graphical Methods for Data Analysis*. Wadsworth International Group, Belmont, California, 1983.

[5] Stephen G. Eick, Clive R. Loader, M. David Long, Scott A. Vander Wiel, and Lawrence G. Votta. Estimating software fault content before coding. In *Proceedings of the 14th International Conference on Software Engineering*, pages 59–65, May 1992.

[6] Stephen G Eick, Clive R Loader, M. David Long, Scott A Vander Wiel, and Lawrence G Votta. Capture-recapture and other statistical methods for software inspection data. In *Computing Science and Statistics: Proceedings of the 25th Symposium on the Interface*, San Diego, California, March 1993. Interface Foundation of North America.

[7] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):216–245, 1976.

[8] P. J. Fowler. In-process inspections of work products at at&t. *AT&T Technical Journal*, March-April 1986.

[9] D. P. Freeman and G. M. Weinberg. *Handbook of Walkthroughs, Inspections and Technical Reviews*. Little, Brown, Boston, MA, 1982.

[10] Watts Humphrey. *Managing the Software Process*. Addison-Wesley, New York, 1989.

[11] *IEEE Standard for software reviews and audits*. Soft. Eng. Tech. Comm. of the IEEE Computer Society, 1989. IEEE Std 1028-1988.

[12] John C. Kelly, Joseph S. Sherif, and Jonathan Hops. An analysis of defect densities found during software inspecitons. In *SEL Workshop Number 15*, Goddard Space Flight Center, Greenbelt, MD, nov 1990.

[13] John C. Knight and E. Ann Myers. An improved inspection technique. *Communications of the ACM*, 36(11):51–61, November 1993.

[14] Dave L. Parnas and David M. Weiss. Active design reviews: principles and practices. In *Proceedings of the 8th International Conference on Software Engineering*, pages 215–222, Aug. 1985.

[15] Kenneth H. Pollock. Modeling capture, recapture, and removal statistics for estimation of demographic parameters for fish and wildlife populations: Past, present, and future. *Journal of the American Statistical Association*, 86(413):225–238, March 1991.

[16] Adam A. Porter and Lawrence G. Votta. An experiment to assess different defect detection methods for software requirements inspections. In *Sixteenth International Conference on Software Engineering*, Sorrento, Italy, May 1994.

[17] Glen W. Russel. Experience with inspections in ultralarge-scale developments. *IEEE Software*, 8(1):25–31, January 1991.

[18] G. Michael Schnieder, Johnny Martin, and W. T. Tsai. An experimental study of fault detection in user requirements. *ACM Trans. on Software Engineering and Methodology*, 1(2):188–204, April 1992.

[19] Sidney Siegel and Jr. N. John Castellan. *Nonparametric Statistics For the Behavioral Sciences*. McGraw-Hill Inc., New York, NY, second edition, 1988.

[20] T. A. Thayer, M. Lipow, and E. C. Nelson. *Software reliability, a study of large project reality*, volume 2 of *TRW series of Software Technology*. North-Holland, Amsterdam, 1978.

[21] Scott A. Vander Wiel and Lawrence G. Votta. Assessing software design using capture-recapture methods. *IEEE Trans. Software Eng.*, SE-19:1045–1054, November 1993.

[22] Lawrence G. Votta. Does every inspection need a meeting? In *Proceedings of ACM SIGSOFT '93 Symposium on Foundations of Software Engineering*, pages 107–114. Association for Computing Machinery, December 1993.

[23] Alexander L. Wolf and David S. Rosenblum. A study in software process data capture and analysis. In *Proceedings of the Second International Conference on Software Process*, pages 115–124, February 1993.

[24] E. Yourdon. *Structured Walkthroughs*. Prentice-Hall, Englewood, NJ, 1979.

An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development

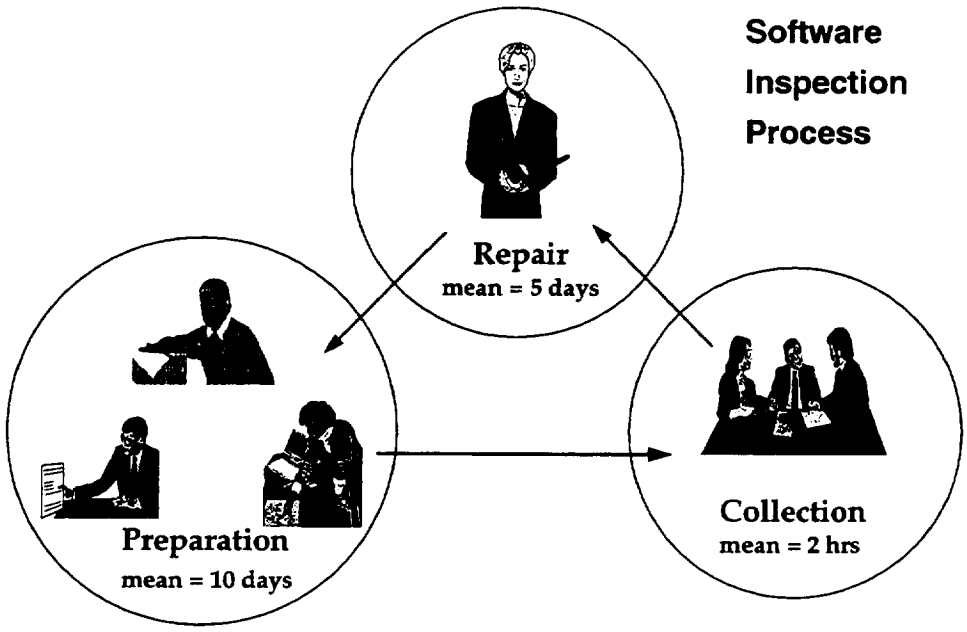
Adam Porter
Harvey Siy

Computer Science Dept.
University of Maryland
College Park, MD 20742
aporter@cs.umd.edu

Carol Toman
Lawrence Votta

Soft. Production Res. Dept.
AT&T Bell Laboratories
Naperville, IL 60566
votta@research.att.com

November 30, 1994



- Many organizations use a three-step inspection process
- Interval - ready for inspection -> completion of repair

Background/Motivation

- Competing views
  - number of sessions: single vs. multiple
  - collection meetings: yes vs. no
  - team size: large vs. small
  - coordination of multiple sessions: parallel vs. sequential
- Empirical validation
- Costs are ignored
  - interval is not normally considered an inspection cost

Hypotheses

- Inspections with larger teams have longer inspection intervals; but do not find significantly more defects.
- Collection meetings do not significantly increase detection effectiveness.
- Multiple-session inspections are more effective than single-session inspections, but significantly increase inspection interval.

Experimental Setting

- AT&T 5ESS
  - 3000 software developers
  - hierarchical organizational structure
- Legacy system
  - 1982
  - design lifetime 20 years
- Other
  - ISO 9001 certified, SEI Level 2
  - 5 MNCSL each in product and support tools
- Project
  - compiler 30K new C++, 6K reused from prototype
  - 6 software developers, plus 4 extra inspectors

Experiment Variables

- Independent
  - number of reviewers per session (1, 2, 4)
  - number of inspection sessions (1, 2)
  - repair between multiple sessions (N, Y)
- Dependent
  - inspection interval (working days)
  - observed defect density (defects/KNCSL)
  - meeting gain rate
  - meeting suppression rate

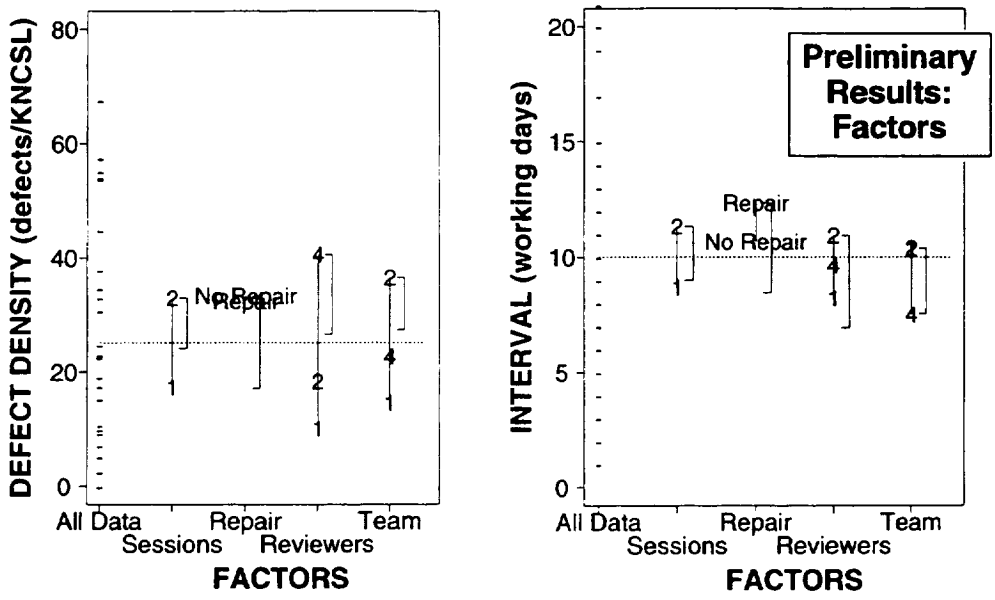
Experimental Design

Number of Reviewers	Number of Sessions			Totals
	1	2		
		Repair	No Repair	
1	1/9	1/9	1/9	1/3
2	1/9	1/9	1/9	1/3
4	1/3	0	0	1/3
Totals	5/9	2/9	2/9	1

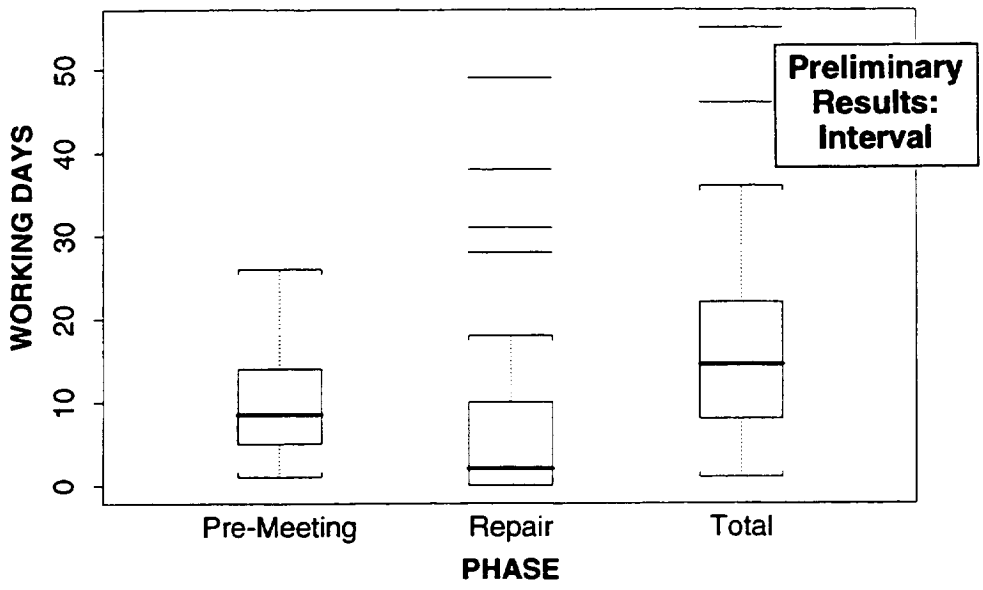
- 2 session, 4 person treatments too expensive
- 1 session, 4 person treatment is common practice

Experimental Validity

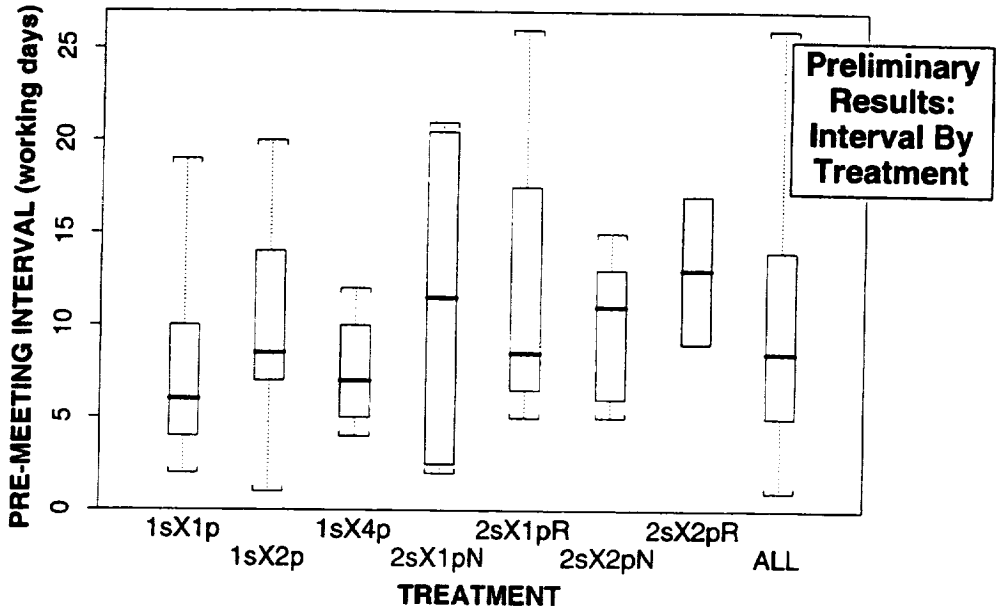
- Internal
  - selection (natural ability)
  - maturation (learning)
  - instrumentation (code quality)
- External
  - scale (project size)
  - subject generalizability (experience)
  - subject representativeness (random draw from population)



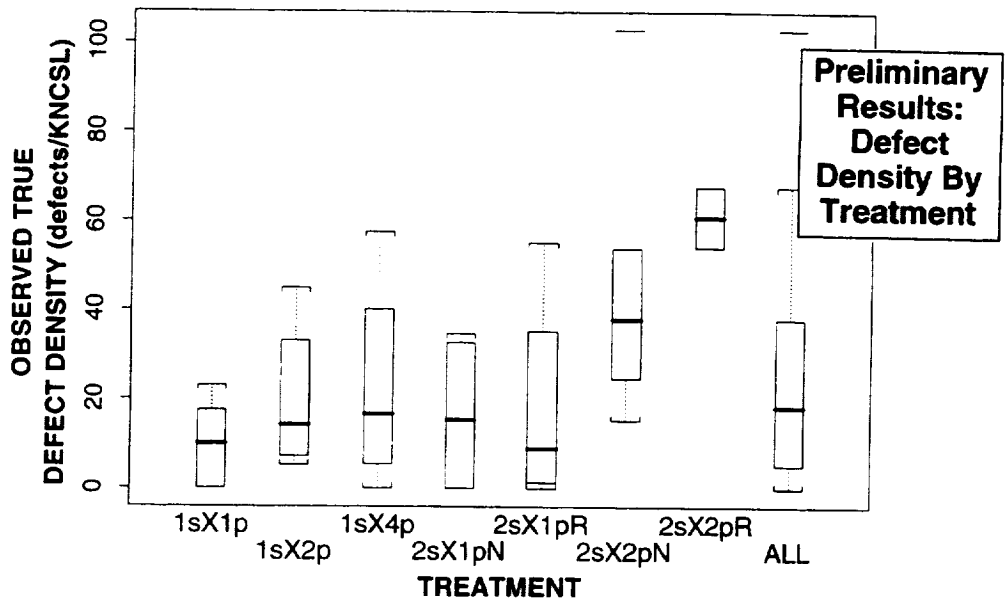
- **Density: Sessions, Reviewers, Team significant**
- **Interval: no significant factors**



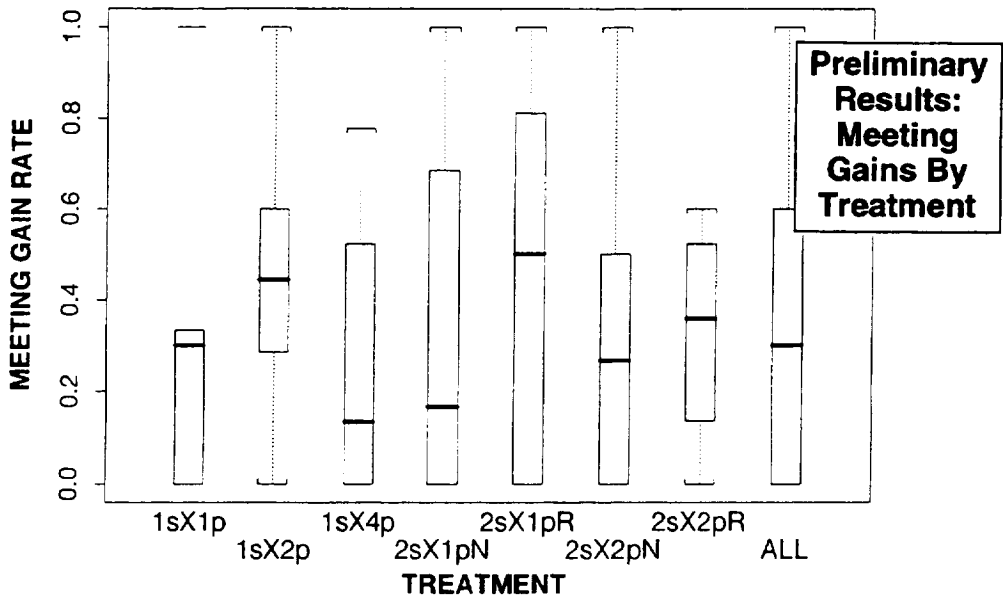
- **Medians: pre-meeting = 8.5 days, total = 14.5 days**
- **Delayed repair sometimes inflates interval data**



- **Distributions are similar**
- **2sX2pR takes longest: median of 13 days**
- **2sX2pN has median of 11 days**



- **2sX2p treatments are the best**
- **2sX2pR better than 2sX2pN by 35%**
- **Team size makes no difference for 1-session treatments**



- **Overall median meeting gain rate is 0.3**
- **1sX4p has lowest meeting gain rate**

Results To Date

- **2sX2p are most effective inspection method.**
- **Repair between improves detection effectiveness by 35 %; at a cost of 2 additional working days of interval.**
- **1sX2p are as effective as 1sX4p but most 1sX4p defects are found at preparation.**

Next Steps

- What makes 2sX2p inspections more effective than 1sX2p?
- Why are the interval differences not more pronounced?
- What are the cost-benefit tradeoffs of meetingless inspections?

A Process Improvement Model for Software Verification and Validation

John Callahan[1] *George Sabolish*
NASA Software IV&V Facility
West Virginia University

Abstract

We describe ongoing work at the NASA Independent Verification and Validation (IV&V) Facility to establish a process improvement model for software verification and validation (V&V) organizations. This model, similar to those used by some software development organizations, uses measurement-based techniques to identify problem areas and introduce incremental improvements. We seek to replicate this model for organizations involved in V&V on large-scale software development projects such as EOS and Space Station. At the IV&V Facility, a university research group and V&V contractors are working together to collect metrics across projects in order to determine the effectiveness of V&V and improve its application. Since V&V processes are intimately tied to development processes, this paper also examines the repercussions for development organizations in large-scale efforts.

1 Introduction

In effort to improve the quality of software products in safety-critical and high-risk projects, many organizations employ verification and validation (V&V) techniques to detect and correct errors made during the development process. Verification involves analyzing software products after each major development stage to ensure that the product agrees with the specification established prior to that stage. Validation involves ensuring that the products after each stage agree with the original specifications. Although validation is traditionally performed only at later stages (i.e., testing) with respect to requirements, we employ the broader definition.

A specific application of V&V can be characterized along three dimensions: orientation, scope, and independence. First, V&V activities can focus on either the software development process or the products produced by that process. Most V&V activities, however, perform a combination of both process-oriented and product-oriented analysis. Second, the scope of V&V activities can range from being comprehensive across all development phases, to being limited to specific subsystems and process stages. Finally, V&V activities can be embedded within or independent of a development effort. Independence can vary over levels of technical, managerial, and financial control [10].

Regardless of its organization, however, all V&V organizations are charged with detecting (and sometimes correcting) errors in software products and processes as early as possible in the development life-cycle. This implies that effective techniques must be employed that help find the most critical

---

[1]This work is supported by NASA Cooperative Agreement NCCW-0040 under the supervision of the NASA Headquarters Office of Safety and Mission Assurance (Code Q) at the Independent Software Verification and Validation (IV&V) Facility in Fairmont, West Virginia.

problems in early phases. Clear correlation must be established between these early errors and their consequences later in the development life-cycle. Otherwise, such problems can be dismissed as false warnings or non-critical.

This paper describes ongoing work at the NASA IV&V Facility to develop a process improvement model for software V&V organizations. Our effort involves establishing a framework for iterative measurement and ongoing improvement of a V&V organization's ability to find critical errors early and more accurately estimate costs and benefits of V&V. Although our model is still evolving, we are working with V&V contractors to assess the effectiveness the approach on existing projects.

2 Related Work

There is a limited amount of empirical evidence on V&V in practice, but most of the research on V&V has focused on (1) determining the cost effectiveness of V&V relative to the cost of the overall software development effort; and (2) developing methods for identifying errors as early as possible in the software development life-cycle. First, the cost effectiveness of V&V has been found to depend heavily on many factors including project size, expected lifetime of the software, volatility of requirements, and the expertise of development and V&V personnel. Secondly, even if these factors warrant the use of V&V, it is most important to determine how much, when, and what types of V&V to apply in each project. Effective methods for detecting critical errors must exist to enable an adequate appraisal of what the V&V effort saved in a project [1].

One of the most comprehensive studies of V&V [2] concludes that V&V is highly cost effective if applied early in the life-cycle of large, complex software projects. This study, conducted by NASA/JPL, consists of a survey of over 80 papers and related projects that include both quantitative and qualitative assessments of V&V cost effectiveness. The JPL study strongly suggests that many projects found V&V to be cost effective because the cost to correct latent errors grows exponentially in later life-cycle phases. According to several key papers in the JPL study [3,4,5], V&V can find errors early and avoid the costs of fixing latent errors. Overall, the JPL study suggests that V&V can pay for itself if started in the requirements phase, but also that V&V can negatively impact a project if started late.

In addition, several papers examined in the JPL study conclude that V&V also has benefits such as significantly reduced software maintenance costs [3,6,7]. These studies find that V&V more than pays for itself in projects with long lifetimes due not only to increased reliability but also to decreased maintenance costs. They suggest that V&V increases external management and technical visibility that is essential in long-term projects where personnel turnover is high and requirements are volatile.

Other research has focused on developing effective V&V methods for detecting errors. Many of these methods are specific to software application domains, development processes, and specification techniques. Some methods have proven nominally effective and even ineffective when applied incorrectly [8,9]. For example, a formal verification of code is considered too costly in low-risk projects. Although a formal verification would increase reliability, it would not be cost effective relative to the impact of errors. In this case, the cost of finding the errors exceeds the cost of the error occurring plus the cost of fixing the problem. The high costs of formal verification, however, can be justified in some safety-critical applications where the costs of failure can be catastrophic.

Finally, there are several reports that advocate the use of V&V based on case studies and expert opinion [7,10]. For example, the NRC assessment of Space Shuttle flight software development [10] strongly advocates the continued use of V&V on Shuttle and other large NASA projects. The NRC committee advises that independent V&V can be highly cost effective and useful in avoidance of catastrophic incidents in large projects because it provides visibility into highly complex interactions (often informal) between large numbers of contractors. Because of the informal nature of many of these interactions and

the high turnover of personnel in large projects, an independent V&V contractor can provide continuity over the long-term on large projects and provide management and technical visibility to the customer.

3 Process Improvement for V&V

We are engaged in establishing a process improvement model for V&V organizations at the NASA IV&V Facility [11]. Our objective is to establish criteria for measuring V&V activities, measure on-going V&V projects, and suggest incremental improvements to both product analysis and a V&V process. Although our collaborations are primarily with highly independent V&V groups, small V&V groups are also involved within specific projects.

To accomplish our objective, we are building a process improvement model for V&V based on measurement of products and processes from both development and V&V efforts. Our proposed model is based on the NASA GSFC Software Engineering Lab's Process Improvement Paradigm that uses measurement as the basis for determining the effectiveness of our efforts to introduce improvements into V&V processes. In general, a process improvement model iterates over the following steps:

1. *Measure* the current process;
2. *Analyze* strengths and weaknesses;
3. *Improve* the process by developing and introducing new technologies to addresses weaknesses;
4. *Measure* the process to determine the effectiveness of the improvement;
5. Repeat steps 2, 3, 4.

Figure 1 depicts an overview of the V&V organization and research group in context of a development process. The next sections describe the aspects of measurement in the V&V process improvement model: cost effectiveness, trend analysis, and error detection.

3.1 Measuring Cost Effectiveness

What is the value of V&V to a project? If V&V finds errors early in a project's life-cycle, what are these worth in terms of cost avoidance to the project in the long-term? Several models of cost avoidance estimation have been proposed in the literature [12,13], but they are very general and many assume that errors are not caught by development until testing at the end of the development life-cycle. More sophisticated models exist, but they are specialized with respect to development and V&V processes.

We propose a framework that can be customized for specific projects to track the cost of fixing errors in each life-cycle phase. The framework is based on existing cost estimation models and provides an evolutionary approach to improving the accuracy of cost-savings estimates throughout the lifetime of a project. This assumes that the development process is cyclic because it affords opportunities for repeated phases on the same project. Fortunately, our experimental V&V projects have cyclic development processes that consist of multiple releases over an extended maintenance phase. It is anticipated that the projects will incur significant functional changes that must undergo cyclic development phases.

For example, if a number of major problems are uncovered during the first requirements analysis phase of V&V, the cost savings can be estimated based on existing models within a wide confidence range [1]. In the next iteration of the requirements phase, we can better estimate the cost savings based on knowledge of costs to fix errors in previous iterations of phases for that project. This allows for increased accuracy of estimates and confidence in V&V assessments.

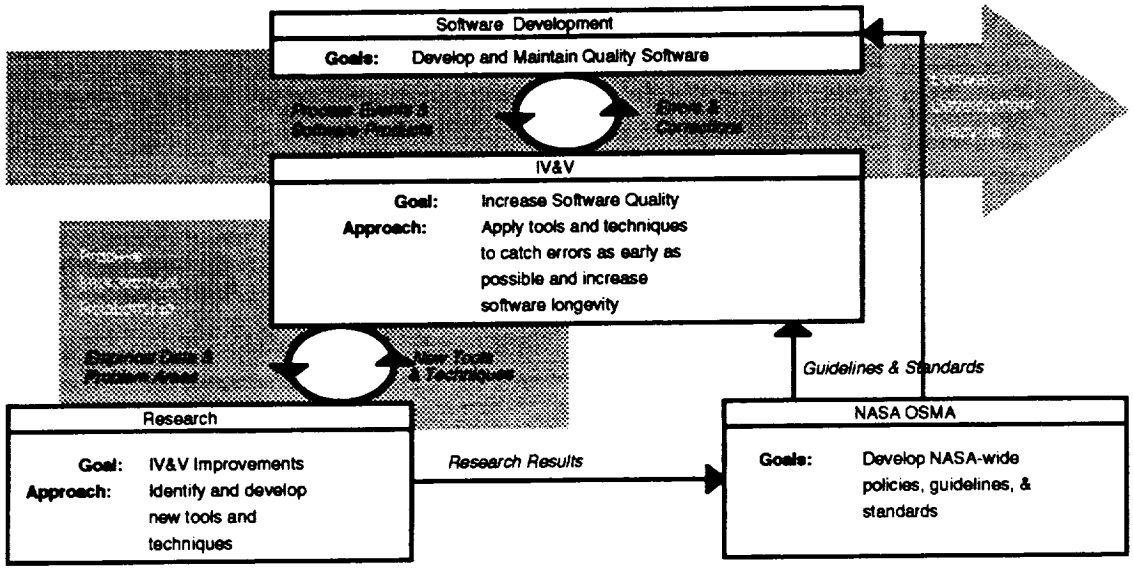


Figure 1: An overview of the V&V process improvement model

Part of our effort also involves factor analysis of V&V measurements to assess their impact on identifying potential problems. A V&V analysis may find problems, but these problems may be of high, moderate, or low impact. It is often difficult to assess the value of a technique at finding high-impact errors. More research is needed to identify effective techniques and incorporate them in V&V processes.

3.2 Trend Analysis

Can V&V help predict problems? The status of a project is more than the analysis of its parts. While the individual product errors may not be severe in a project, their cumulative effect can be serious. V&V efforts will yield analysis in the form of metrics on development processes and products. These metrics can be used by a V&V organization to predict trends that may result in schedule slippage, increased errors, costs, and other composite effects. It is necessary for a V&V organization to spot process problems early in the life-cycle and must have effective means to predict them. Our model relies on the cyclic phases of development to allow us to identify trends in software processes based on the analysis of correlation to find leading indicators in a project [14,15] that foreshadow potential problems. Once these indicators are identified and validated, they can also increase the accuracy of estimates and confidence in V&V assessments.

V&V has also been shown to have an influence on software reliability and maintenance. We are still modifying existing models to incorporate the ability to estimate the impact of V&V on reliability and maintainability. These qualities, however, are very difficult to quantify and only meaningful in the context of a project's goals. We are still exploring ways of quantifying such qualities in our model so that the full value of V&V on a project can be assessed.

As we identify improved V&V measurements and techniques, we will need to introducing new methods into the V&V life-cycle. Again, the cyclic nature of our associated projects allows for the incorporation of changes at strategic points in the process. Like the SEL model, our on-going measurements will allow us to assess the impact of such changes on the effectiveness of V&V.

3.3 Error Detection

How much and what types of V&V are required on a project? It is necessary to improve the ability of V&V to find problems in a software development project and focus analysis on the most critical aspects of development products and processes. Our framework will analyze the success and failure of existing V&V techniques to detect specific errors by auditing errors (i.e., V&V discrepancy reports) backward in the V&V process. Auditing these problems should help identify gaps in the V&V processes. For example, errors can be missed due to several problems in the V&V process including:

- *Omission.* The problem was caused by an error that could have been caught by the V&V process, but was overlooked due to the lack of V&V personnel expertise or the difficulty in applying the analysis;
- *Incompleteness.* The problem could have been avoided via existing techniques but the lack of information from the development process prevented its application;
- *Lack of Resources.* The problem could have been found but there was insufficient time or personnel needed to find it;
- *Lack of Capability.* The problem was caused by an error that could not have been caught by the V&V process because of the inadequacy of the methods and tools involved or the inherent complexity of the error.

This is not a complete list of reasons why errors are missed, but they are typical of the way in which errors can be classified in order to help improve detection of errors in earlier life-cycle phases. Analysis of classified V&V errors can lead to discovery of common types of errors that may suggest new methods, specifications, or processes.

4 Approach

The need to change V&V methods as part of an ongoing improvement program will impact the development process. For this and other reasons, much debate has surrounded the need for V&V. Some argue that it is more important to improve the quality of the development organization. It is beyond the scope of this paper to completely sort out the arguments, but we see the two views as compatible. A V&V should not simply assess the status of a development effort, but also provide feedback for improvement of the development process itself. In other words, V&V can act as a process improvement organization for development. The next sections describe our long-term strategy related to this view and our short-term tasks for achieving this goal.

4.1 Long-Term: Verifiable Development Techniques

Initially, we are focusing on the ability of the V&V process to find problems effectively and not on improving the capabilities of the software development process itself. However, because V&V and development are intimately related processes, we have developed a strategy for transferring improvements to development processes based on the need for improvements in V&V.

Our long-term strategy is to demonstrate that changes to development are needed in cases where V&V is unable to perform its task due to inappropriate or unavailable information from development. The goal of process improvement on a development organization is to enable it to produce high-quality software, on time, and within budget. This implies that the development effort is predictable and measurable. Ultimately, this will lead to development techniques that are highly amenable to V&V activities. We have labeled these *verifiable development techniques* (VDTs) to identify them as enabling effective V&V over

other approaches. A verifiable development technique is comprised of many different phases that are highly amenable to V&V. For example, the requirements for a safety-critical project might be expressed in specification language that is amenable to formal analysis. In a VDT, such analysis is not simply a spot check but coordinated with analyses performed in other phases.

4.2 Short-Term Tasks

Current research activities are focused on the short-term tasks to construct the V&V process improvement framework. The framework is needed to form the basis of any future improvements in the area of V&V. While it is true that V&V activities have been conducted on projects for many years, industry has yet to define and document V&V processes involved with any degree of consistency. Working with real projects using real project data gives our research effort the unique ability to define a baseline set of processes that can then be improved through use of a structured improvement process.

Many metrics, models, techniques and processes exist that can be incorporated into our framework. We must identify those that currently exist and attempt to formulate the characteristics of new approaches. Our short-term tasks related to our long-term vision include:

- *Metrics*. We have identified some metrics that are highly effective in predicting the potential occurrence of problems in software projects. We are paying particular attention to existing metric "success" stories and studies. In addition, we are examining the "Hawthorne effect" in software development that occurs when a V&V organization is employed. We are working with the NASA Langley SEES effort to establish V&V baselines and compare experimental results of employing V&V.

- *Processes*. We are examining existing development processes and determining how to map V&V processes to them. In addition, we are examining V&V as related to non-standard development processes, particularly in large-scale projects where requirements change dramatically during development.

- *Classification*. Because V&V cannot be applied uniformly across all phases and products due to resource limitations, we are seeking means to classify software products according to their impact on system failure. Such classification schemes will help tailor V&V processes to direct their attention to appropriate problems.

- *Testing*. This traditional role of V&V cannot be totally ignored, but we plan to move "testing" to earlier stages in the software development life-cycle. For instance, a "test" of the requirements specifications can be posed as a challenge to be disputed by some analysis on the project requirements. We are also exploring the possibility of evolving early tests into executable test suites.

Work in these areas will help establish the criteria for validating our framework employed on ongoing projects. They are needed to establish means of assessing the cost estimates and error detection methods at all phases of the development and V&V life-cycles.

4.3 Validation Through Application

The concept of "Strategic Alliances" formed between government, industry and academia plays a critical role in the process of validating research artifacts. The research strategy used at the IV&V Facility consists of working relationships between research and select projects and organizations. Potential prospects for collaboration are selected through initial discussions that focus on determining if there is some mutual interest to serve as a basis for the collaboration. The ability to gain access to an independent research organization that has the potential to improve processes and products without disrupting the

normal schedule of project activities is usually a very attractive incentive to induce project cooperation. It provides the project with research derived information and insight that would otherwise be absent. The only cost to the activity, in return, is to supply the research organization with "real" project data that is needed to corroborate their efforts.

Figure 1 also depicts the relationship described above. It describes the relationship between a developing agent, an IV&V agent, a research agent, and a governing body. However, the process could work just as well without an IV&V agent in which case research would interface directly with the developing organization. Both cases are in effect at the Facility and seem to offer equal benefit.

For each project, software quality is achieved through process improvement. First, one must define a starting point or baseline. If improvement is to be made we must know where we are at. This, in the case of the Facility is achieved by understanding the current practices of each of the selected projects or activities and using it as a baseline. Second, there must be a method by which to measure the improvements that are made. This can be accomplished using existing project metrics augmented by the introduction of any research specific metrics that may be needed. Third, an organization is needed whose focus is the introduction and measurement of new processes and products. This is the role played by the research organization, Fourth, there must be a governing body that is responsible not only to fund the improvement process, but to transform the results into usable products through establishment of policy, standards, and guidelines that in turn can be shared throughout the industry.

In this model, research plays a crucial role. A developing agent seldom has time allocated to explore potential improvement initiatives. Project cost and schedule matters are almost always take precedence over evolving technology. Access to a research organization whose charter is technology improvement allows advances to be made with a minimum amount of impact to the developing agent. Research in turn, benefits from the real-time validation it receives because results have been derived on real projects as opposed to projections based on theory and classroom trials.

4.4 A Case Study: EOSDIS

One example of this type of collaboration is our on-going work with the EOSDIS IV&V contractor to provide V&V process improvement on a long-term development project within NASA. The EOSDIS project is well-suited because it is still in its earliest development phases and open to collaboration. It is a large project with significant risks that can benefit from V&V because its development life-cycle is cyclic due to staged releases of program functionality and anticipated upgrades. We view this has a unique opportunity to introduce a process improvement model for V&V in order to ensure increasing confidence in the face of functional enhancement and a long-term maintenance phase.

It is still too early in the EOSDIS effort for substantive measurements, but initial audits of discrepancy reports generated by V&V suggest that a major obstacle is the lack of timely and appropriate products from the development organizations supplied to the V&V contractor. For example, project schedules were provided in Gantt chart form with little information about associated effort or context. Furthermore, the time allotted to V&V to analyze the schedule did not allow the application of cost and schedule estimation models. This limited the type and extent of V&V analysis on the development schedule.

The preliminary requirements analysis of the ECS portion of EOSDIS was completed at the end of October 1994. Currently, we are in the process of performing cost avoidance estimates on the preliminary requirements analysis and assessing the effectiveness of the analysis. The cost avoidance of errors found in this early phase will be estimated based on available models and later compared with actual performance. We will also produce confidence levels associated with these estimates.

There is also serious concern in the EOSDIS V&V effort over the fidelity project requirements and designs. While several errors were found in the requirements, it is questionable whether or not they are in

agreement with current design artifacts. The V&V contractor discovered this problem and the development contractor is currently fixing it before the start of the next V&V phase.

5 Summary

The NASA IV&V Facility was established in 1994 as part of a larger effort within NASA to focus attention on software issues. It currently houses efforts related to the Earth Observing System (EOS) and Space Station projects. It also houses a university research team committed to measurement-based research on actual V&V projects. This unique environment will create a testbed for new techniques in software product and process analysis.

Ultimately, we hope to improve the quality of computer software and the organizations that develop or help develop it. This paper does not seek to justify the use of V&V in projects but to (1) establish guidelines for determining its effectiveness and (2) improve its practice. By basing our work on a sound measurements program, we hope to frame V&V effectiveness within the context of its application. We hope that our process improvement model for V&V can benefit both V&V and development efforts.

Many barriers still remain to conducting research on software development and V&V efforts. First, many vendors are reluctant to provide measurements because it will expose them to criticism. Second, visibility into proprietary techniques and processes may harm their competitive advantage. Finally, measurements provided by the measured project will always tend to be skewed optimistically. We are trying to address these barriers through memorandums of understanding and other contractual mechanisms.

On large software efforts, several agencies of the US government, including NASA, have invested heavily in independent V&V as insurance against catastrophic errors. As development methods evolve, V&V processes must also improve. Since V&V is a complementary process, its improvement will drive improvements in development. We see the relationship as mutually beneficial in achieving high quality software.

References

[1] Lewis, R., Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software, John Wiley & Sons, New York, 1992.

[2] The Cost-Effectiveness of Independent Software Verification and Validation, NASA Jet Propulsion Laboratory, 1985.

[3] Radatz, J., Analysis of IV&V Data, Final Technical Report, Rome Air Development Center, March 1981.

[4] Kosowski, E., Perspectives on Software Development and Verification - Boeing 757/767 AFDS, *Proceedings of the IEEE/AIAA 5th Digital Avionics Systems Conference*, IEEE, October 31 - November 3, 1983, pp. 6.5.1 - 6.5.4.

[5] Nicolai, R., Verification and Validation of IRAS On-Board Software, *Proceedings of the ESA/ESTEC Software Engineering Seminar*, ESA-SP-199, October 11-14, 1983, pp. 221-226.

[6] Daggett, P., M. Forshee, S. Forest, T. Fox-Daeke, G. Ingram, and D. Papa, *Handbook for Evaluation and Life-Cycle Planning for Software, Volume IV, Test and Independent Verification and Validation*, ESD-TR-84-171 (IV), 1983.

[7] Sapp, J. and C. Southworth, *Orlando I -- Final Report -- Panel B -- Independent Verification and Validation (IV&V)*, Joint Logistics Commander JPCG-CRM-CSM Conference, October 1983.

[8] McGarry, F., What have we learned in the last 6 years -- measuring software development technology, *Proceedings of the 7th Annual Software Engineering Workshop*, NASA/GSFC, December 1982.

[9] Brosius, D., *Software Validation Study*, SAMSO TR-73-99, 1973.

[10] National Research Council, An Asssessment of Space Shuttle Flight Software Development Processes, National Academy Press, Washington, D.C., 1993.

[11] NASA Office of Safety and Mission Assurance, *Proccedings of the Verification and Validation Workshop*, Morgantown, WV, December 1993.

[12] Boehm, B., *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[13] Wolverton, R., *Airborne Systems Software Acquisition Engineering Guidebook for Software Cost Analysis and Estimating*, ASD-TR-80-5025, Aeronautical Systems Division, 1980.

[14] Dyson, P., K. Dyson and J. McGhan, *Streamlined Integrated Software Metrics Approach (SISMA) Guidebook*, Software Productivity Solutions, Indiatlantic, FL, 1993.

[15] Callahan, J., T. Zhou and R. Woods, Software Risk Management Through Independent Verification and Validation, *Proceedings of the 4th International Conference on Software Quality*, American Society for Quality Control, Mclean, VA, October 305, 1994.

A Process Improvement Model for Software Verification and Validation

John Callahan
George Sabolish

NASA Independent Software Verification and Validation Facility
West Virginia University
Fairmont, WV

Overview

- Introduction, Dimensions, Objectives
- Related work
- Process Improvement for V&V
  - Cost Effectiveness
  - Trend Analysis
  - Error detection
- Approach
  - Long-term
  - Short-term
  - Collaborations
  - Case study: EOSDIS

Introduction - V&V

- Verification
  - analyze output of development phases to ensure it agrees with input specifications
  - *Are we building the product right?*
- Validation
  - output of each phase agrees with original specifications (i.e., requirements)
  - *Are we building the right product?*
- Includes many techniques
  - formal methods
  - testing
  - inspections

Dimensions of V&V

- Orientation
  - products
  - processes
- Scope
  - comprehensive
  - process limited
  - product limited
- Independence
  - technical
  - managerial
  - financial

V&V Objectives

- Find errors as early as possible
- Develop effect analysis methods
- Establish correlation between early errors and potentially latent errors

Otherwise...V&V analysis can be refuted or dismissed as non-critical by development

Related Work

- Two types
  - Cost avoidance models
  - Analysis methods
- Conclusion: V&V is cost effective if...
  - started early
  - on large, complex projects
- Quantitative studies show
  - Significant reduction in maintenance costs
  - Effectiveness is dependent on many factors
    - » project size, requirements volatility, expertise, ...
    - » techniques used

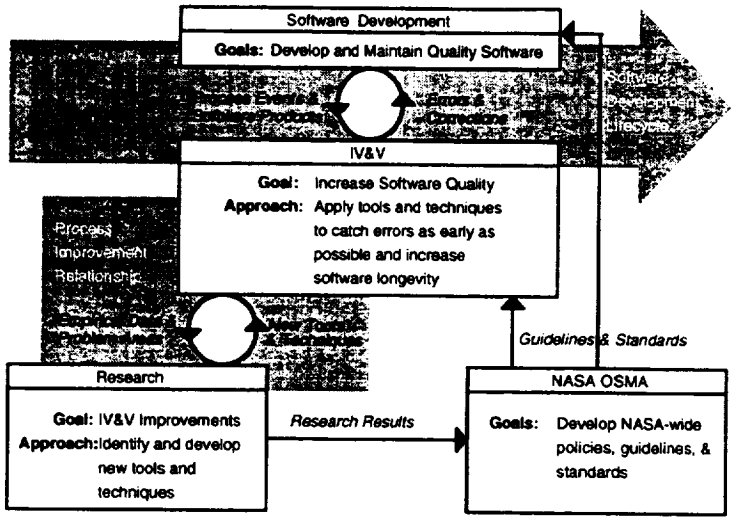
Related work (con't)

- V&V provides
- visibility into complex interactions between vendors in large, complex projects
- continuity on large, complex projects in face of personnel turnovers

Sources: NRC STS Assessment, JPL IV&V study, Orlando I Report, Mitre V&V report, Space Division Management Guide, Lewis IV&V book, ...

Process Improvement

- Objective: Improve V&V Processes
- Strategy: Apply SEL Process Improvement Paradigm
  - Measure the current process
  - Analyze strengths & weaknesses
  - Improve the process through new technology & methods
  - Measure the process to determine effectiveness of change

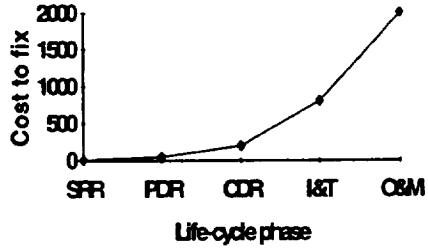


Aspects of V&V Measurement

- **Cost Effectiveness**
  - What is the value of V&V?
- **Trend Analysis**
  - Can V&V help predict problems?
- **Error Detection**
  - How much and what types of V&V?

Cost Effectiveness

- What is the value of V&V to a project?
- Finding errors early in life-cycle
- Improve estimation of cost avoidance



Trend Analysis

- Can V&V predict problems?
- Assess the cummulative effect on
  - schedule
  - cost
  - effort
  - error trends
- Need factor analysis on current V&V to determine effectiveness
- Improve predictive capabilities of V&V

Error Detection

- How much and what types of V&V?
- Audit errors backwards in process
  - Omission
  - Incompleteness
  - Lack of Resources
  - Lack of Capability
- Will improve detection methods

Approach

- V&V can act as process improvement organization (i.e., provides feedback)
- V&V improvements will precipitate development process changes
- Long-term: Verifiable Development Techniques (VDTs)
- Short-term tasks

Long-Term: VDTs

- A verifiable development technique is
  - repeatable
  - measurable
  - ammenable to analysis
  - coordinated
- Similar to process improvement goals

Short-Term Tasks

- Metrics
- Processes
- Classification
- Testing
- Get involved in on-going projects

Case Study: EOSDIS IV&V

- Too early for substantive measurements
- Major obstacles
  - timeliness of products
  - lack of appropriate products
- Currently performing cost avoidance analysis on preliminary requirements analysis
- Investigating disjoint requirements & design
- Improving access to artifacts

ECS Analysis

- Requirements are structured via functional levels (Level 0, Level 1, ...)
- Lack of functional threads
- Lack of consistency with scenarios
- Next requirements analysis may be augmented with task thread analysis

Risk Analysis

- Based on GQM
- Continuously assess *probability* of meeting project goals
- Risk = (1-uncertainty) x Importance of Goal
- Unknown metrics contribute to uncertainty
- ICSQ 94 paper

Summary

- Many barriers remain
  - exposure to criticism
  - proprietary considerations
  - skewed measurements
- NASA, DoD, others have invested heavily in V&V
- Must improve practice of V&V
- Explore roles of V&V as development improvement agent

For more information...

- **WWW server**
  - http://research.ivv.nasa.gov/
- **Email**
  - callahan@cs.wvu.edu
  - sabolish@orion.ivv.nasa.gov
- **USPS**
  - 100 University Drive
  - Fairmont, WV 26554
- **Phone**
  - 304-367-8215 (George)
  - 304-367-8235 (Jack)
- **Fax: 304-367-8203**

Session 4: Experience Reports

Leveraging Object-Oriented Development at Ames
Greg Wenneson, Sterling Software

Lessons Learned in an Organization Transitioning to an Open Systems Environment
Dillard Boland, Computer Sciences Corporation

Lessons Learned Deploying Software Estimation Technology and Tools
Nikki Panlilio-Yap, International Business Machines Canada Corporation

Leveraging Object-Oriented Development at Ames

Greg Wenneson and John Connell
Software Engineering Process Group
Sterling Software at NASA Ames

ABSTRACT

This paper presents lessons learned by the Software Engineering Process Group (SEPG) from results of supporting two projects at NASA Ames using an Object Oriented Rapid Prototyping (OORP) approach supported by a full featured visual development environment. Supplemental Lessons Learned from a large project in progress and a requirements definition are also incorporated. The paper demonstrates how productivity gains can be made by leveraging the developer with a rich development environment, correct and early requirements definition using rapid prototyping, and earlier and better effort estimation and software sizing through object-oriented methods and metrics. Although the individual elements of OO methods, RP approach and OO metrics had been used on other separate projects, the reported projects were the first integrated usage supported by a rich development environment. Overall, the approach used was twice as productive (measured by hours per OO Unit) as a C++ development.

Combining Object Oriented (OO) methods with a Rapid Prototyping (RP) approach supported by a rich development environment holds promise for highly productive development done right the first time. This combined Object Oriented Rapid Prototyping (OORP) approach was used on several projects at NASA Ames and measured over twice as productive as C++ productivity metrics collected by Capers Jones of Software Productivity Research. These projects were supported by training, consulting and mentoring from the Software Engineering Process Support Group (SEPG). Conclusions and lessons learned are presented here for two of the projects now in production: NASA Science Internet (NSI) Service Request (NSR) Tracking System and SoftLib, a reusable software library management system, internally developed by the SEPG.

SEPG Presence, Supported Methods and Approach

The Sterling SEPG acts as a software and process clearinghouse while providing no or low cost engineering and software process, methods and consulting support for contract staff and NASA scientists at Ames Research Center. The SEPG locates, adapts and champions new technology and productivity improving methods primarily as a demand driven resource. We promote several primary methods, approaches and tools which we support by providing training, process guidebooks, consulting, tools and procurement assistance, analysis and design assistance and project mentoring. When requested, we will approve methods and approaches if they are defined by published works and leveraged by available tools, but we prefer a more proactive support presence. Our preferred methods and approaches are:

- The integration of Coad-Yourdon object-oriented analysis (OOA) and design (OOD) methods with a rapid prototyping development approach
- OO Software sizing metrics
- High-level visual-programming development environments.

The Coad/Yourdon (C-Y) methods were selected because they are moderately simple, easily taught and provide consistent analysis and design representation. During this last year with

newly published works by Booch, Yourdon and others, object methods are converging and borrowing the best from each other. Thus Ivar Jacobson's Use Cases and other current approaches are being incorporated into the methods we support. We find these methods and approaches are scaleable for small and large project size in simple to complex problem domains.

The SEPG supported approach is to use C-Y OOA/OOD methods [1, 2] combined with the formal *Object Oriented Rapid Prototyping* approach defined in the new Yourdon Press book by that title [3]. This involves evolutionary development with refinements based on feed back from customer hands-on experimentation during approximately one to two week iteration cycles. The process model for this approach is shown in Figure 1. The identification of customers (requirements owners), their level of involvement and their buy-in are obtained up-front. An initial analysis and an OO model are produced in the first few days of the project for early project planning and then iterated concurrently with the prototype through many incremental additions and refinements. Formal inspections of requirements and design specifications occur at two or three points during this evolution:

- Before prototype development
- After user approval of prototype, before tuning
- Any other time the development team feels a need to resolve emerging design issues.

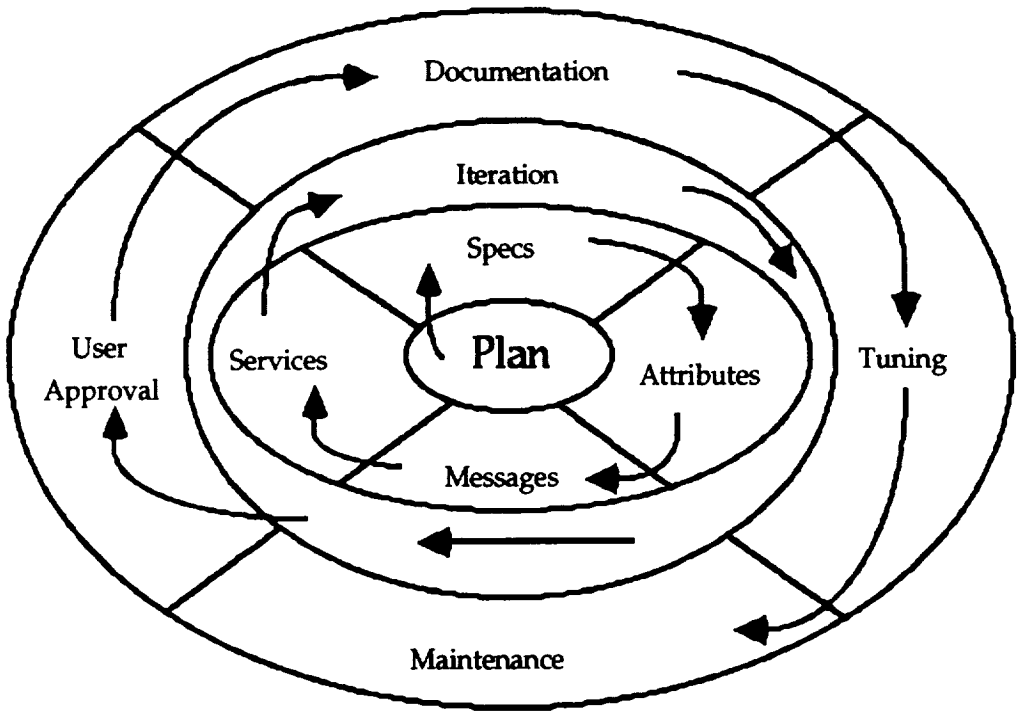


Figure 1 Object-Oriented Rapid Prototyping (OORP) Process

We have been experimenting with some new OO sizing and estimating metrics at Ames. These metrics are similar to those presented by Lorenz [4] but were actually derived as a modification of Dreger's Function Point Analysis [5] adapted to OO methods. The OO Unit metrics were first published in a paper by Connell and Eller in 1992 [6]. OO Unit metrics for components (classes/objects) and services (methods) are given OOU counts depending on the number of attributes in the object. An object with 8 attributes is of average complexity and has an OOU metric of 5. Each of the services would also count at 5 OOUs. Services have different counts

for add/modify/delete, output, computationally intense, and system service but are clumped here for simplicity. External Entities are the sources and sinks of a Source Sink Diagram which defines the system boundaries. External entities receive counts dependent on the number of interfaced objects in the system. Figure 2 provides guidelines for determining OO Unit metrics counts.

OO Units provide an advantage over the Lorenz sizing metrics in that they allow for differentiating object classes according to size, depending on the number of attributes, services, complexity of services, and external interfaces. The differentiation scale is based on a similar scale provided by Dreger and Capers Jones

	Simple < 7 Info Items	Average 7-14 Info Items	Complex > 14 Info Items
Component	3 OOUs	5 OOUs	8 OOUs
Service	4 OOUs	5 OOUs	6 OOUs
External Entity	< 3 Components 7 OOUs	3-5 Components 10 OOUs	> 5 Components 15 OOUs

Figure 2 Object-Oriented Unit Metrics Matrix

Using the C-Y OOA/D methods and Connell/Shafer rapid prototyping approaches, an early estimate of total effort can be made from the initial analysis and OO model generated at project startup. The OO unit metrics are counted from the initial model using the number and complexity of the objects, services and external interfaces. The final delivered application usually grows during prototype iteration to three times the size of the initial model. The estimated times to develop the initial prototype and then the fully deployed application are dependent upon the implementation language and environment. In our estimates, we used a figure of 4 hours to implement an OO Unit, equivalent to the figure Dreger uses for Objective C and Smalltalk. We reasoned that a powerful visual programming environment would be at least as effective as ObjectiveC and Smalltalk. Our project's end results produced figures equal to or better than that, 4 hours per OOU for one project and about 3 hours to deliver an OOU on the other.

We recommend use of high level visual programming environments for development and iteration of rapid prototypes. Ideally, a powerful development environment would provide integrated capability to manipulate GUI, control, functionality and data management abstractions at a higher level than coding in a 3GL. This is still the holy grail of development environments. While waiting for that future momentous unveiling and heeding the current (Summer 1993) call of requirements, we evaluated numerous vendors and selected Sybase's GainMomentum product as a development environment which met selection requirements. GainMomentum (here after referred to as Gain) provides object-oriented visual development tools for RDBMS access, graphic user interface, and user defined objects. Gain also has extensive function libraries, a good debugging capability, and a 4GL scripting language (GEL - Gain Extension Language) to augment visual development tools. Programming in C/C++ code is generally not required. There is an instant context switch from edit to run mode and standalone run-time executables can compiled when an application is complete to restrict user access. Gain was available for Unix environments only, though recently a Windows version was released.

For analysis and design modeling support we used the drawing and data dictionary capabilities of Iconix's ObjectModeler for the Macintosh. Although ObjectModeler can generate code templates, we only used the drawing capabilities of the tool. Consequently, the object models and the tool's capabilities were not integrated into the developers' environment. One project elected to use a Macintosh drawing tool with just as effective results.

For both projects, SEPG members acted as external consultants, trainers and mentors. Just in time training was provided in C-Y OO methods, Rapid Prototyping, development tools and management approaches. The SEPG also provided development tools during the early stages of development so that development could get started on the right foot while project startup procurements proceeded concurrently. During early stages of development, SEPG members provided hands-on assistance with OOA and OOD modeling, prototype development, prototype iteration and refinement methods, estimating, and planning support in conjunction with project staff. When staff were completely comfortable with the methods, they assumed all development activities from the SEPG.

The Projects' Specifics

Both the NSI and SoftLib projects were small and low risk. NSI planned for staff at 3 Full Time Equivalents (FTEs) and the SoftLib project was planned at about 1 FTE. Due to personnel and organizational changes, neither project reached their full planned staffing. The NSI project was estimated to take 6 calendar months and the prototype was approved and completed in 7 months. When the approved prototype was delivered, the users required further work which was completed 4 months later. The SoftLib development was initially scheduled to take 11 months and completed on schedule. With the organizational changes, we consider both projects to have completed within projected time and costs. Project effort and metrics are discussed in the next section.

Both of the projects used inexperienced staff assisted by SEPG consultants. The projects were the staff's first introduction to object methods, rapid prototyping, full life cycle implementation, advanced development environments, RDBMS and SQL.

The NSI project developed a new application to manage Internet connectivity requests stemming from world-wide NASA science projects. The development involved creation of two complex data entry forms: the NSI Service Request (NSR) and the Request for Service (RFS). The combination of these two vehicles and supporting data structures provides on-line entry of customer profile and organization data, funding authorization, and Internet service connectivity requirements. The application replaced and integrated manual and ad hoc systems for several groups, adds new functionality and provides the opportunity for further automation.

The SoftLib project re-engineered to modernize an existing reuse library management system. The old system provided a character based front end to a database of metadata about software components. Users had to grapple with the character mode interface to find reusable software descriptions and then locate the actual software outside the domain of the library management system. The new application provides a graphic X-Windows user interface to increased capabilities. Combo-box list widgets now provide selectable keywords and other search parameters. When users find interesting component descriptions in the hit list, the location is presented and they can download the file using another window. The application also provides interfaces to other applications such as a New Technology Database and other reusable libraries including a NASA-wide BBS.

In addition to the commonalty in development approach, inexperienced staff, estimating metrics and visual development tools used, these two projects had certain other elements in common. Both were in environments where users and developers did their work on networked combinations of Macintosh and Sun workstations. The networks extended over many Macintosh zones and Internet domains within the Ames domains. Both applications required intensive user interaction and an interface to an existing relational database.

There were also several differences between the projects in that SoftLib was developed using a very new alpha/beta release of a truly object-oriented version of the Gain development environment, while the NSI project used the current production release. Mentoring on SoftLib was fairly smooth because the project was internal to the SEPG. The NSI project used multiple and conflicting sources for consulting causing some confusion and lost time due to thrashing back and forth between divergent approaches — information engineering versus object-oriented rapid prototyping.

Perhaps one of the primary differences was in user's profiles and expectations. SoftLib replaced a single text based system. The users, although from different application domains, were familiar with a single interface. Whereas on NSI the users were from different functional groups. There was no single application to replace; indeed, many users had evolved their own applications using spreadsheets to support their work. Some of the replaced functionality was being performed by data entry staff. The NSI authors felt that many of the users did not think that they would be using the system.

Access and data security were issues for both projects. NSI's solution was at the network administrator layer—disallow access outside project domains. SoftLib specifically needed to allow access and file download capability throughout the Ames domains but not to outsiders. The initial design was for security daemons, user accounts and client-server pairs for file transfer. The access and security features were written in C due to apparent limitations of the Gain environment. Very late in the development, the entire SoftLib security/file transfer implementation was replaced with an Xmosaic shell with "allow" access capability for the Ames domain and a separate Xmosaic window for file transfer. Distribution and installation packaging were also replaced due to portability problems of executable code to heterogeneous workstations. As a result, no software is required to be distributed to potential users and the developer has greater control over enhancements and problem fixes. All SoftLib capabilities are available (in the Ames domain) through the World Wide Web.

Results and Lessons Learned

The NSI NSR/RFS application is in production and being used. When the approved prototype was delivered, the users were not happy and required 4 additional months of part time development. Most of the users are actually on Macintoshes using MacX for X-Windows emulation, although the system was mostly developed and demonstrated on a Sun workstation. The result is that the delivered system is very different from what the users expected. The system feels slow for this application on this network. Part of the problem appears due to heavy server loading and the earlier version, reduced-capability of Gain data managers. Macintosh client performance is less than half Sun workstation performance due to remapping for MacX screen display. Also screen size and pixel density are very different, giving a degraded look and feel on the Macintosh. NSI Macs are currently being upgraded with larger screens and graphics accelerators. Because an earlier version of Gain was used on this project, much additional GEL scripting was needed for database transaction management.

The SoftLib application has recently gone into production (October 1994) after successful beta testing in August and September. The performance is faster than NSI's application and quite acceptable. It was developed in the newer Gain version and deployed on a different host. As with the NSI project, the SoftLib prototype was primarily developed and demonstrated on a Sun workstation while many users are on Macintoshes. However, with the SoftLib application, the Librarian is promoting the reuse library and is using the colorful, more capable interface as advertising leverage to attract users.

These projects are characterized as successful because they went into production and are being used. They completed within 20% of originally planned schedule and resources. The C-Y OOA/D methods were introduced, learned and used in development. The initial C-Y object class models and OO Unit metrics provided an acceptable basis for project estimating and planning. Data points were generated to calibrate the metrics methods. Connell/Shafer rapid prototyping approaches were used to iteratively generate a hands-on requirements model the users requested and then a deliverable product. A new object oriented development environment was used to produce applications which are fairly easy to change. Preliminary measurement of development time is about 3 hours per OO Unit for SoftLib and about 4 hrs/OOU for NSI. The NSI figures are higher due to the larger amount of GEL and SQL written. These results are from inexperienced developers leveraged by visual development environments. And we had fun!!!

We feel that an OO Unit is very similar to a Function Point as described by Dreger [5]. Dreger (based on work by Capers Jones) provides a list of relative effectiveness of implementation languages including 4GLs. However, Dreger only provides one single-figure productivity metric—an average of 20 hours of COBOL development to produce one Function Point. (Capers Jones [7] declines to give language-dependent single figure metrics. Jones prefers to give high-low ranges for productivity, probably to prevent comparisons in papers like this.) We generated single-figure productivity figures by taking the median of the productivity ranges provided in Dreger's and Jones' figures. Since Jones' and Dreger's figures are given in Function Points per staff month, we assumed 21 working days per month and 6 working hours per day to normalize to hours per FP. From this, we show productivity figures for C (24 hours/FP), FORTRAN (20 hours/FP), C++ (15 hours/FP), Ada (14 hours/FP) and ObjectiveC/Smalltalk (4 hours/FP).

We are not entirely comfortable with our single-figure interpretations of Jones figures. Jones' collected metrics are from a wide range of project types and environments including MIS, military, and system software among others. We feel that today's versions of the languages would permit at least twice the productivity of our medians of Jones ranges. Using that adjustment, C would be 12 hours/FP, C++ 8 hours/FP and Ada 7 hours/FP. These adjusted figures are consistent with the high end of the productivity range Jones does provide for each of the languages. Following that, what our projects with inexperienced developers accomplished in 3 and 4 hours still compares favorably to what Dreger/Jones data shows as 8 hours per Function Point in a standard OO programming language such as C++ or 12 hours in a lower level language such as C.

We feel our productivity could have been even better. We think about 10% of total effort on the NSI project was spoilage due to conflicting advice provided by competing consultants from different organizations. On both projects, productivity was lessened by the steep learning curve of multiple elements (OOA/OOD, Gain, rapid prototyping, SQL and basic development experience). We estimate that overall on-the-job learning constituted at least 30% of total implementation cost on these two projects. On both projects the majority of implementation problems and effort expended were related to overcoming the data managers and database

interface. On NSI, much GEL scripting was written to overcome the earlier version of data managers. On SoftLib, the newer production release data managers are quite powerful, but developers had to struggle with alpha versions and multiple Beta releases. All in all, we estimate it took at least 20% additional development time for each project to overcome the maturing data base interface.

Prototype size growth from initial OO model to delivered system was flat for SoftLib and about 2.5 for the NSI system. Both systems were estimated to grow to three times the OOU counts from the initial OO model to the final system. The initial SoftLib model had an unrealistically high OOU count because the graphic widgets were modeled as separate objects with services rather than services of objects. A remodeling of the SoftLib initial OO model produced a 25% lower OOU count with an actual growth of 0.5 to delivered system. The SoftLib growth was incorrectly estimated because the SoftLib model was a detailed and almost complete model of an existing application rather than an initial OO model of a future system. The actual hours needed to complete SoftLib were also less than half that initially estimated, partially due to learning from the NSI experiences.

Reuse was minimal due to the mismatched capabilities of the development environment versions and the different application domains. The overall application framework and a few of the GUI widgets were reused between the two projects. With a bit more care, several of the NSI object classes (person, organization, etc.) might have been reused within SoftLib. Many of the NSI's classes hold the possibility for future reuse in any resource management system.

In SoftLib, the Gain development environment allowed easy modification of the applications. Because very little code is written outside the development environment, the production version is still as flexible as the prototype was during iteration. The small amount of C code written to provide SoftLib security and controlled file transfer was easily replaced using the more portable Xmosaic's file transfer and security features.

There was some learning transferred from one project to the other. The SoftLib developers were able to make some use of NSI lessons learned. The different versions of Gain data managers prevented more knowledge from being transportable. The SEPG members consulting on the NSI project also consulted on the SoftLib project. That connection was lost as the project team members assumed all responsibilities from the SEPG.

There were also some harder to measure productivity loss factors. The inexperienced developers made some mistakes that more senior software engineers might have avoided. One side effect of inexperienced prototypers was the hesitation to demonstrate a prototype that didn't appear excellent. This resulted in fewer iterations and less frequent user feedback. User commitment to requirements approval was difficult to obtain. On SoftLib there was one primary developer. With better initial team building and work partitioning, communication would have improved and the primary developer's workload lessened. Both projects had to pick between a less capable GainMomentum version 2 or an in-development alpha/beta version 3. There were many handicaps to overcome with either choice. Additionally, the inexperienced developers were not always amenable to the mentoring available from the SEPG. This is because the application was their first masterpiece and suggestions and proposed alternatives were often perceived as criticism and therefore not well received.

A major lesson learned from these two projects relative to the application of the Connell/Shafer rapid prototyping approach is that delivering a system (Macintosh) with a different look and feel from the user approved system (Sun) diminishes much of the requirements stability gained from

prototype iterations. In order to achieve requirements completeness, correctness, and exactness through rapid prototyping, the following must occur:

- real requirements must exist
- correct identification of user representatives in a development plan
- establishment of requirements ownership in a development plan
- user commitment to prototype review and approval as planned.

Execution of these basic rapid prototyping principles was flawed on both projects, resulting in some user dissatisfaction. Experienced rapid prototypers know that successful rapid prototyping is an evolving team-based process owned mutually by users and developers. The Space Station Centrifuge project, for instance, proved that the OORP approach can be used to overcome group dynamics or political fragmentation problems if users become sufficiently involved in prototype iterations. On the Centrifuge project, solid requirements definition was achieved in 14 iterations over a 10 week period with approval from 100 users. These users were in three different groups (operations, controls, and human factors) each competing for system resources and requirements implementation.

Good News

These projects were sized and scheduled using estimates derived from OO metrics applied for the first time to real projects at Ames. A conservative factor of four hours per OO unit was used for NSI project estimating. The actual productivity figure is just about that. On SoftLib, we used 2 hours per OO unit based on an preliminary estimate of the NSI metrics and hopes for the more mature version of Gain. Preliminary figures indicate a productivity figure at about 3 hours per OO unit. With the results and the offsetting productivity losses mentioned above, we feel the metrics have been initially validated and will continue to be used and refined. From previous and concurrent experience with other prototyping tools, we feel the metrics can be generalized for the entire class of visual-programming in very high level rapid prototyping tools similar to GainMomentum. These kinds of tools are much faster than procedural languages such as C and FORTRAN. They measure several times faster than OO languages such as C++, and hold promise to be significantly faster than OO development environments such as ObjectiveC and Smalltalk. On our projects, when such tools are combined with a formalized approach to OORP, the development time (with inexperienced developers on first time projects) has been measured at equal to any other approach we customarily use. If we adjust our project's productivity's by the estimated losses for learning curve and tool problems, we have an approach about twice as fast as the figures put forth by Dreger for ObjectiveC and Smalltalk.

The OO paradigm is difficult for many developers to master. We have found at Ames that non-complicated modeling methods assist developers in learning and users in understanding. Our modeling activities provided an easy way to depict the initial requirements and explain them to the user before the first prototype was started. The model also acted as an alternate design mechanism with the alternatives or not-yet-built components shown in a different color or otherwise called out. The modeling activities paralleled or led the early development; however, once the major components and inheritance were established, the modeling activity fell to a lower priority. Subsequent metrics determination required the model to be updated and reviewed—which should have been done concurrent with development.

The services provided by the SEPG proved to be valuable and particularly necessary for new developers. The learning curve was too steep for the inexperienced staff to contemplate without the training and on-project consulting provided by SEPG members. At Ames, the SEPG advises, rather than controls, projects. This means that staff may always feel free to ignore

SEPG advice. It has been found that staff are much more likely to heed the advice when it is perceived as free help rather than criticism.

In the course of these two projects a happy accidental discovery was made: there need be no difference between a good Coad/Yourdon OOD object class model and a good RDBMS schema. A mapping can be done such that each object class on the OOD model maps to a table in the database and all required database tables are modeled as object classes. This mapping is possible because the Coad/Yourdon methods work very well for data oriented applications. The methodology guidelines for identification of good object classes map well to normalized RDBMS tables. This is not to say that these methods do not work well for other kinds of applications. One of the most successful applications of OOA/OOD and OORP at Ames is the development of real-time data acquisition software for the new 250,000 LOC Standardized Unitary (wind tunnel) Data System.

Summary

The Rapid Prototyping approach combined with Object Oriented methods and leveraged by visual programming development environments show solid promise to significantly improve development productivity while generating the system the users request. Although the productivity metrics are preliminary and based on a few data points, it appears possible to easily exceed the productivity compared to creating an application with ObjectiveC or Smalltalk. The projects' productivity measures about twice as effective compared to the high productivity range of Dreger/Jones' C++ metrics. We have also shown it is possible for less seasoned engineers using these approaches and assisted by skilled mentors to exceed the productivity of seasoned developers using less effective techniques. We look foreword to measuring fully experienced developers using these highly leveraged environments.

On future OORP projects, there are some things we will do differently, as a result of these projects. We will strive to make sure that we always deliver the system the users really approved, and not slip in a new, unapproved look and feel for delivery! We will pay more attention to psychological factors in dealing with inexperienced staff and uncommitted users. We also need to keep our OO models in better synchronization with the development activity; perhaps that can identify more intentional reuse opportunities. We will try to be more thorough in assuring that original plans are carried through to ensure that users' needs are truthfully identified and responsibilities met.

We would like to compare our metrics to other OO projects in different domains and environments. We used project data captured by Lorenz and Kidd [4] to do a rough comparison to Smalltalk and C++. Their averaged data indicates Smalltalk productivity of less than 1 hour per OOU and a lower productivity for C++ at 3 hours per OOU. This three-to-one ratio is consistent with the Jones and Dreger data. The Lorenz data are from only a few projects but imply a higher productivity than our projects. However, as with the Jones data, we would need more contextual information about developer experience, environment capability and accuracy of the collected data to gauge the comparison and possibly the leveraging effect of different methods on productivity.

There are some other things these projects have caused us to think about, but we have not as yet come to any conclusions. We need to devise some more efficient means for providing expert design guidance to projects so that guidance is heeded more consistently. We need object oriented design and quality metrics in addition to sizing and estimating metrics. We also need object reusability guidelines and metrics. Furthermore we wonder if it would improve

application of SEPG services and better serve the customer if we withdrew SEPG support to a project rather than compete as one of several consulting sources.

Bibliography

1. Coad, P. & Yourdon, E. *Object-Oriented Analysis*, New York: Yourdon Press (Prentice-Hall), 1990, 1991.
2. Coad, P. & Yourdon, E. *Object-Oriented Design*, New York: Yourdon Press (Prentice-Hall), 1991.
3. Connell, J. & Shafer, L., *Object-Oriented Rapid Prototyping*, New York: Yourdon Press (Prentice-Hall), 1995.
4. Lorenz, M. & Kidd, J., *Object-Oriented Software Metrics,* New York: Prentice-Hall, 1994.
5. J. Brian Dreger, *Function Point Analysis*, New York: Prentice-Hall, 1989.
6. Connell, J. and Eller, N., "Object-Oriented Productivity Metrics", NASA Quality and Productivity Conference, 1992.
7. Jones, Capers, *Applied Software Measurement, Assuring Productivity and Quality*, New York: McGraw-Hill, Inc., 1991.

LEVERAGING OBJECT ORIENTED DEVELOPMENT at NASA AMES

Greg Wenneson and John Connell

SEPG
Sterling Software
at NASA Ames

November 30, 1994

STERLING SOFTWARE

SEPG Experiences, Lessons Learned

Combination of :

- OO Methods
- Rapid Prototyping
- OO Metrics and Estimating
- Leveraged by Tools

SEPG

Supports By:

- **Training, Consulting, Guidebooks and Tools**

"Supported" Methods:

- **Coad-Yourdon OOA/D**
- **Connell-Shafer Rapid Prototyping**
- **OO Metrics and Estimating**
- **HyperCard, JAMM, GainMomentum**

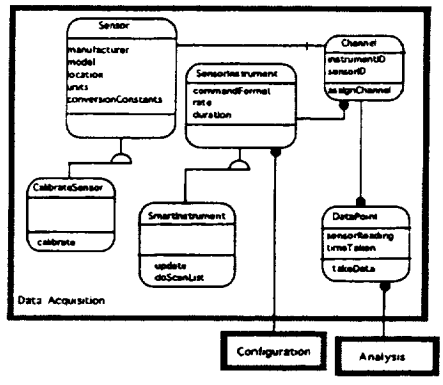
"Approved" Methods ...

Coad-Yourdon OO and...

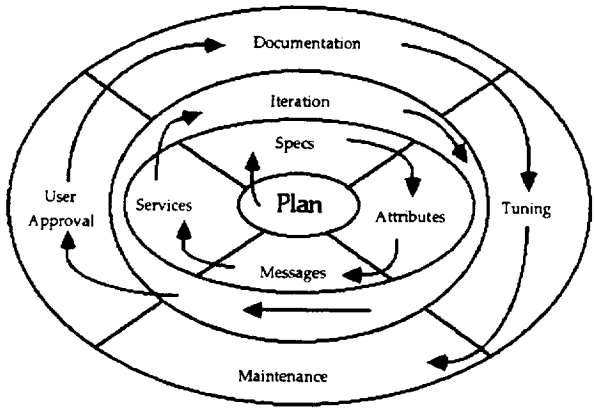
- **Object Classes, Attributes and Services**
- **Subject Layering**
- **Problem, Human I/F, Task and Data Mgt Domains**

plus

- **Source-Sink Diagram**
- **Object Control Matrix**



OORP Process Model



OO Rapid Prototyping

- **Identify Requirements Commissioners**
- **Initial Analysis, OO Model, Estimate and Plan**
- **Prototype Initial OO Model**
- **Iterate with User: ~ 1hr New Functionality**
- **Final Req'ts User Approval**
- **Tune, Re-engineer, Document, Inspect**
- **Acceptance Test and Deliver**

OO Unit Metrics Matrix

	Simple < 7 Info Items	Average 7-14 Info Items	Complex > 14 Info Items
Component	3 CUs	5 CUs	8 CUs
Service	4 CUs	5 CUs	6 CUs
External Entity	< 3 Components 7 CUs	3-5 Components 10 CUs	> 5 Components 15 CUs

- **Object Complexity by Number of Attributes**
- **External I/F Complexity by Number of Objects**
- **Effort = OOUs X Hrs/OOU X 3 (Prototype Growth)**

Visual Development Tools

GainMomentum (Selected in 1993)

- **Object Oriented**
- **GUI Development**
- **Data Management**
- **Function Libraries**
- **4GL-like Scripting Language**

Project Descriptions

NSI Service Request (NSR): Internet Connections

- Manage, Track and Schedule Resouces
- Automate Manual and Separate Systems
- Potential 100+ users

SoftLib Library Management System

- Reusable Library Component Xwindow Interface
- Re-engineer Text Based System
- Ames-wide User Base

Common Factors

- Both systems Small and Low Technical Risk
- Staff Inexperienced; Then Trained
- Introduced OO and RP
- Introduced New Development Tool GainMomentum v 2. and Beta v3.0
- Users Spread Over LANs: Macs and Suns

Results - NSI

- Planned 6 mo.; Approved Delivered in 7 mo.
- Initial Model ~140 OOUs; Delivered ~550 OOUs
- Estimated 4 hrs/OOU; Delivered ~4hr/OOU
- Performance Not as Expected
- Needed Much Additional SQL
- Delivered Approved Prototype Not Used
- 4 More Months Development
- Problems Not Technical

Results - SoftLib

- Planned 11 mos; Delivered in 11 mos.
- Initial 453 OOUs; Delivered ~ 450 OOUs
- Estimated 2 hrs/OOU; Delivered ~3hrs/OOU
- Some C Code; Replaced by xMosaic
- Newer Version of GainMomentum
- Some Structure and Widget Reuse
- System Performance Satisfactory!

What Worked

- OO & RP Work Well
- OOU Metrics and Estimates Work
- Development Tool Leverages Productivity
- SEPG Assistance Critical to Success
- Productive Development Approach

Improvement Needs

- SEPG Advice Optional
- Steep Learning Curve: 30%
- NSI 10% Multiple Consultant Spoilage
- Following RP Approach
- Non-Technical Issues
  - User Buy-In / Commitment
  - Developer Ego
- Reuse Criteria

Learned

- Deliver what Users Approve ...
- Make sure Users knowledgeably Commit
- RP Can Help Overcome Scattered Users
- Tools Have Warts - Know Them!
- C-Y OOD Obj-Class Model like RDBMS Schema
- C-Y OOA/D Methods Simple and Powerful
- Promoted Methods Leverage Productivity

Lessons Learned in Transitioning to an Open Systems Environment

Dillard E. Boland, David S. Green, Warren L. Steger
Computer Sciences Corporation
10110 Aerospace Road
Lanham-Seabrook, Maryland 20706

Abstract

Software development organizations, both commercial and governmental, are undergoing rapid change spurred by developments in the computing industry. To stay competitive, these organizations must adopt new technologies, skills, and practices quickly. Yet even for an organization with a well-developed set of software engineering models and processes, transitioning to a new technology can be expensive and risky. Current industry trends are leading away from traditional mainframe environments and toward the workstation-based, open systems world. This paper presents the experiences of software engineers on three recent projects that pioneered open systems development for the National Aeronautics and Space Administration's (NASA's) Flight Dynamics Division of the Goddard Space Flight Center (GSFC).

Introduction

How can an organization effectively accomplish technology transition? Introducing a new technology into an organization requires an investment. But what is the nature and size of that investment, and how long will it be before benefits are realized? How can one quantitatively define these benefits and measure the results? Whatever the ultimate reward of the technology, transition is a step into uncharted waters. Technology infusion requires managers to rethink the way they approach the ordinary project management challenges of developing effort estimates, achieving planned productivity, and dealing with evolving requirements.

The authors of this paper develop software systems under contract to the NASA/GSFC Flight Dynamics Division (FDD). For more than two decades, the FDD has successfully fielded software systems to support NASA spacecraft missions in a relatively stable mainframe/-minicomputer environment. This stability has allowed the FDD to optimize its software development process. During the first half of the 1990s, the authors worked on three projects in the forefront of the FDD's transition from its legacy environment to a workstation-based open systems environment. We discovered that our established development process had to transition as well, in unanticipated ways. Our experiences in this transition and our lessons learned are recorded here with some recommendations for managing technology transitions.

A model commonly used for technology transfer conceives of technology as moving from a producer to a consumer organization. The transition moves through the phases of early experimentation and exploration to technical maturity. The projects discussed in this paper fall primarily within the exploratory phase, where work has progressed from initial experiments to full-scale development, but the technology is still used by a

minority of the organization's staff. Marvin Zelkowitz defined these phases in a paper presented at the 18th Annual Software Engineering Workshop, "Software Engineering Technology Transfer: Understanding the Process."

This paper provides information on the software development organization, then summarizes our observations on each of the case study projects. We then organize the lessons learned and recommend elements of a technology transition plan and ways in which new technology projects might be better managed.

The FDD Software Development Organization

The FDD entered the transition with a mature software development organization that included the Software Engineering Laboratory (SEL), a research and process improvement group whose mature measurement program, cost and schedule estimation models, and management guidelines support software development and technology transfer in this environment.

The FDD had patterned its success on a basic scientific method of gradual, continuous improvement in software engineering technology in a stable computing environment. Controlled innovations were introduced to test new techniques and tools. Studies usually were conducted through pilot projects that applied the new technology under strict controls, with the results evaluated against the organization's norms. The FDD would then incorporate proven beneficial technologies into the standard technology suite.

The FDD had made little investment in exploring open systems technologies. The FDD's few projects outside the mainframe environment were considered out of the organization's mainstream. Developers collected few statistics, and few software engineering experiments were conducted on these projects. When the computing industry began to shift toward workstations, the C language, and open systems concepts, the FDD had little background in these technologies.

Since 1990, the FDD has been moving toward workstation computing platforms and open systems technology, driven primarily by factors external to the development organization. They have done so without the benefit or lead of SEL experiments. Figure 1 illustrates the FDD's

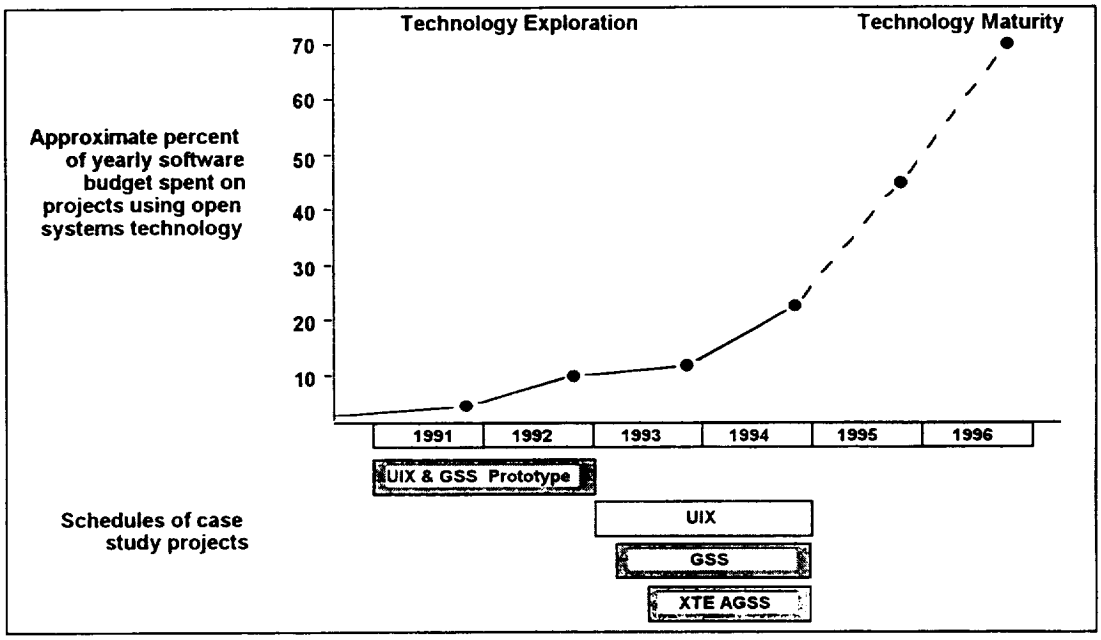


Figure 1. FDD Transition to Open Systems

investment in new technology exploration and the quickening pace of the transition. The case studies discussed in this report are shown at the bottom of the figure in their chronological context.

The Case Study Projects

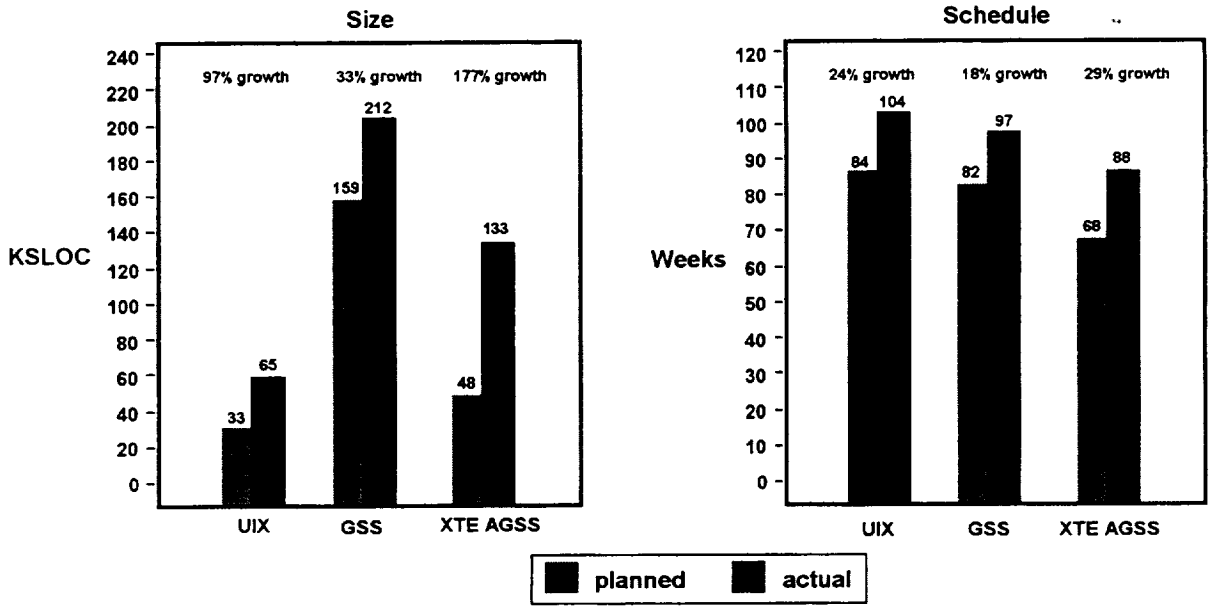
Table 1 provides an overview of the three case study projects, listing the size and language, operational computing environments, and development tools. The projects were planned by tailoring the domain-specific FDD cost and schedule models. The tailoring allowed for some training on specific new technologies. As work progressed, plans were revised to reflect the technology issues. Figure 2 summarizes the development results compared to the plan.

Case Study 1: User Interface Executive (UIX)

The FDD saw a need for a common framework in the new environment. The FDD planned the UIX as a common user interface and executive framework for distributed mission support systems. The decision to base the user interface on X/Motif was primarily driven by industry trends. The aim was to create a configurable system to be used by developers working in Ada, C, or FORTRAN to build application programs that shared a common set of interactive tools. The application developers would not be required to code in X/Motif or to use a GUI builder. The UIX would allow application users to control multiple, distributed processes in a platform-transparent manner. Finally, the FDD required that the UIX support existing hardware

Table 1. Case Study Project Characteristics

Project Descriptors	**Case Study 1: UIX**	**Case Study 2: GSS**	**Case Study 3: XTE AGSS**
Size in KSLOCs and Language	65,000 C	212,000 Ada	66,000 C 58,000 FORTRAN 9,000 User Interface Language (UIL)
Platform and Infrastructure Software	♦ 386 and 486 PCs ♦ Santa Cruz Operation (SCO) UNIX ♦ HP 9000/7xx series workstations ♦ HP/UX ♦ External Data Representation (XDR) ♦ X/Motif (X11R4, later R5)	♦ Digital Equipment Corp. (DEC) VAX 8820 (later Alpha AXP/4000), open VMS ♦ 486 PCs ♦ SCO UNIX ♦ HP 9000/7xx series workstations ♦ HP/ UX 9.0.3 or higher	♦ Hewlett-Packard (HP) 7xx workstations ♦ HP/UX ♦ X-terminal and VT2000 emulation ♦ X/Motif (X11R5)
Development Tools	♦ Intersolv PVCS version control ♦ SCO Open Desktop toolset	♦ DEC Configuration Management System (CMS) ♦ DEC VAXSet Development Toolset ♦ DEC Ada Compiler Version 2.2 ♦ Rational Software Corp. VADSelf Ada for 486 SCO, HP/UX	♦ Builder Xcessory (X Window GUI builder) ♦ HP full-screen editor ♦ HP desktop environment



Compare to size growth in 20% to 40% range and schedule growth in 5% to 35% range on recent maintenance and VAX development projects

Figure 2. Planned Versus Actual Size and Schedule

(the IBM mainframes and Intel-386 PCs) to the maximum extent possible.

Prototyping played a critical role. The ambitious goals of the UIX project were all the more challenging because it was the first to use open systems technology within the FDD. To learn the technology and refine the requirements, the development team built a prototype that covered all major facets of the proposed UIX. Development and evaluation of the prototype ultimately spanned a year and a half. In parallel with the prototype evaluation, the team began specifying the content of the actual UIX. The prototyping experience led to architectural and conceptual changes in the specified product, including abandoning the goal of supporting the IBM mainframe as an application host and deferring implementation of distributed process control until industry capabilities had further evolved.

Lack of a technical infrastructure and an organizational transition plan caused difficulties. Without a preestablished infrastructure ("middleware" such as a network file server), the traditional separation of concerns between the systems support and software development organizations was blurred. It was sometimes unclear whether responsibility for selecting an infrastructure product lay with the project that first needed the capability (in this case, the UIX) or with the support organization that maintained the FDD's institutional hardware resources. Although cross-organizational groups addressed these issues, the lack of an overall transition plan led to misunderstandings and organizational friction.

The FDD's traditional functional requirements and specifications methodology was not sufficient for establishing the infrastructure. Software developers, especially those from mainframe backgrounds, tend to take the existence of a computing system architecture for granted, but this was not the case with the UIX. The developers attempted to define the required software infrastructure using data flow diagrams and functional specifications, the method with

which they were familiar. Unfortunately, their limited knowledge of the technologies involved and the immaturity of available products muddied the development effort. One round of prototyping followed by one round of specification development was not sufficient, nor was the specification formalism conducive to iterative refinement.

Prototyping experience led to technical learning but not better planning. Although the prototyping experience clarified technical issues, it taught the developers little about planning the development project. They believed that the effort saved by rapid prototyping would offset the additional effort needed to come up the learning curve on the new technologies. In the actual project experience, there was still a substantial learning curve in spite of an overlap of development team members with the prototyping team. (For example, the complexity of X/Motif coding was underestimated.) The prototyping team achieved the organization's average productivity based on historical data. However, productivity on the actual UIX development was initially only half that of the prototype project, as the team faced continued technical learning as well as the documentation and inspection demands of a disciplined development methodology. Furthermore, the final system was larger (by a factor of about two) and more complex than indicated by the prototyping.

Case Study 2: Generalized Support Software (GSS)

The GSS project transitioned the post-integration development phase only. The GSS is a multiapplication flight dynamics support class library designed to interface with the UIX. The GSS project was the FDD's first Ada language software development project to make the transition to the open systems workstation environment. Unlike the other two case studies presented in this paper, the GSS was not developed in an open systems environment. The GSS was designed, coded, and integrated in the standard development environment for Ada-based software projects in the FDD, which was a DEC VAX system (later, a DEC Alpha system). The code was then ported to the SCO UNIX environment on PCs for integration with the UIX to create the operational system (an attitude telemetry simulator), with the UIX providing the user interface services. Thus, the technological "leap" taken by GSS was considerably smaller.

The infrastructure needed for a workstation-based development was underestimated. When the GSS project started production in January 1993, the FDD did not have sufficient workstations and associated Ada development tools to support a development the size of the GSS on workstations. The GSS project was not budgeted to procure the workstations and tools needed to develop the system totally in a workstation environment. FDD management decided that the most cost-effective approach would be to develop the GSS software on the institutional Ada development platform, a VAX 8820 minicomputer, until the build integration test phase. At that time, the software for the build would be ported to the workstation environment.

A familiar development environment helped control system growth. The growth in size of the GSS is fairly consistent with FDD projects over the past 5 years. The reasons for the relatively limited growth compared to the other case studies are

- GSS is developed in Ada, a language FDD software developers have been using for almost a decade.
- The developers were familiar with the GSS development environment and toolset, and only the latter phases of the life cycle (build integration through independent test) were performed on the workstation platforms.

The GSS project comprised pure computational applications software, not interactive software. The GSS project did not have to deal with user-system interface issues in the new open systems environment. Because the UIX system provides the GUI for GSS-based flight dynamics applications, the GSS project was "shielded"

from many of the technological hurdles and learning curve relating to building GUIs on workstation platforms. This experience suggests that scientific application development is less affected when moving to open systems platforms than is user interface software development.

Case Study 3: X-Ray Timing Explorer Attitude Ground Support System *(XTE AGSS)*

The FDD faced a new requirement to deliver software on workstations. On this project the FDD developed mission attitude ground support applications in an open systems workstation environment. The FDD had developed these types of applications before but only in an IBM mainframe environment. The FDD was required to deliver the applications to a separate GSFC organization, the Mission Operations Division (MOD), for integration into their operational system. Such applications had previously been installed and operated only within the FDD environment. The MOD systems use a locally developed package called Transportable Payload Operations Control Center (TPOCC) to provide the client-server framework.

Project planning was largely based on experience in the legacy environment. The project planners estimated size (in lines of code) of the applications based on previous FDD systems. The planners determined they could reuse a large amount of FORTRAN computational code being developed concurrently on the mainframe. Since XTE AGSS was a first-of-a-kind project, the planners lacked good comparisons to help estimate how the use of TPOCC and X/Motif graphics would affect the size. A productivity rate 20 percent lower than the FDD norm was used to account for the new technology learning curve.

The XTE development effort was significantly underestimated. As it turned out, the size of the applications was underestimated by a factor of three, primarily because

- Planners underestimated the size of the TPOCC and graphics-related code.
- Reused code was larger than expected.
- Requirement changes added major new functionality.

Productivity on the initial builds was considerably lower than expected. The main causes of the lowered productivity were underestimation of the complexity of the new technology, the lack of X/Motif expertise on the team, and skill mix problems. Productivity increased in the later builds as the team became more experienced with the technology and as the skill mix improved; some builds met or exceeded the FDD norm.

The traditional methodology had to change to incorporate iteration. Only about half the unit designs had been completed by the time of critical design review. (FDD methodology called for all unit designs to be complete at that point.) This indicated trouble, but the developers and their management did not realize the full extent of the effort underestimation until the coding phase. Then it became clear that they could not complete the project according to the original plans, and they had to renegotiate the delivery schedule and add staff. The new schedule was still highly compressed because of XTE mission deadlines, forcing the developers into an iterative approach of designing and coding build by build. For the most part the iterative approach worked well, though it made assessing progress difficult.

Requirements instability exacerbated problems. It is common in FDD development projects that software requirements evolve during the course of development. The XTE project encountered challenging, though not unprecedented, requirements instability, partly because the FDD analysts thought of ways to make the software more generic well after design and implementation were underway. System specifications were changed on several occasions to serve the best long-term interests of the FDD. The resulting perturbations were far more severe than they normally would have been because the project was in technology transition.

The development team needed immersion in the technology to come up to speed. One of the major challenges of the project was learning the TPOCC system. This amounted to technology transfer from the MOD to the FDD. The TPOCC system is large and complicated, and the XTE development team could find no single person who was expert in all aspects of the system. Early in the implementation phase, part of the development team relocated to the MOD development facility for 2 months. The relocation was very useful for promoting communications, though interaction was limited because the MOD developers were busy with their own projects. The early builds implemented the TPOCC interfaces and were kept relatively small to allow quick feedback. To get a testable framework in place, the team split the first build in two when it turned out to be far larger than planned.

Unrecognized technological assumptions created transition problems. The biggest problem encountered with TPOCC was not in implementing the application interfaces, but in installing TPOCC in the FDD. Differences between the MOD and the FDD computer environments and system administration approaches became evident. For instance, the FDD used network user accounts, with which TPOCC was not compatible. Other problems developed when the MOD moved to new releases of the HP operating system and Motif before these versions were available to the FDD. In retrospect, the memoranda of understanding between the FDD and the MOD, which only addressed XTE AGSS release dates, should have also specified TPOCC version delivery dates, versions of system and support software to be used, and all applicable standards.

Increasing personal interaction and emphasizing skill mix helped alleviate problems. After the FDD tested the releases in-house, the plan called for delivering them to the MOD for integration into the operational environment. Because of all the unexpected problems encountered thus far in the project, the FDD development team decided to work with the MOD developers informally to integrate the system before formal delivery. The main problems found during informal integration and testing were with installation instructions, not with the software itself.

A final factor very important to the success of the XTE AGSS was staffing. Once the true magnitude of the development effort was understood, project management committed highly experienced and motivated individuals to the team. They provided a good skill mix that included both software development and application domain knowledge and C and FORTRAN experience. In spite of the pressures, this commitment led to a very good team spirit and a successful product.

Lessons Learned

The complexity of open systems was much deeper than anticipated in all three case study projects. The developers learned that "industry standards" are often evolving or competing conventions, that COTS products are marketed before they are mature, and that interoperability does not always live up to advertised expectations. They discovered how much middleware it really takes to make a distributed system work. The organization realized how significant the choice of hardware is to the viability of the final system, how much hardware is needed to fully support a distributed development effort, and that the costs for support software and development environments can rival or exceed the cost of the hardware. They also had to find ways to overcome compartmentalization of open systems knowledge in their own and in interfacing organizations. We have grouped these lessons around organizational, technological, and managerial themes.

Organizational Lessons

Organizational transition plan. A planned transition for the entire organization, backed by management commitment, is needed. The case studies indicate that the FDD approached the transition on a project-by-project basis, not only

reducing coordination but also slowing the dispersion of knowledge. Management did attempt to coordinate activities at the top levels of the organization, but the staff on the individual projects received little information as to how their project fit into the plan. As a consequence, people focused almost exclusively on the challenges of using the new technology on their own projects, with little incentive to share their experiences with others in the organization.

Changing organizational roles. Changing technology can blur traditional roles, garble communications, and cause friction. No doubt this is part of what makes transition plans hard to create in the first place. Effects of technology change can ripple across organizations in ways they cannot readily accommodate. The leaders of the organization must define and communicate a vision for doing business using the new technology and help the staff make organizational changes stemming from it. Changing technology does not necessarily mean business reengineering, but if the organization is making a major technology change it should carefully evaluate the impact on its business model as well.

Outreach across organizational boundaries. Sharing experiences across project and department boundaries is critical during technology transition. "Department" here means any portion of the organization that traditionally practices "information hiding" from other portions. The case studies show that information barriers can exist even at the lowest levels. Groups of 5 or 10 people down the hall from each other may not share information even though they are engaged in parallel transitions. This may seem counterintuitive to anyone who has experienced the "office grapevine," but people do not grasp organizational plans through the grapevine. Personal contact works well for transferring detailed knowledge when people have a focus and goals, but it takes a special effort to find that focus. Management must provide forums, whether formal or informal, for sharing new technology experiences in real time without "turf" issues interfering.

Disseminating lessons learned. The FDD has a tradition of writing good history documents after each project to capture lessons learned, but often they come out too late to help the project planners who really need them. Also, if a procedure for using them is not integral to the development methodology, the lessons may sit on the shelf unheeded. An organization should document lessons learned at points in the development process well before the project's end and should make producing and using them part of the development procedure. The lessons should be disseminated in a way that will make them easy to access (for example, in a cross-indexed on-line library). The goal should be to coalesce the lessons into an institutional knowledge base.

Technological Lessons

Cultivating market awareness. The competitive marketplace drives the evolution of open technologies, so using them effectively requires cultivating and maintaining market awareness. An organization coming from a stable mainframe environment that does not emphasize compatibility with the world beyond the vendor may be a "closed shop," especially if that organization produces a very specialized product (such as space ground support systems). The case studies suggest that the FDD was not prepared to deal with rapid market evolution. In the past, the organization usually had time to choose technologies carefully and experiment with "seed" projects. This approach was not geared to the pace of change the developers had to adopt to accomplish the transition to open systems. The transition forced a cultivation of market awareness, which in turn requires applying the discipline and resources to track all aspects of industry evolution. Management must actively encourage technical staff to follow market trends and pursue continuing education.

Training for front-line workers. Beware of unrealistic optimism on the part of both managers and technical staff regarding the ease with which staff can master the new technologies. The case studies revealed that people had a tendency to think in terms of distinct skills to be

learned, new, but similar to existing skills. In reality, the myriad interrelationships of a new suite of technologies, and the industry context in which they are evolving, are very complex. Our experience was that the amount of ramp-up time needed to learn new technologies, from least to most, was for UNIX, C, networking, and X/Motif (most difficult to acquire even using a GUI builder). When most of the team has to learn all the technologies together, the time invested is significant.

Technical compatibility. When a software development shop first adopts open systems technology, it may expect to easily interface with open systems in client and peer organizations. This expectation was not realized in the case study projects; "plug and play" is not yet the norm. Incompatibilities result if the organization does not have detailed knowledge of the technologies used by the interfacing organization. Open systems invite cooperation but do not guarantee compatibility. Interacting organizations should discuss and document their agreements on issues such as standards, COTS product versions, and configuration management assumptions.

Retooling the infrastructure. Organizations such as the FDD with long-standing stable computing environments have usually developed customized software development toolsets and a supporting infrastructure. When moving to a new technology, problems that were previously solved in the legacy environment may need to be solved again because the infrastructure and tools have changed. Even a technically mature organization may be unprepared for the extent to which it must develop new approaches to basic software engineering problems that it thought it had solved long ago. A mature organization may be at a disadvantage because of a high comfort level with its proven techniques.

System engineering. In all three cases studied, the transition to open systems caused the developers to shift from a purely software engineering viewpoint to more of a system engineering perspective. In the absence of a stable technical infrastructure, the developers had to devote considerable time and effort to understanding engineering topics for which their previous project experiences had not prepared them. Both hardware and software components had to be treated more or less equally. Emphasis shifted from crafting systems from lines of code to selecting and integrating the right combination of hardware and software components. When no established computing infrastructure exists, developers must perform systems engineering analysis at the start of the project to plan for and procure sufficient resources.

Project Management Lessons

Realistic expectations. Project managers cannot expect to achieve all the goals during a technology transition that the organization achieved in the stable technology. Aiming for these goals can lead to over-commitments and compromise the success of the transition. The project manager must be strategically aggressive but tactically conservative, and careful when making commitments.

Accurate effort estimation. Technology transition requires investment. The SEL *Manager's Handbook*, source of the FDD's project estimate models, recommends applying an additional effort multiplier of 2.3 when a project type and the technical environment are new to the organization. Had the case study projects followed this guidance, the UIX and XTE AGSS projects would have started with much more realistic effort estimates. The GSS project, which did not involve the same degree of transition as the other two, came closer to the standard model, and the effort multiplier may not have applied to it.

Staffing and skill mix. The manager in the legacy environment faces a particularly difficult staffing and training issue. The case study projects used "hot" technology, but because the FDD's existing technology was mainframe based, it did not tend to attract and retain people with expertise in new technologies. Those recruits who did have open systems experience generally were not experienced in either

application development or in the FDD's legacy systems and problem domain.

Training for technical managers. One problem with this technology transition was that the technical managers and senior technical people were reared in an older technology. The case studies show a tacit assumption that project managers would somehow "pick up" the open systems concepts sufficiently to competently plan and manage these projects. In fact, when project planners lack an understanding of the technology their team is using, they may not understand the real issues and cannot make good planning decisions. Open systems approaches bring significantly different problem-solving tools and techniques. Technical managers need training and hands-on experience. They need to know what they are up against when setting schedules and budgets.

Role of prototyping. Although useful for avoiding disaster, prototyping is not in itself a sufficient basis for project planning. A prototype does not confer organizational learning. Even a second-time use of a technology may not uncover all the possible pitfalls. Organizations have to assimilate information until they reach the point of "intuition."

Methodology. Methodology requirements oriented toward the routine design problem may actually impede learning, because they assume the problem-solving technology is already well understood. For example, the requirement that all unit designs be completed before any units are coded makes it impossible to feed lessons learned about the new environment into the design process. Although progress is harder to measure, iteration promotes learning the new environment. When introducing new technologies, a more appropriate approach may be to develop the system framework first and the application functionality later. The project can then be broken into numerous small builds and progress and expended effort assessed after each build. The development plan should be readjusted accordingly. To gain integration experience in the new environment, functionality should be slipped from early to later builds rather than delaying delivery of early builds.

Software metrics. Metrics are critical to understanding the new technology. However, measurement programs established for the old technology may not be adequate for the new. Predictors based on source lines of code may not be meaningful when using GUI builders, code generators, and COTS packages.

Conclusions and Recommendations

A technology transition plan. While it is not our purpose to develop a model for technology transition planning, our observations do suggest issues that a transition plan should address. Table 2 presents our suggestions from the perspective of a fairly large organization with a mature and stable, but dated, technology infrastructure. (The ordering of topics does not imply a procedural sequence.)

Climbing the hills of technology infusion. Adopting a new technology is like climbing a hill representing the cost of the transition. Few computing professionals and managers are expert at estimating the height of the hill and the rate of progress over it. Yet as Figure 3 shows, the increasing pace of change brings whole ranges of hills to climb. The FDD, having successfully applied the SEL process improvement concepts in a stable environment, was unprepared for the rapid pace of the transition to open systems. Perhaps the FDD, with its stable environment and funding, had become accustomed to investigating technologies at its own pace. In the current technological environment, however, we may not have the luxury to control which technology hills we will climb or when.

Can we learn to adopt technologies faster and more efficiently? A common element in these case studies is a failure to realize that technology transition alters the essence of the design problem. The literature on the design process distinguishes between *routine* design and *variant*, or

innovative, design. In routine design, both the problem domain and the problem-solving process are well understood, and the main issue is accommodating an established solution to project-specific needs. But in variant design, while the problem domain may still be well understood, the problem-solving process is not. Approaching the variant design problem as if it were just a more difficult instance of the routine problem, to which slightly adjusted models and procedures can be applied, leads to problems.

Improving management models for "emerging technology" projects. Variant design problems can be expected whenever new technologies are adopted. The software industry needs to systematize its knowledge of them. Project planners must understand when the organization is going through a transition that fundamentally changes the problem-solving process so they can approach it the right way. Of course, the problem is compounded by the fact that technology drives organizational structures; as industry

Table 2. Recommended Content of Transition Plan

Topic	*Comments*
Vision	What is the purpose of adopting the new technology? Will it support and improve the organization's business position? Examine assumptions regarding these issues to reveal aspects of the transition that otherwise might go unnoticed.
Industry assessment	How mature is the technology? Look at the hardware/software solutions being adopted by other organizations. Attend trade shows and conferences. Challenge the assumption that your organization is unique in its needs or functions. Be proactive in defining business direction in terms of new technologies.
Organizational assessment	Consider the impact of technology on the whole organization, not just your department. How will the technology change the roles of supporting organizations? How will the organization operate after adopting the technology?
Hardware/ software tradeoffs	Challenge the assumption that existing equipment must be retained for cost-effectiveness. The cost of software development and development environments may outweigh equipment cost.
Standards, conventions	Make sure all the required standards are identified and followed. Understand the difference between established standards and industry conventions.
Pilot projects	Define realistic goals for pilot projects; avoid developing products best left to industry (such as distributed operating systems). Concentrate on using new technology to bolster the organization's traditional strengths. Keep initial transition projects small.
Staffing	Staff transition tasks carefully. Experienced, motivated, capable people are essential to the transition project. Articulate the special skill needs to management, back the request with evidence, and be aggressive about resolving staffing problems and retaining team members.
Personal contact	Expedite personal contact across department boundaries. Establish mechanisms such as cross-department working groups, but avoid too much structure. Allow teams flexibility to discover what areas need focus and how to work together.
Training plan	Ensure that training is available; err on the side of too much. Project planners need exposure to the new technology at the time of planning; developers need it when assigned to the project. Support staff also need training.
Methodology	An iterative approach promotes learning. Use numerous small builds to gain integration experience in the new environment. Slip functionality rather than delay delivery. Challenge methodology requirements focused on the routine design problem.
Metrics	Challenge the assumptions of measurement programs established for the routine design problem. The new technology may demand different metrics.

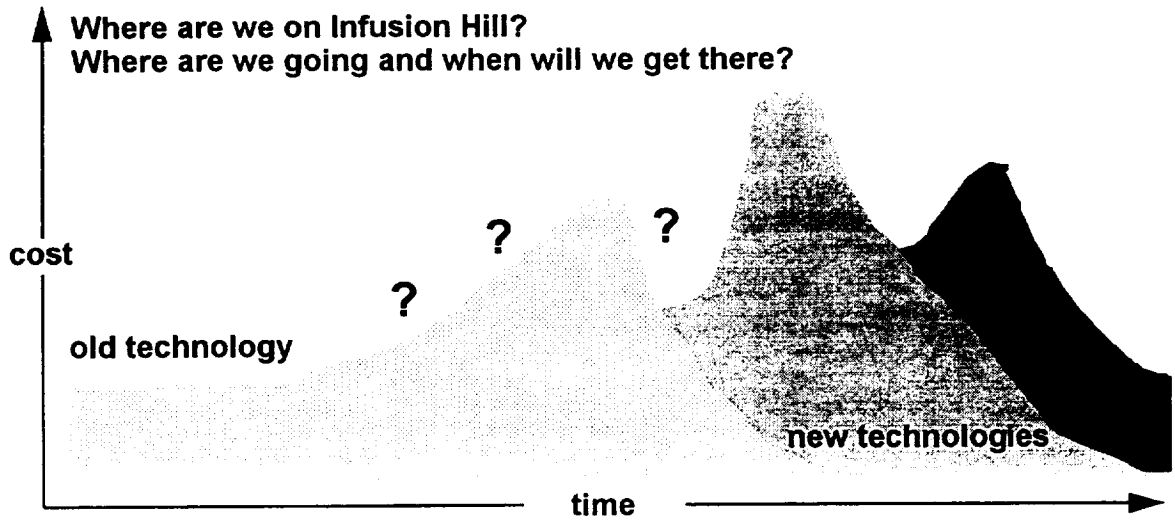


Figure 3. The Hills of Technology Infusion

retools, organizations discover possibilities that prompt them to reexamine their missions. Although this exploration can be guided only in broad outline, the need to steer projects through these uncharted waters remains.

New Directions for the FDD

Despite transition problems, the software developed by the projects we studied appears to be of good technical quality. The XTE systems were proved reliable in testing and are being reused by other projects for upcoming missions. New projects are using the UIX and the GSS in their designs. The FDD itself is embarking on full-scale conversion to a distributed system, porting or replacing up to 6 million lines of legacy software. A stable infrastructure for open systems is beginning to evolve within the FDD, improving prospects for success.

Moving a large organization from a mainframe legacy to a new environment of open systems is a complex technology transition problem. The transition involves much more than a simple switch of tools and techniques. Transitions that cause sudden shifts from routine to variant design problems are likely to become more common in the future. Our challenge is to apply organizational learning techniques in staying abreast of industry developments, and to effectively incorporate them in our experience base.

Acknowledgments

The authors gratefully acknowledge the help of Marvin Zelkowitz of the University of Maryland and Myrna Regardie of CSC in clarifying our concepts and consulting on the presentation. The original inspiration for this report was Zelkowitz's paper on the technology transfer process.

References

Landis, L., S. Waligora, F. E. McGarry, *Recommended Approach to Software Development (Revision 3)*, Software Engineering Laboratory, SEL-81-305, June 1992

Landis, L., F. E. McGarry, S. Waligora, et al., *Manager's Handbook for Software Development (Revision 1)*, Software Engineering Laboratory, SEL-84-101, November 1990

Zelkowitz, M. V., "Software Engineering Technology Transfer: Understanding the Process," *Proceedings of the Eighteenth Annual Software Engineering Workshop*, Software Engineering Laboratory, SEL-93-003, December 1993

Lessons Learned in Transitioning to an Open Systems Environment

Dillard Boland
Dave Green
Warren Steger

10022890 1

Purpose and Method

- Problem: Transition to a new technology requires investment before benefits are realized — how can we plan and manage efficient transitions in the midst of rapid industry evolution?
- Method: Study three projects in the GSFC Flight Dynamics Division (FDD) moving from a mainframe environment to "open systems" workstation technology
- Goal: Improve our understanding of technology transition and identify lessons learned

10022890 2

Background: The FDD Software Development Organization

- Through the SEL process, the FDD has achieved a track record of continuous improvement in reuse, error rates, and other software characteristics
- Stable development environment: IBM mainframe with FORTRAN, and DEC VAX with Ada
- Focused SEL experiments: OO, Ada, Cleanroom, IV&V, resources usage
- Computing environment held relatively constant while process and products evolved

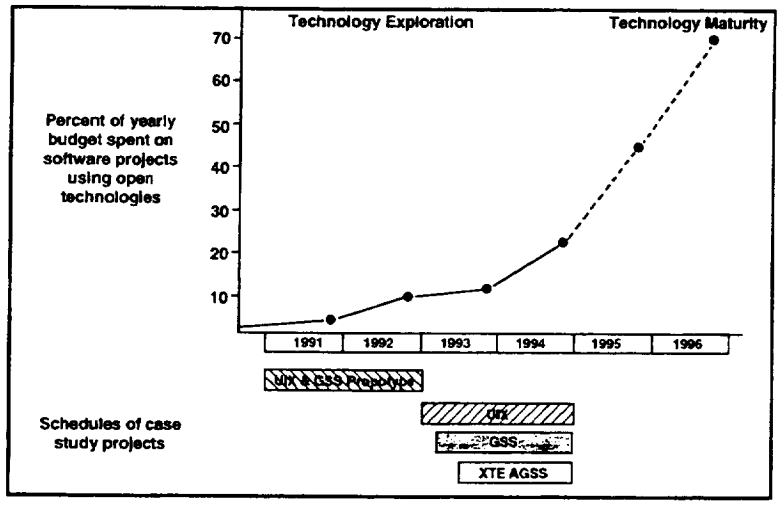
10022890 3

Background: Transition of the FDD to Open Systems

- FDD/SEL achievements were within the context of stable mainframe and VAX computing environments
- Now FDD is moving toward open systems
  - Workstation computing platforms, industry standards, and conventions
  - Use widely available COTS products; emphasize portability and interoperability
  - Goals are economic and technical: less vendor dominance, more competing solutions, "more bang for the buck"
- How will this dramatic change in computing environment affect our products and processes?

10022890 4

Background: Transition of the FDD to Open Systems



CSC

10022890 5

The Case Study Projects

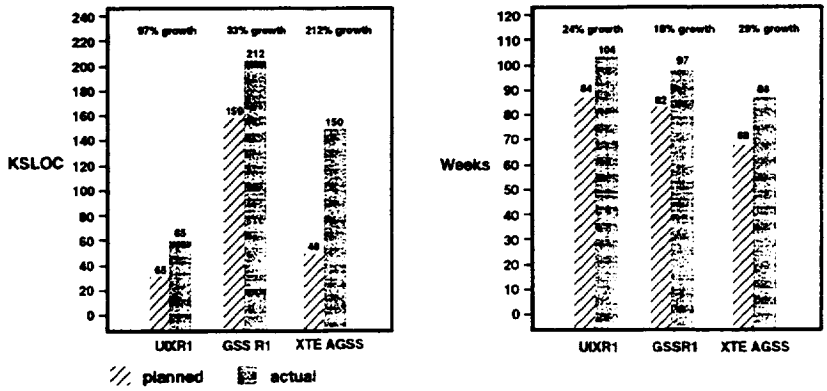
Project	Platform	Size (SLOC)	Purpose
UIX	PC (SCO UNIX), HP	65,000 C	Multiapplication user interface system
GSS	DEC Alpha, HP	212,000 Ada	Multiapplication attitude support components
XTE AGSS (two subsystems)	HP	150,000 C and FORTRAN	Mission attitude ground support applications

Planned using SEL models based on local mainframe and VAX experience with adjustments for new technology

CSC

10022890 6

Case Study Projects: Comparision of Results



Compare to size growth in 20% to 40% range and schedule growth in 5% to 35% range on recent maintenance and VAX development projects

10022690 7

Case Study 1 — UIX

- We wanted to develop a common user interface and executive framework for interactive, distributed mission support systems
- We did the logical thing: up-front prototyping
  - Led to necessary architectural and conceptual changes
  - Not a good basis for project planning: final system is much larger and more complex than prototype indicated
- Lack of a preestablished system architecure ("middleware") proved to be a significant technical and organizational stumbling block
- The project was refocused on the user interface and extended: wait for industry middleware to evolve before attacking distributed executive

10022690 8

Case Study 2 — GSS

- We wanted a class library of flight dynamics capabilities from which we could build our systems; we prototyped it along with the UIX
- We wanted to transition Ada development to workstation environment, but have not been able to except for integration and test phases
- We discovered that matching development toolset capabilities available on DEC/Alpha/Open VMS is not yet cost effective on our target platforms
- Current plan is to phase in development tools as market forces drive the costs of Ada development systems down (this is already happening)

CSC

10022690 9

Case Study 3 — XTE AGSS

- We needed to integrate with client/server software developed by another group at GSFC, and to provide our first interactive X/Motif system for mission support (UIX was not ready)
- We assumed we could achieve our current norms: compressed development schedules and reusable software
- We severely underestimated the complexity and functionality required to meet these goals in a new environment
- We underestimated the difficulties of interfacing with other group's software (same "open" technologies, but environment differences such as COTS products at different version levels)
- Technology transfer facilitated by relocating developers to the other organization's site to infuse their technology, and by adopting highly iterative implementation approach

CSC

10022690 10

Lessons Learned: Organizational

- A coordinated organizational transition plan, with management commitment, is essential
- Changing technology can blur traditional roles, garble communications, and cause friction, because the "old ways" do not always adapt well to new technology
- The organization must find ways to cooperate and share lessons learned across departmental boundaries; technology transition is not the time for information hiding!

Lessons Learned: Technological

- Open systems and rapid industry change demand we cultivate market awareness to replace our "closed shop" outlook
- Open systems invite cooperation but do not ensure compatibility: stress coordination and communication
- Early training is important for both the technical managers and the frontline workers
- Problems previously solved in legacy environment (e.g., CM, reuse) often must be solved again in the new environment

Lessons Learned: Project Management

- Open systems require open minds: awareness of market trends, continuous organizational learning, structured feedback of lessons learned
- Use prototyping to avoid disaster but not as a basis for project planning
- Don't expect to achieve the goals of a technologically mature organization while you are transitioning
- We need a better management model for "emerging technology" projects

10022690 13

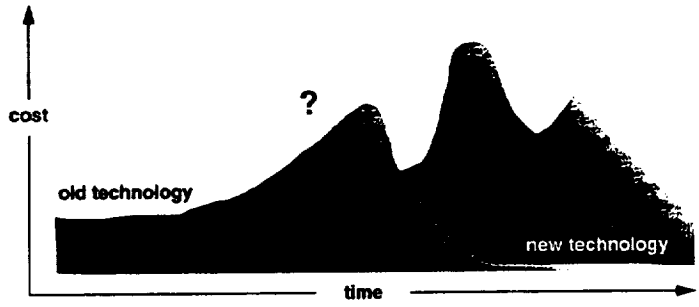
Conclusions and Recommendations

- We need scientific data about technology transitions: The industry needs to honestly appraise successes and failures and learn from them
- Our existing SEL models are not adequate for technology transitions: Upgrade them
- Open systems concepts and decreasing hardware costs force a systems (not just software) engineering approach
- Personal contact is the most effective means of information sharing on technology transition – need an institutional mechanism
- We must plan for continuously infusing technology and commit resources to that end

10022690 14

The Hills of Technology Infusion

- In a rapidly evolving industry and an open marketplace, we must learn better skills for evaluating and adopting new technologies



Where are we on Infusion Hill?
Where are we going and when will we get there?

CSC

New Directions for the FDD

- XTE AGSS subsystems are being reused for upcoming missions
- EOSTGSS project just completed PDR
  - Up-front emphasis on system engineering
  - Using UIX as part of infrastructure
- Flight Dynamics Distributed System
  - Port or replace the 6 million SLOC of our mainframe and VAX legacy
  - Will use GSS and UIX
  - Infrastructure is now coming into place

CSC

Lessons Learned in Deploying Software Estimation Technology and Tools

Nikki Panlilio-Yap and Danny Ho
IBM Canada Ltd.

Abstract

Developing a software product involves estimating various project parameters. This is typically done in the planning stages of the project when there is much uncertainty and very little information. Coming up with accurate estimates of effort, cost, schedule, and reliability is a critical problem faced by all software project managers. The use of estimation models and commercially available tools in conjunction with the best bottom-up estimates of software-development experts enhances the ability of a product development group to derive reasonable estimates of important project parameters.

This paper describes the experience of the IBM* Software Solutions (SWS) Toronto Laboratory in selecting software estimation models and tools and deploying their use to the laboratory's product development groups. It introduces the SLIM* and COSTAR* products, the software estimation tools selected for deployment to the product areas, and discusses the rationale for their selection. The paper also describes the mechanisms used for technology injection and tool deployment, and concludes with a discussion of important lessons learned in the technology and tool insertion process.

1.0 Introduction

Developing a software product involves estimating project parameters such as effort, cost, duration, and reliability. Estimates are crucial to developing the project schedule and allocating the necessary staff and resources. Estimating is typically done in the planning stages of the project when there is much uncertainty and very little information. Nonetheless, estimation is very important to software development since it forms the basis for project planning and management. It is a cross life-cycle discipline that applies to all phases of the development life cycle. During the course of running the project, constant re-estimation is vital to assess the risks at various stages of the project. In some situ-

---

*

- ***IBM***, ***AS/400***, ***OS/2***, ***AIX*** and ***BookManager*** are registered trademarks of International Business Machines Corporation.
- ***SLIM*** is a registered trademark of Quantitative Software Management, Inc.
- ***COSTAR*** is a trademark of Softstar Systems.

Nikki Panlilio-Yap is on a leave of absence from IBM Canada Ltd. and can be reached at Loral Federal Systems, 6600 Rockledge Drive, Bethesda, Maryland 20817, U.S.A. Her e-mail address is nikki@lfs.loral.com on Internet.
Danny Ho can be reached at IBM Canada Ltd., 844 Don Mills Road, North York, Ontario M3C 1V7, Canada. His e-mail address is danho@torolab2.vnet.ibm.com on Internet.

ations, the estimates have to be revised and the project has to be rescheduled.

This paper captures the experience of the IBM SWS Toronto Laboratory in deploying software estimation technology and tools, and summarizes the key lessons learned.

2.0 Estimation Technology and Tools Deployment

The deployment of software estimation technology and tools in the IBM SWS Toronto Laboratory [10] consisted of three major stages as illustrated in Figure 1. Activities associated with each stage are shown; each stage is described in the following subsections.

2.1 Understanding - The Early Stage

The Software Engineering Institute (SEI) self-assessment conducted by the IBM SWS Toronto Laboratory in 1991 revealed a critical need for software estimation techniques and tools. Probably the best tools for estimation are those that use models based on historical data from one's own organization or environment [1, 4]. In the absence of an internally developed tool based on historical data from the IBM SWS Toronto Laboratory or from similar IBM laboratories that develop multiple software products across multiple hardware platforms, it is logical and practical to use one or more commercially available estimation tools. Some of these tools have underlying models based on thousands of software development projects from industry. These tools typically use input on the size of the product to be developed, project constraints, characteristics of the development team, complexity of the product, and characteristics of the development environment.

The Tool Evaluation and Introduction Process described in Ho [7] was adopted in conducting pilots and early experiments. Once several promising tools and vendors had been selected, the vendors were requested to send detailed information or demonstration diskettes of the tools for evaluation. Pilot experiments with some software-development projects were also conducted by obtaining trial licenses or borrowing tools available at other IBM Canada Ltd. sites.

2.1.1 Criteria Used in Tool Selection

Several criteria were used to evaluate software estimation tools. Required basic features include the ability to:

- Give accurate estimates
- Perform automatic recalculation whenever some parameters are altered
- Break down the estimates into different phases of the development life-cycle
- Support different software sizing methods.

Some desirable and advanced features are the ability to:

- Track project actual data
- Conduct re-estimation if needed
- Perform what-if analysis to experiment with different parameters
- Be extensible to include user-specific parameters
- Be adaptable to user-specific development environments.

2.2 Installation - Making the Selected Models and Tools Available

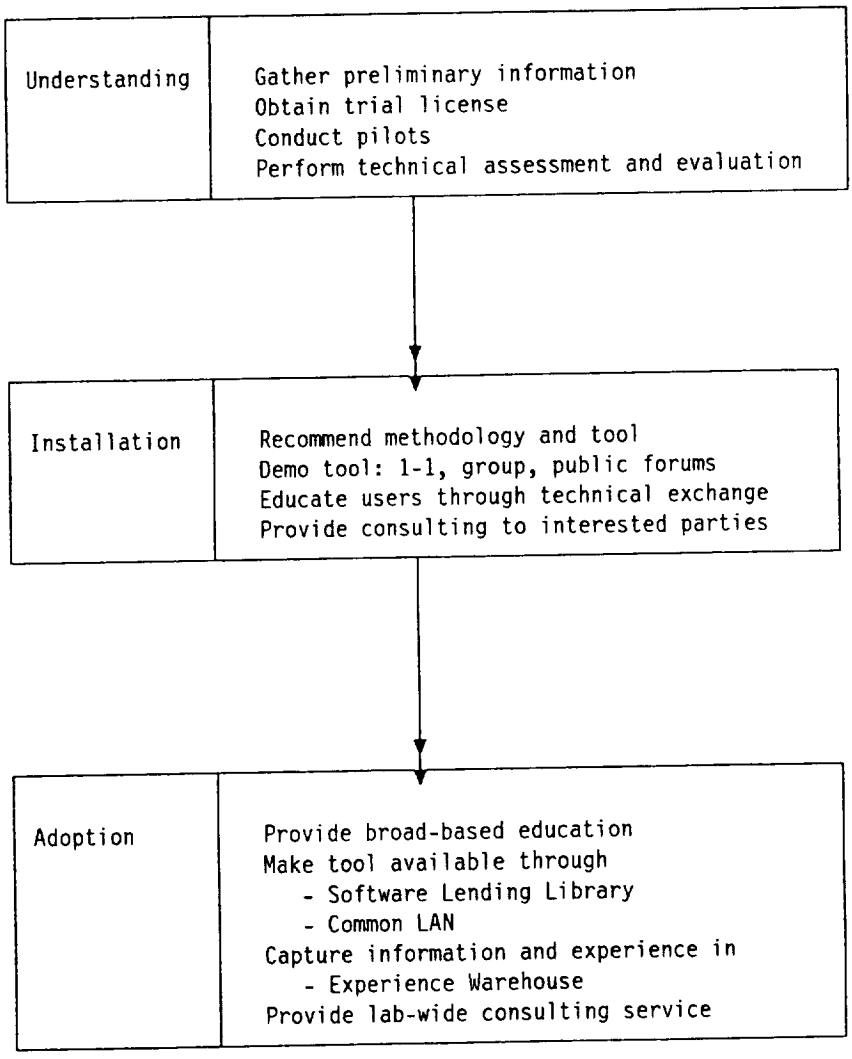


Figure 1. Stages of Software Estimation Technology and Tools Deployment

2.2.1 The Selected Models and Tools

The final decision in the choice of software estimation techniques and tools depended on the results of the pilot experiments. Both the SLIM and COSTAR products satisfied the basic requirements and possessed some desirable features for good software estimation tools. Both tools produced good pilot results.

The amount and complexity of input required for these tools is not nearly as cumbersome as that required for some other commercially available tools. In addition, the underlying theory of the SLIM and

COCOMO models is well-known and published in the public domain. Two models were adopted because neither one gave 100% accurate estimates. The use of more than one model may make up for some of the shortcomings of each one.

2.2.1.1 The COCOMO Model and the COSTAR Tool: The COnstructive COst MOdel [5] is a mathematical model that estimates the duration, staffing level, and cost of software projects. The model makes use of the effort equation as its fundamental calculation, using lines-of-code (LOC) as its fundamental input.

$$Effort = K_1 \times KDSI^{K_2}$$

where

Effort is in staff-months.
K_1 and K_2 are constants whose values are dependent upon the mode of development.
KDSI is kilo-delivered source instructions.

The effort equation is refined by multipliers from product, computer, personnel, and project parameters. The calculated effort also forms the basis for estimating the project duration and staffing.

The COSTAR tool, a DOS-based estimation product by Softstar Systems [8, 15], implements COCOMO. COSTAR 3.0 is currently deployed in the laboratory. Estimates are provided for the intermediate and detailed models, and estimation can be performed in a structured manner using subcomponents. The output consists of a development summary and a variety of reports.

2.2.1.2 The SLIM Model and Tool: The Software LIfe-cycle Model (SLIM) is a metrics-based estimation model developed by Putnam [11, 12], using validated data from over 3000 projects from industry. The projects are stratified into nine application categories ranging from microcode to business systems. The category into which most of the IBM SWS Toronto Laboratory products fall is system software.

The following gives the key equation for the SLIM model.

$$ESLOC = PP \times (\frac{PY}{b})^{\frac{1}{3}} \times Y^{\frac{4}{3}}$$

where

ESLOC is executable source lines-of-code.
PY is effort in person-years.
Y is duration in years.
b is a special skills factor that is a function of system size.
PP is a productivity parameter that translates into the productivity index (PI).

The formula is used to establish a cost-and-time schedule for development of a system of certain size. The productivity parameter can be baselined through historical project data and mapped through a translation table to the productivity index (PI). PI is a macro measure of the total development environment. It possesses different averages and deviations for different application categories.

The SLIM tool is a software product that embodies the SLIM model. It was developed by Quantitative Software Management, Inc. (QSM). The tool can be customized to a specific organization through calibration using historical data. It automates the calculation of the optimum solution based on project assumptions and constraints. It also

has a rich set of what-if capabilities for the assessment of time, effort, and cost risks. A more detailed description of the tool and its capabilities can be found in [6], [9], [13], and [14].

2.2.2 Demonstrations and Technical Exchange Sessions

During the course of injecting software estimation techniques and tools, the SLIM and COSTAR products were demonstrated on different occasions:

- To individuals in one-on-one sessions
- To development teams in group sessions
- In public forums such as conferences and tools expositions.

2.2.3 Limited Consulting

In addition to the demonstrations and technical exchange sessions, in-depth consulting was offered to a number of projects whose personnel showed commitment to learning and using the selected software estimation techniques and tools. We sat down with project managers, planners, and other key project personnel, and walked them through the software estimation process with the aid of the selected tools. We also provided analysis and interpretation of the estimates and tips on their use.

2.3 Adoption - Expanding the User Base

2.3.1 Broad-Based Education

To increase the penetration of software estimation techniques and tools within the laboratory, we developed a two-day course. Its objectives were to:

- Teach the underlying theories of the SLIM and COCOMO models
- Provide in-depth training on the SLIM and COSTAR tools
- Provide hands-on experimentation with the tools.

2.3.2 Tool Availability

One of the most important tasks in deploying promising tools is to make them available throughout the laboratory. The target users for the SLIM and COSTAR tools are primarily planners, project managers, and team leaders.

Since the majority of the laboratory community is LAN-connected, the Toronto Lab Common LAN [7] is used to make the tools generally available. The Common LAN is basically a collection of OS/2* file servers, AIX* file servers, and end-user OS/2 and AIX workstations, connected by multiple token rings. A license control mechanism limits the number of users concurrently accessing the tools to the maximum license count. The mechanism also provides a means to electronically invoke the tools in a more automated manner, as opposed to traditional manual software distribution.

The Software Lending Library is a central location used to distribute the tools to non-LAN users. A user who signs out a software package is given two weeks to experiment with the software. When the software is returned to the library, an online survey is sent to the user to gather feedback on the tool.

2.3.3 Information Availability

Availability of tools must be accompanied by availability of tool information and ease of access to the information. Tool information is accessible from:

- The Laboratory Experience Warehouse (EW) — the Toronto Laboratory's version of an Experience Base used to

store some forms of packaged experience as described in the Experience Factory concept proposed by Basili and his colleagues [2, 3]. It is a central repository for a wealth of information useful to the software development community. Its tool section consists of four matrices collecting information on tools under evaluation, under pilot (unsupported), supported (by the Tools Support Group), and rejected (not promoted). The tools within each matrix are grouped by development life-cycle, and the tool documents can be accessed through BookManager* hypertext links. The information includes some general description of the tool, formal evaluation report, and user feedback.

- The Window on the World (WOW) utility is an online utility to retrieve information for quick reference. Information for supported tools is kept on WOW. This includes general description, operation, licensing constraints, installation, environment constraints, invocation mechanism, support, and license agreement.
- The Software Lending Library was described in the previous section. Available information includes tool description, user feedback, mechanism for requesting the tool center of competence to contact the user and provide consulting, and manuals of the tools accessible through the Common LAN.

2.3.4 Lab-Wide Consulting

As more and more project groups demonstrated a need, we made software estimation consulting services available to the laboratory's development community. Because of resource constraints at the laboratory level, most consulting was provided to the project groups through project personnel who had been trained on the use of the software estimation techniques and tools. This allowed the project groups to develop their own local experts. It also allowed us to provide service to more development project groups.

2.4 Level of Deployment

Five sessions of the Software Estimation and Tools course have been offered to the laboratory. Over 70 laboratory personnel covering all major sub-business areas of the laboratory have been educated on the use of the software estimation tools. Several projects from each sub-business area have experimented with or used the SLIM and COSTAR tools. Client contacts have been established within and outside the laboratory. Five of the seven products submitted by the laboratory to the Market-Driven Quality (MDQ) Assessment in 1993 stated that they used estimation models and tools as their initiatives to improve their overall estimation process and the accuracy of their project estimates.

3.0 Key Lessons Learned

The experience we have described is based on over three years of solid work. The process we have followed can be applied in general to the deployment of other techniques and tools. Following are the key lessons learned from this experience.

3.1 Technology Injection Takes Time

Deploying state-of-the-art technology and tools takes time. Table 1 shows the elapsed time for each stage of deployment and the effort required on the part of the technology champions for each stage shown in Figure 1.

Table 1. Time and Effort for Technology Injection

Deployment Stage	Time (months)	Effort (PMs)
Understanding	7	7
Installation	12	14
Adoption	18 (On-going)	21 (0.2 PM/mo)

It took 37 calendar months and 42 person-months (PMs) of effort on the part of the champions to inject the technology and tools to the point where only 0.2 person-months per month is now required to maintain the level of deployment.

Users have to overcome many barriers to become knowledgeable in the field. In addition to learning the methodologies and tools, they have to learn about accessing the tools through the LAN or installing the tools (if not LAN connected). In some situations, users may have to configure, install, or upgrade certain components of the operating system and learn about it prior to using the tools. These are overhead tasks the users must face before any true benefit in adopting the methodologies and tools can be realized.

3.2 Management Commitment Is Essential

Long-term management commitment is essential to the successful deployment of technology and tools hitherto foreign to an organization. Management support is critical for both the consultants and the client organizations in terms of time and resource allocation to tackle the overhead tasks, education, cost of software and hardware, etc.

3.3 Champions Must Be Pro-active and Proficient

The technology champions must be in a position to give advice, provide consultation, and offer assistance. They must be able to conduct thorough analyses of project estimates, and point out both the strengths and weaknesses of the methodologies and tools to their clients.

3.4 Easy Access to Tools and Information Facilitates Deployment

The Toronto Lab Common LAN facilitates license sharing and tool invocation. There is a cost-saving benefit since acquisition of individual copies of software for each end user is avoided. Furthermore, end users are relieved from the burden of tools upgrade and maintenance.

It is important to document tool information, formal tool evaluation results, pilot results and user feedback, and to keep these documents up-to-date. The use of online surveys captures valuable tools experience that will benefit the other users within the laboratory and will help in defining the strategy for software estimation techniques and tools in the future.

3.5 Increasing the Laboratory Community's Awareness Promotes Buy-In

Demonstrations and technical exchange sessions are useful for introducing new technology and tools to the laboratory. These occasions have given some people an increased awareness in the area of software

estimation and allowed others to gain in-depth technical knowledge.

3.6 Broad-Based Customized Education Is Effective

Broad-based customized education is highly effective and rewarding. We strongly encourage the same infrastructure in deploying technology and tools in other areas. It saves the organization money. Course participants typically get more value from a course taught by local experts using real development data collected within the laboratory on more than one model and tool, compared to one taught by a tool vendor. Vendor courses tend to teach limited theory and are confined to their product offering.

3.7 Historical Data Collection Is Crucial

The collection of historical data is critical to process improvement. There is a crucial need to continuously capture historical data on in-process project parameters. The estimated and actual values of the schedule, resource allocation, defects, etc. should be collected to improve the quality of subsequent estimation. Having this data is critical for calibrating commercially available estimation tools and tuning them to the development environment.

3.8 Understanding How Data Will Be Used Is Essential

Many software developers resist capturing estimates and the actual values of project parameters They are afraid of how the numbers or measures will be used or misused by management or other groups. It is important to make them understand that the collected data will help managers identify strong points and bottlenecks, and help them set realistic goals for future software development projects.

4.0 Future Directions

Although the SLIM and COSTAR tools have been successfully deployed, much work still remains. In addition to the technology injection techniques discussed in the earlier sections (for example, demonstrations, lectures), users group meetings should be conducted periodically to update the users on the latest developments or breakthroughs. The group meetings will also provide opportunities for the users to exchange ideas and experience.

Another area that requires immediate attention is the technical assessment, evaluation, and recommendation of size estimation techniques and tools. Size estimates are critical inputs to software estimation models and tools. Other related activities that complement estimation are tracking and project management. The feasibility of integrating software estimation tools with project management tools should also be investigated.

As product development groups switch from the traditional approaches to object-oriented development, the models for software estimation are expected to change accordingly. It is unclear at this moment how well the existing software estimation models apply to object-oriented software development.

5.0 Acknowledgements

The authors thank Ann Gawman who provided valuable editorial improvements.

References

[1] John W. Bailey and Victor R. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the IEEE Fifth International Conference on Software Engineering*, 1981.

[2] V. R. Basili, "Software Development: A Paradigm for the Future," *Proceedings of the IEEE 13th International Computer Software and Applications Conference*, 1989.

[3] V. R. Basili, G. Caldiera and F. McGarry, et al, "The Software Engineering Laboratory - An Operational Software Experience Factory," *Proceedings of the IEEE 14th International Conference on Software Engineering*, 1992.

[4] V. R. Basili and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," *Proceedings of the IEEE Ninth International Computer Software and Applications Conference*, 1985.

[5] Barry W. Boehm, *Software Engineering Economics*, Englewood Cliffs: Prentice-Hall, Inc., 1981.

[6] Kelvin B. Fowler, A Software Cost Estimating Tool for IBM., IBM ASD-Bethesda, Technical Report 86.0021, 1991.

[7] Danny Ho, Deploying Software Development Tools on the Toronto Lab Common LAN, IBM PRGS Toronto Laboratory, Technical Report 74.112, 1993.

[8] Danny Ho, Software Estimation Using the COCOMO Model and the COSTAR Tool, IBM SWS Toronto Laboratory, Technical Report 74.135, 1994.

[9] Nikki Panlilio-Yap, Software Estimation Using the SLIM Tool, IBM Canada Ltd. Laboratory, Technical Report 74.102, 1992.

[10] Nikki Panlilio-Yap and Danny Ho, "Deploying Software Estimation Technology and Tools: The IBM Software Solutions Toronto Lab Experience," *Proceedings of the Ninth International Forum on COCOMO and Software Cost Modeling*, 1994.

[11] Lawrence H. Putnam, *Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers*, New York: The Institute of Electrical and Electronics Engineers, Inc., 1980.

[12] Lawrence H. Putnam, *MEASURES FOR EXCELLENCE Reliable Software on Time, within Budget*, Englewood Cliffs: Yourdon Press., 1992.

[13] Quantitative Software Management, Inc., SLIM Version 2.2 User Manual, McLean, 1991.

[14] Quantitative Software Management, Inc., SLIM Version 3.1 User Manual, McLean, 1993.

[15] Softstar Systems, COSTAR Version 3.0 User Manual, Amherst, 1990.

About the Authors

Nikki Panlilio-Yap works in the Software Engineering Department, Group Technical Staff, at Loral Federal Systems (formerly IBM Federal Systems Company) in Bethesda, Maryland. She is presently on a leave of absence from the IBM SWS Toronto Laboratory where she had been part of the Software Engineering Process Group (SEPG) since its inception in August 1990. She joined IBM Canada Ltd. in July 1989 and worked in Utilities Product Evaluation for the AS/400* system. Before joining IBM, she had several years work experience in the government, academic, and commercial sectors of the computing industry. She obtained her Bachelor of Science in Chemical Engineering from the University of the Philippines, Master of Arts in Computer Science from Duke University in Durham, North Carolina, and Master of Science in Computer Science from the University of Maryland at College Park. She was a Fulbright Scholar and a World Fellowship recipient of the Delta Kappa Gamma Society International. She is a member of the Honor Society of Phi Kappa Phi and the IEEE Computer Society.

Danny Ho works in the IBM Microelectronics Toronto Laboratory as a team leader in infrared wireless development. He joined the IBM SWS Toronto Laboratory in 1989 and worked in the Tools and Technology Group for four years. His areas of special interest are software estimation, object-oriented software development, complexity analysis, tools delivery mechanisms, and tools platforms. Prior to joining IBM, Danny worked as a Software Engineer in the Communications Division of Motorola Canada Limited and was responsible for the analysis, design, and implementation of wireline and radio frequency communication systems and protocols. Danny received his Honours Bachelor of Science in Computer Science with Electrical Engineering and Master of Science in Computer Science from the University of Western Ontario. He is currently a member of the Association of Professional Engineers of Ontario.

Lessons Learned in Deploying Software Estimation Technology and Tools

November 30, 1994

Nikki Panlilio-Yap and Danny Ho
IBM Canada Limited
Software Solutions Division
Toronto Laboratory
844 Don Mills Road
North York, Ontario M3C 1V7
Canada

Internet: nikki@lfs.loral.com &
danho@torolab2.vnet.ibm.com

AGENDA

- Introduction
- Stages of Deployment
- Level of Deployment
- Key Lessons Learned
- Future Directions

DESCRIPTION OF ORGANIZATION

- Sub-businesses:
  - Application Development Technology Center
  - Database Technology
- Number of projects/products: close to 50
- Number of people: approx 1300 (approx 1000 developers)
- Skill Mix: OS/2, AIX, OS/400, VM
- SEI assessment in 1991 revealed a critical need for software estimation techniques and tools
- Joint effort by Software Engineering Process Group, and Tools and Technology Group to assess, evaluate, recommend and deploy

SOFTWARE ESTIMATION

Estimate duration, effort, cost and reliability of software development, based on product size

Why is software estimation important?

- Crucial to project schedule and staff/resource allocation
- Uncertainty of project parameters in planning stages
- Cross life-cycle discipline which applies to all phases of software development
- Vital to assess parameters at various stages of the project and re-estimate if necessary

SOFTWARE ESTIMATION

Why use estimation models?

- Form basis for disciplined planning
- Calibrate to experience
- Allow sensitivity and what-if analysis
- Provide insights in productivity and quality improvement
- Validate bottom-up estimates

Models are not perfect, so use more than one

STAGES OF DEPLOYMENT

Understanding - The Early Stage

Installation - Making the Selected Tools Available

Adoption - Expanding the User Base

UNDERSTANDING - THE EARLY STAGE

- Gather preliminary information
  - Literature search
  - Detailed information or demonstration diskette from vendors
- Obtain trial license
- Conduct pilots
- Perform technical assessment and evaluation

INSTALLATION - MAKING THE SELECTED TOOLS AVAILABLE

- Recommend the selected models and tools (SLIM and COSTAR) based on results of pilot experiments
  - Level of input required
  - Comparison with project actual data
  - User satisfaction
- Demonstrations and technical exchange sessions
  - One-on-one
  - Group
  - Public forum
- Direct project involvement - provide consultation and advise on:
  - Model and tool usage
  - Tool calibration

ADOPTION - EXPANDING THE USER BASE

- Broad-based education: two-day course
  - Teach underlying theories
  - Provide in-depth training on the selected tools
  - Provide hands-on experimentation with the tools
- Lab-wide consulting
- Tool and information availability
  - Experience Warehouse
  - Software Lending Library
  - Common LAN

LEVEL OF DEPLOYMENT

- Offered 5 Software Estimation and Tools courses
- Trained over 70 laboratory personnel
- 5 of 7 products submitted for Market-Driven Quality Assessment in 1993 have used estimation models/tools
- Established client contacts within and outside the laboratory

KEY LESSONS LEARNED

Deploying state-of-the-art technology and tools takes time

Table 1. Time and Effort for Technology Injection

Deployment Stage	Time (months)	Effort (PMs)
Understanding	7	7
Installation	12	14
Adoption	18 (On-going)	21 (0.2 PM/mo)

KEY LESSONS LEARNED

Management commitment is essential

- Need long-term management commitment of time and resources

Champions must be pro-active and proficient

- Must be in a position to give advice, provide consultation, and offer assistance

KEY LESSONS LEARNED

Easy access to tools and information facilitates deployment

- LAN facilitates license sharing, tool invocation, tool upgrade and maintenance
- Online surveys capture valuable tools experience
- Access to tool information, formal tool evaluation results, pilot results, and user feedback help others in defining strategy

Increasing the laboratory community's awareness promotes buy-in

- Demonstrations and technical exchange sessions are useful for introducing new technology and tools

KEY LESSONS LEARNED

Broad-based customized education is highly effective

- Create local focal points in the product areas
- Deploy more than one theory and tool
- Tailor course to suit local development environment
- Reduce cost

Collection of historical data is crucial to process improvement

- Improve the quality of subsequent estimation
- Calibrate commercially available tools and tune them to the development environment

KEY LESSONS LEARNED

Understanding how collected data will be used is essential

- Reduce developers' resistance to capturing estimates and actual values of project parameters
- Help managers identify strong points and bottlenecks
- Help set realistic goals for future projects

FUTURE DIRECTIONS

- User group meetings
  - Update users on latest developments
  - Provide opportunities for exchange of ideas and experience
- Size estimation and project tracking – new areas to investigate
- Software sizing, estimation, project tracking and management tools should be integrated
- Tools that truly exploit LAN
  - Client-server computing model
  - Using servers as repository for both data and software
  - Utilize remote LAN data services
- Object-oriented software development – how well do these models fit?

Session 5: Reliability and Safety

Using Formal Methods for Requirements Analysis of Critical Spacecraft Software
Robyn Lutz, Jet Propulsion Laboratory

Experimental Control in Software Reliability Certification
Carmen Trammell, University of Tennessee

Generalized Implementation of Software Safety Policies
John Knight, University of Virginia

Experience Report: Using Formal Methods for Requirements Analysis of Critical Spacecraft Software

Robyn R. Lutz *
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

Yoko Ampo †
NEC Corporation
Tokyo, Japan

November 21, 1994

Formal specification and analysis of requirements continues to gain support as a method for producing more reliable software. However, the introduction of formal methods to a large software project is difficult, due in part to the unfamiliarity of the specification languages and the lack of graphics. This paper reports results of an investigation into the effectiveness of formal methods as an aid to the requirements analysis of critical, system-level fault-protection software on a spacecraft currently under development. Our experience indicates that formal specification and analysis can enhance the accuracy of the requirements and add assurance prior to design development in this domain.

The work described here is part of a larger, NASA-funded research project whose purpose is to use formal-methods techniques to improve the quality of software in space applications [2]. The demonstration project described here is part of the effort to evaluate experimentally the effectiveness of supplementing traditional engineering approaches to requirements specification with the more rigorous specification and analysis available with formal methods.

The approach taken in this investigation was to:

1. Select the application domain. The primary criteria were, first, to select portions of the *requirements* of an large, embedded software project currently under development, and, secondly, to select *mission-critical software*, meaning that its failure could jeopardize the spacecraft system or mission.

   The selected applications were the requirements for portions of the Cassini spacecraft's system-level fault-protection software. This on-board software autonomously detects and responds to faults that occur during operations. About 85 pages of documented requirements describing the software that commands the spacecraft to a known safe

*First author's mailing address is Dept. of Computer Science, Iowa State University, Ames, IA 50011.

†Second author's mailing address is Space Station Systems Division, NEC Corporation, 4035 Ikebe-cho, Midori-ku, Yokohama 226, Japan. This work was performed while the author was a visiting researcher at Jet Propulsion Laboratory, Pasadena, CA 91109.

state and a software executive that manages the fault protection were involved in the study. System-level fault protection was targeted as a domain which merited the extra assurance possible with formal specification and analysis.

2. Model the selected applications using object-oriented diagrams. The object-oriented modeling tool used in this work was Paradigm Plus, an implementation of OMT, the Object Modeling Technique [6] [1]. This effort built on earlier work in this research project in which OMT diagrams were found to be a useful complement to formal specification in a reverse-engineering application [1]. Our work differs in that we applied OMT to software currently in the process of being developed, with formal proofs as well as formal specifications being created.

3. Develop formal specifications. The formal specification language used in this study was that of PVS, the Prototype Verification System [8]. PVS is an integrated environment for developing and analyzing formal specifications including support tools and a theorem prover.

4. Prove required properties. We determined properties that must hold for the target software to be hazard-free and function correctly, specified them in PVS as lemmas (claims), and proved or disproved them using the interactive theorem-prover.

5. Feedback results to the Project. Because we were analyzing requirements that were still being updated, part of our task was to keep current with the changes and to provide timely feedback to the Project as they resolved the remaining requirements issues and began design development.

The experiment described here produced 25 pages of PVS specifications and 15 pages of OMT diagrams. 37 lemmas were specified. Of these, 21 were proven to be true and 3 were disproven. An additional 13 lemmas were stated but not proven. Five of these unproven lemmas were obviously true from the formal specifications; four were out of the scope of our application; and four remain to be proven. The lemmas that were proved were claims or challenges which must be true if the specifications are accurate and the requirements are hazard-free.

The lemmas were divided into three categories: requirements-met, safety, and liveness properties. Requirements-met lemmas traced the documented requirements to the formal specifications. For example, a documented requirement "If a response can be initiated by more than one monitor, each monitor shall include an enable/disable mechanism" led to a lemma demonstrating that the specifications satisfied this requirement. We proved or disproved 10 such requirements-met lemmas.

Safety properties were "shall-not" claims, which can be stated informally as "nothing bad ever happens [9]." Examples are, "The software shall not activate any response that is not requested by a monitor" and "The response shall not change the instrument's status during a critical sequence of commands." We were able to prove 7 such safety properties, adding assurance that the software did not introduce hazards into the system.

[1] Paradigm Plus is a registered trademark of Protosoft, Inc.

Liveness properties described the positive aspects of the correct behavior of the software: "something good eventually happens [9]." Examples are, "If a response has the highest priority among the candidates and does not finish in the current cycle, it will be active in the next cycle" and "If the response occurs during a non-critical sequence of commands, then the instrument is turned on." We proved 7 such liveness properties, adding assurance that no hidden assumptions were required for the software to function correctly.

The results obtained from the specification and analysis (including proofs) of the requirements were of two types: issues found in the requirements and an evaluation of the process itself.

A total of 37 issues were found in the requirements. These were categorized as follows:

- Undocumented assumptions: 11. The formalization of the requirements revealed several assumptions that were not explicit in the documentation. An example of such an assumption is, "if the spacecraft is in a critical attitude, then the software is executing a critical sequence of commands." Frequently, these assumptions involved interface issues between software modules or subsystems, historically a frequent source of errors that persist until system testing [4]. In almost every case, the hidden assumption was currently correct. However, several assumptions merited documentation, especially since future changes can invalidate current assumptions.

- Inadequate requirements for off-nominal or boundary cases: 10. These issues usually involved unlikely scenarios in which a pre-condition could be false. We often had to consult spacecraft engineers to know whether such boundary cases were credible. For example, the case in which several monitors with the same priority level detect faults in the same cycle was not described. By concretely specifying the possibility of off-nominal scenarios, the formal analysis can contribute added robustness to the system.

- Traceability and inconsistency: 9. These issues included lack of traceability between the high-level requirements and low-level requirements, as well as inconsistency between the software requirements and the design of subsystems. Many of these issues were significant in that they could affect both the logic and the correctness of the formal specifications. An example is that although the high-level requirements assume that multiple detections of faults occuring within the response time of the first fault detected are symptoms of the original fault, the lower-level requirements (correctly) cancel a lower-priority fault response to handle a higher-priority response.

- Imprecise terminology: 6. These were documentation issues, frequently involving synonyms or related terms. The definition of types in PVS enforced their resolution.

- Logical error: 1. The logical error involved the handling of a request for service from a monitor in the case that a higher-priority request occurred. The question as to whether such a request could face starvation was first raised during the initial close reading. The formalization of the issue as a lemma which could be disproven provided insight and certainty.

The evaluation of the process we used to specify and analyze the requirements led us to three conclusions:

1. *Using object-oriented models.* For the target applications, object-oriented modeling offered several advantages as an initial step in developing formal specifications. First, the object-oriented modeling defined the boundaries and interfaces of the embedded software applications at the level of abstraction chosen as appropriate by the specifiers. In addition, the modeling offered a quick way to gain multiple perspectives on the requirements. Finally, the graphical diagrams served as a frame upon which to base the subsequent formal specification and guided the steps of its development. Since the elements of the diagrammatic model often mapped in a straightforward way to elements of the formal specifications, this reduced the effort involved in producing an initial formal specification. We also found that the object-oriented models did not always represent the "why," of the requirements, i.e., the underlying intent or strategy of the software. In contrast, the formal specification often clearly revealed the intent of the requirements.

2. *Using formal methods for requirements analysis.* Unlike earlier work in this research project on software in which the requirements were very mature and stable and the formal specification entailed reverse engineering (Space Shuttle's Jet Select Subsystem), the work on Cassini's fault-protection subsystem analyzed requirements at a much earlier phase of development. Consequently, the requirements that we analyzed were known to be in flux, with several key issues still being worked (e.g., timing details, number of priority levels). A negative effect of the lack of stability was that time was spent staying current with changes. A positive effect was that issues identified during our analysis could be readily fed back into the development process before the design was frozen.

   We were concerned as to whether it was a waste of time to formally specify requirements while they were still likely to change. Certainly, there was inefficiency in rewriting specifications to conform to changes that occurred during the experiment. However, based on our experience with this trial project, the formal specification of unstable requirements had the following advantages:

   - Laid the foundation for future work.
   - Allowed rapid review of proposed changes and alternatives.
   - Clarified requirements issues still being worked by elevating undocumented concerns to clear, objective dilemmas.
   - Complemented the lower-level FMEA (Failure Modes and Effects Analysis) already being perfomed on the software, by providing higher-level verification of system properties.
   - Added confidence in the adequacy of the requirements that had been analyzed using formal methods.

   Rushby's recent study of formal methods for airborne systems reached the similar but even stronger conclusion that formal methods can be most effectively applied early in the lifecycle [7].

3. *Using formal methods for safety-critical software.* For a safety analysis it is important to ensure that a hazardous situation does not occur, as well as that the correct behavior does occur [5]. Fault Tree Analysis, which backtracks from a hazard to its possible causes, is one method used for this kind of hazards analysis [3]. However, unlike formal methods of specification and proof, Fault Tree Analysis is an informal method which in practice permits ambiguous or inadequate descriptions.

   Formal methods helped us find hazardous scenarios by forcing us to show every condition and prompting us to define new, undocumented assumptions. The process of developing formal specifications and proofs led us to think about the full range of cases, some of which were unanticipated.

In conclusion, one of the goals of the larger research project within which this investigation was performed is to evaluate the effectiveness and practicality of formal methods for enhancing the development process and the reliability of the end product. Our main contributions to this work in the Cassini demonstration project have been:

- Applying formal methods to the software requirements analysis of a project currently under development,
- Using object-oriented diagrams to guide the formal specification of software requirements,
- Formally specifying and proving a set of properties essential for the correct and hazard-free behavior of the software, and
- Demonstrating that formal methods can be used to specify and analyze an application involving critical software.

Acknowledgments

Other contributors to the formal methods work at Jet Propulsion Laboratory are Rick Covington, John Kelly, and Allen Nikora. Ken Abernethy contributed to this work while visiting JPL. The authors also thank Sarah Gavit and Jan Berkeley for helpful discussions.

The work described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitue or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

References

[1] B. H. C. Cheng and B. Auernheimer, "Applying Formal Methods and Object-Oriented Analysis to Existing Flight Software," *Proc 18th Annual Software Eng Workshop 1993*, NASA/Goddard Space Flight Center, SEL, Dec 1993, 274–282.

[2] *Formal Methods Demonstration Project for Space Applications, Phase I Case Study: Space Shuttle Orbit DAP Jet Select*, JPL, JSC, and LARC, December 1993.

[3] N. G. Leveson, "Software Safety in Embedded Computer Systems," *Commun ACM*, 34, 2, Feb 1991, 35–46.

[4] R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proc IEEE Internat Symp on Requirements Eng.* IEEE Computer Society Press, 1993, 126-133.

[5] *NASA Software Safety Standard*, NSS 1740.13, Interim, June, 1994.

[6] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[7] J. Rushby, *Formal Methods and Digital Systems Validation for Airborne Systems*, SRI-CSL-93-07, Nov 1993.

[8] N. Shankar, S. Owre, and J. M. Rushby, *The PVS Specification and Verification System*, SRI, March, 1993.

[9] J. M. Wing. "A Specifier's Introduction to Formal Methods," *IEEE Computer*, 23, 9, Sept 1990, 8–24.

Experience Report: Using Formal Methods For Requirements Analysis of Critical Spacecraft Software

Robyn R. Lutz
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

Yoko Ampo
NEC Corporation
Tokyo, Japan

Nineteenth Annual Software Engineering Workshop
December 1, 1994

The work described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

Introduction

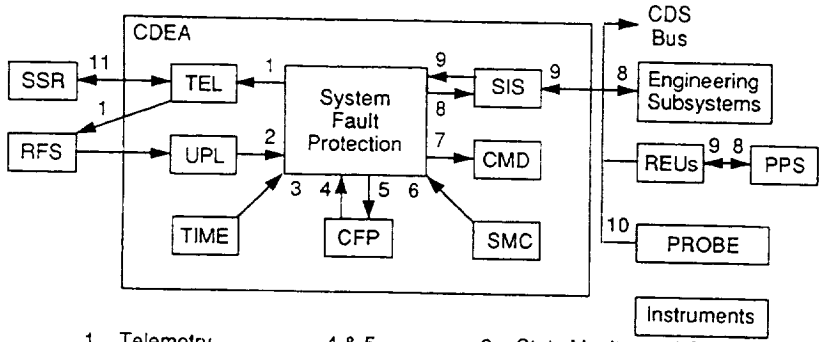
- Task is part of NASA RTOP to demonstrate Formal Methods techniques and their applicability to critical NASA software systems. (RTOP: Research Technical Objectives and Plans)
- Formal Methods (FM) refer to the use of techniques and tools based on formal logic and mathematics to specify and verify systems, software, and hardware.

Approach

- **Step 1: Select Application**
  - » **Criteria:**
    - **Software requirements**
    - **Currently under development (critical software failure could jeopardize system or mission)**
  - » **Selection:**
    - **Requirements for portions of Cassini spacecraft's system-level fault protection software**
    - **Autonomous detection, isolation, and recovery from on-board faults required**

CDS Fault Protection

CDS Interfaces to SFP–2



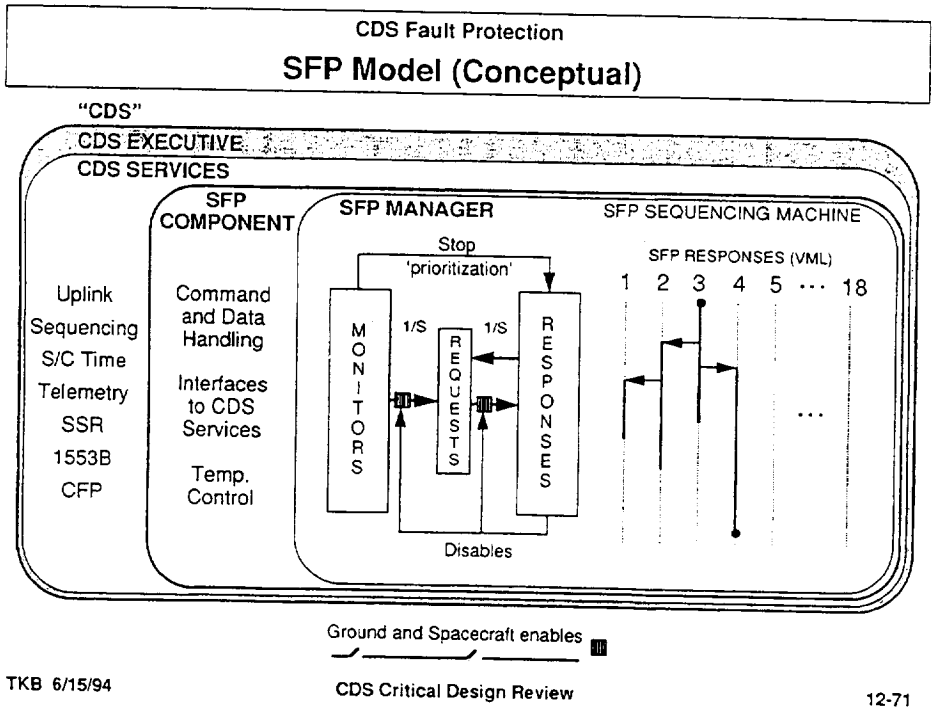
1 Telemetry
2 Uplink
3 Time Services
11 Memory Loading and SSR dual recording

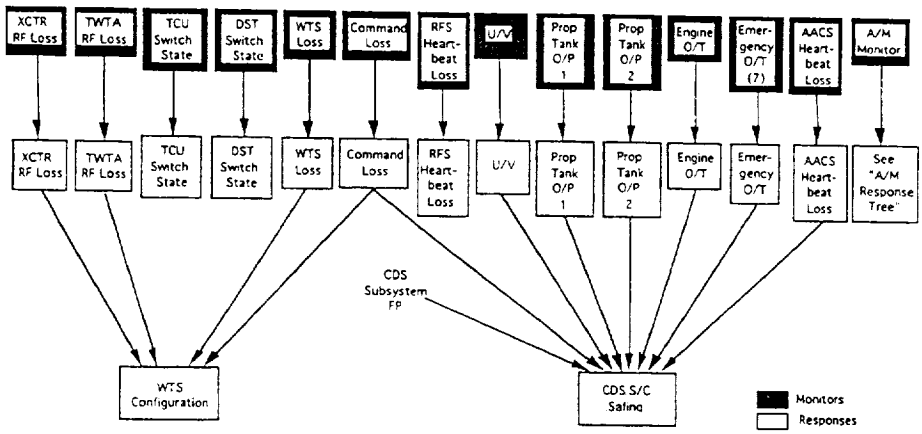
4 & 5 CDS Fault Protection

6 State Monitor and Control
7 Commanding
8 S/C Intercommunication RT Reception
9 S/C Intercommunication RT Transmission
10 Dual recording (from both PSAs to both SSRs)

Approach (continued)

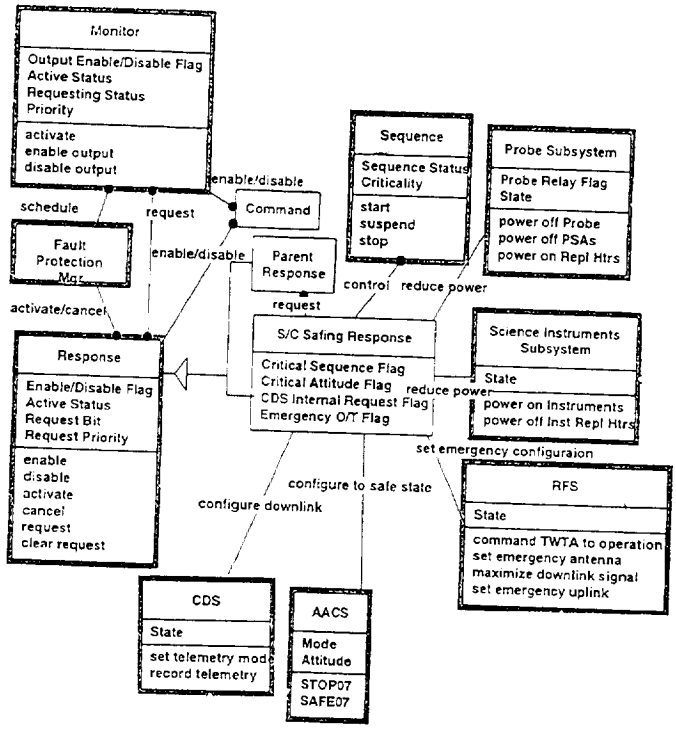
- **Safe State Response**
  - » Mission-phase dependent
  - » Commands safe attitude, minimizes power usage, cancels non-essential activities, reconfigures hardware
- **Fault Recovery Executive**
  - » Selects which request to service
  - » Preemptive priority scheme
  - » Special cases complicate requirements





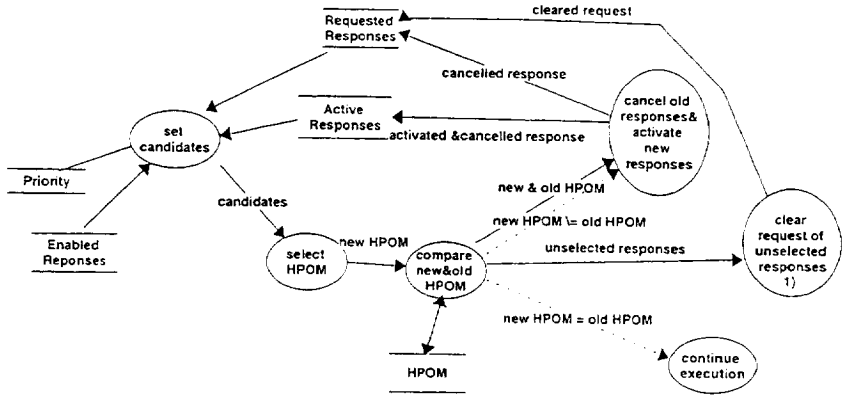
Approach (continued)

- Step 2: Model with Object-Oriented Diagrams
  - » Builds on earlier RTOP work [Cheng and Auernheimer, 93]
  - » Object Modeling Technique (OMT) tool [Rumbaugh, et. al., 91], Paradigm Plus® [Protosoft]
- Step 3: Develop formal specifications
  - » OMT diagrams guided specification
  - » Formal specification language was that of PVS (Prototype Verification System) [Shanker, Owre, Rushby, 93]



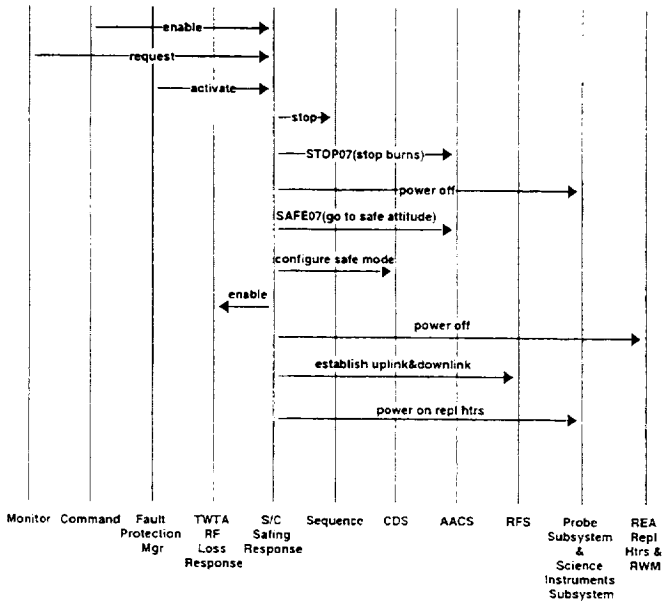
S/C Safing Response Object Diagram

Fault Protection Mgr Functional Diagram



1) CAS-3-331 dated Jan.28 says that only cancelled responses have their requests cleared.

S/C Safing Response Event Trace



Approach (continued)

- **Step 4: Prove required properties**
  - » **Specify properties in PVS as claims to be proven**
  - » **Prove/disprove claims using interactive theorem prover**
- **Step 5: Feedback results to Cassini Project**

Results

- Summary: 15 pages of OMT diagrams
  25 pages of PVS specifications
  37 properties specified as claims
    - 24 proven/disproven
    - 5 true from specifications
    - 4 out of scope
    - 4 remain to prove
- Two types of results:
  - » Issues found in documented requirements
  - » Evaluation of process

Results: Issues Found

- 3 categories of claims specified and proven
  - » "Requirements-met"
    - Demonstrate that formal specifications accurately represent key requirements
    - Example: "If a response can be initiated by more than one monitor, each monitor shall include an enable/ disable mechanism."
    - 10 proven/disproven, adding assurance that specifications are correct
  - » Safety properties
    - "Shall-not" claims that "nothing bad ever happens" [Wing, 90]
    - Example: "The response shall not change the instrument's status during a critical sequence of commands."

Results: Issues Found (continued)

» Safety properties (continued)
- 7 proven, adding assurance that software does not introduce hazards into system
- Example: "The response shall not change the instrument's status during a critical sequence of commands."

» Liveness properties
- Describe correct behavior: "something good eventually happens" [Wing, 90]
- Example: "If a response has the highest priority among the candidates and does not finish in the current cycle, it will be active in the next cycle."
- 7 proven/disproven, adding assurance that no hidden assumptions required for correct behavior


```
saf:  THEORY

% Example below is excerpted from saf theory.

% Spacecraft safing commands the AACS to homebase mode, thereby
% stopping delta-v's and desats.

BEGIN

aacs_mode:  TYPE = {homebase, detumble}
attitude:   TYPE

cds_internal_request:  VAR bool
critical_attitude:     VAR bool
prev_aacs_mode:        VAR aacs_mode

aacs_stop_fnc (critical_attitude, cds_internal_request, prev_aacs_mode):
 aacs_mode =
 IF critical_attitude
    THEN IF cds_internal_request
            THEN prev_aacs_mode
            ELSE homebase
         ENDIF
    ELSE homebase
 ENDIF

% Lemma asserts that if Spacecraft Safing is requested via a CDS internal
% request while the spacecraft is in a critical attitude, then no change is
% commanded to the AACS.  Otherwise, the AACS is commanded to homebase.

 aacs_safing_req_met_1:  LEMMA
   (critical_attitude AND cds_internal_request)
   OR (aacs_stop_fnc(critical_attitude, cds_internal_request, prev_aacs_mode)
       = homebase)

END saf
```

Results: Issues Found (continued)

- 37 issues found:
  - » Undocumented assumptions: 11
    - Example: "If the spacecraft is in a critical attitude, then the software is executing a critical sequence of commands."
    - Frequently involved interface issues, historically a source of errors that persist until integration and system testing [Lutz, 93]
    - Assumptions almost always currently correct, but future design changes could invalidate them.
  - » Inadequate requirements for off-nominal or boundary cases: 10
    - Example: Requirements for case in which several monitors with same priority level detect faults in same cycle were not described

Results: Issues Found (continued)

- » Inadequate requirements for off-nominal or boundary cases (continued)
  - Involved unlikely scenarios in which pre-condition could be false
  - Concretely specifying possible cases builds in robustness
- » Traceability and inconsistency: 9
  - Example: High-level requirements assume that detected faults occurring during response time of initial fault are symptoms of initial fault; low-level requirements (correctly) cancel lower-priority response
  - Formal specification forced resolution of discrepancies

Results: Issues Found (continued)

» Imprecise terminology: 6
- Example: "Stop" and "cancel" sometimes synonymous; sometimes not
- Automatic type checking enforced precision

» Logical error: 1
- Example: can a request for service face starvation due to higher-priority requests?
- Formalizing question as lemma which could be disproven provided insight and certainty

Results: Process Evaluation

- Benefits of combining Object-Oriented Models and Formal Methods
  - » Frames the problem
  - » Basis for technical discussion
  - » Road map
    - Mapping of elements
    - Reduced effort
  - » Complementary roles
    - OMT: informal
      multiple perspectives
      communicates key elements
    - PVS: formal
      unambiguous specification
      analysis of completeness

Results: Process Evaluation (continued)

» OO model did not represent the "why" of the requirements (underlying intent or strategy) as clearly as the formal specifications

- Using Formal Methods for requirements analysis
  - » Requirements were not yet stable
  - » Waste of time to formally specify?
    - Time consuming to stay current
    - Interactive process

Results: Process Evaluation (continued)

» Advantages of formal specification of unstable requirements

- Laid foundation
- Rapid review of proposed changes
- Clarified issues being worked: undocumented concerns elevated to clear, objective dilemmas
- Complemented lower-level FMEAs (Failure Modes and Effects Analyses)
- Added confidence in adequacy of requirements analyzed using formal methods
- Issues identified fed back and resolved early in development

Results: Process Evaluation (continued)

- Using Formal Methods for safety-critical software
  - » FM helped find hazardous situations
  - » Forced analysis of full range of cases, some unanticipated
  - » Prompted definition of undocumented assumptions, some of which are not always true
  - » Proofs of safety properties ensured that some unsafe states do not occur

Conclusion

- Contributions of this work:
  - » Applied FM to software requirements of project currently in development
  - » Used object-oriented diagrams to guide formal specifications of requirements
  - » Formally specified and proved some properties essential for correct and hazard-free behavior
  - » Demonstrated use of FM in safety-critical application

EXPERIMENTAL CONTROL IN SOFTWARE RELIABILITY CERTIFICATION

Carmen J. Trammell and Jesse H. Poore
Software Quality Research Laboratory
University of Tennessee

There is growing interest in software "certification," i.e., confirmation that software has performed satisfactorily under a defined certification protocol. Regulatory agencies, customers, and prospective reusers all want assurance that a defined product standard has been met.

In other industries, products are typically certified under protocols in which random samples of the product are drawn, tests characteristic of operational use are applied, analytical or statistical inferences are made, and products meeting a standard are "certified" as fit for use. A warranty statement is often issued upon satisfactory completion of a certification protocol.

The statistical principles that underlie such product protocols have long been advocated by Mills and colleagues [1,2,3,4] and Musa and colleagues [5,6,7] as the basis for software reliability certification. The terminology used by Mills and Musa differs slightly, but their ideas are similarly drawn from scientific approaches to product certification in mature engineering disciplines. The terminology of Mills will be used in this paper.

"Statistical testing" was conceived by Mills and has been advanced by his colleagues at IBM, Software Engineering Technology Inc., and the University of Tennessee. In statistical testing,

(1) expected operational use is represented in a usage model of the software,

(2) test cases are randomly generated from the usage model,

(3) test cases are executed in an environment that simulates the operational environment, and

(4) failure data are interpreted according to mathematical and statistical models.

Methods for the construction of usage models (8,9) and the interpretation of failure data (10) have been given. Usage models are developed *before* testing, and interpretation of failure data occurs *after* testing. Proper experimental control *during* testing is critical to the integrity of the protocol, however, and has not previously been addressed.

This paper outlines specific engineering practices that must be used to preserve the validity of the statistical certification testing protocol. The assumptions associated with a statistical experiment are given, and their implications for statistical testing of software are described. The ideas in this paper have evolved from experience in fifteen Cleanroom projects conducted in the Software Quality Research Laboratory at the University of Tennessee.

The Slippery Slope

It was a typical day in the testing phase of a software development project at ACME Software.

Jane had been testing for hours, and her mind was drifting. She took a break. When she returned and ran the next test case, she noticed something unexpected, but she knew this unexpected event had to have been happening all along. She realized she had been too tired to observe it when it first occurred. She didn't know when it had first shown up.

John and Mary were both running test cases. John saw a screen event and

thought it was expected behavior. Mary saw the same event and recorded it as unexpected behavior.

Joe suddenly realized that there was an error in his part of the code, and he was anxious to fix it. He waited until testers had stopped for the day, made the change, and recompiled. The testers would continue their work the next day using his new version. He knew he had made the change and recompiled properly, so there was no need to bother the test team about this.

Michael looked over the stack of test cases and saw that they varied greatly in length. He knew that they had been randomly generated, so he assumed that they were all equally usable test cases. He rifled through the stack and picked the shortest ones so he could run the most cases in the least time.

Deborah was a new hire assigned to take the place of a certification engineer who left the company abruptly. She worked with the experienced engineer for a day, and then started testing on her own. She couldn't really read the spec to check the details of correct output, so she decided to just use her best judgment and not bother the others unless she was really confused.

Bill had an extremely long test case. In the middle of the test case, the prescribed events led him back to the Main Menu. Ordinarily, a test case would end at this point, but this case called for a second major scenario. Bill decided the case was unreasonably long, and counted the second major scenario as a new test case.

These very common events are threats to the integrity of a statistical approach to software testing. Statistical software testing, as a scientific endeavor in the real world, inevitably requires some compromises in methodological purity, and it is important to understand the nature of the slippery slope. The assumptions underlying a statistical experiment must be understood, the practical threats to experimental integrity must be recognized, and a strategy for experimental control must be employed.

Software Testing as a Statistical Experiment

In statistical certification testing, software testing is viewed as a statistical experiment. A subset of all possible uses of the software is generated, and performance on the subset is used as a basis for conclusions about general operational reliability. In standard experimental parlance, a "sample" is used to draw conclusions about a "population." Figure 1 shows the parallel between a classical statistical experiment and statistical software testing. Under a testing protocol that is faithful to the principles of applied statistics, a scientifically valid statement can be made about the expected operational performance of the software based on its test performance.

The premise that must be accepted as a starting point in this analogy is that it is not possible to test *all* ways in which software may be used. This is apparently not a premise that can be assumed as obvious. In a discussion of software testing with the top software manager in a large aerospace corporation, the infeasibility of testing all possible usage scenarios was cited as the motivation for statistical testing. "But we *have* to test every possible use of the software," he said. "The kind of software we develop could cause deaths if it is not tested completely."

Software with an unbounded input sequence length has a theoretically infinite number of possible usage scenarios. For software with only two user inputs, A and B, the possible scenarios of use are A, B, AA, AB, BB, BA, AAA, AAB, ABA, BAA, and so on. Software with a bounded but large input sequence length has a finite

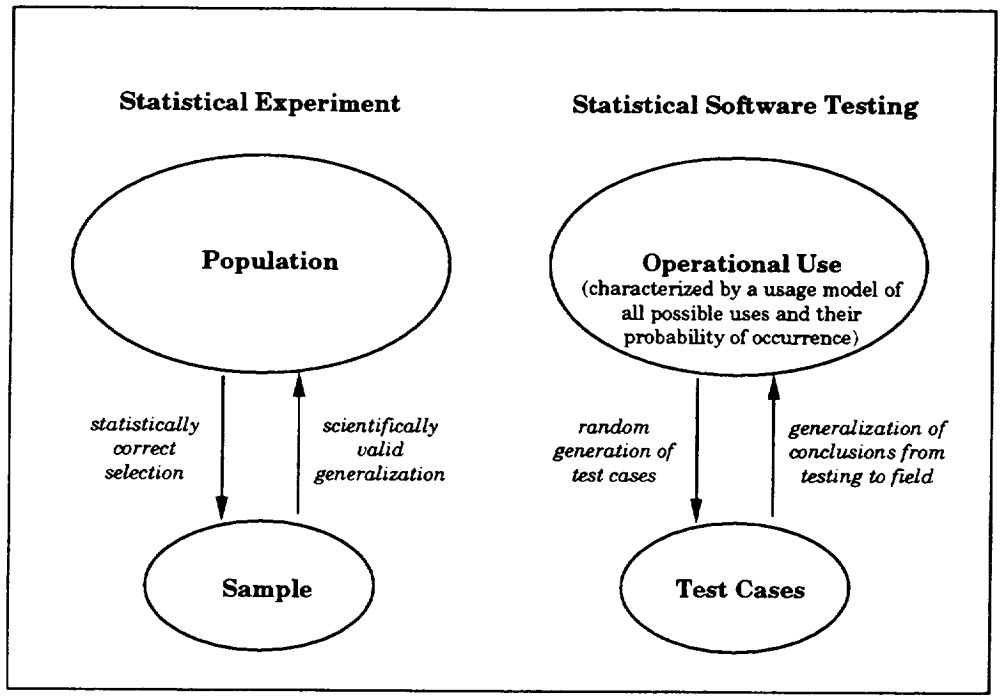


Figure 1. Software Testing as a Statistical Experiment

but astronomical number of possible usage scenarios.

The functional testing community measures test coverage in terms of function coverage. But testing every function is not the same as testing every combination of functions. And testing every combination of functions is not the same as testing every possible sequence of functions.

The structural testing community measures test coverage in terms of code coverage. But testing every line of code is not the same as testing every path. And testing every path is not the same as testing every possible sequence of paths.

There is really no question about whether all possible scenarios of use will be tested. They will not. The only questions are how the population of uses will be characterized, and how a subset of test cases will be drawn. A random sample of test cases from a properly characterized population, if applied to the software with proper experimental control, will allow scientific generalization of conclusions from testing to operational use. Any other set of test cases, no matter how thoughtfully constructed, will not.

Assumptions in a Statistical Experiment

In a statistical experiment, a well-defined procedure is performed under specified conditions, and produces one of two or more possible outcomes. Each performance of the procedure is called a "trial" of the experiment. The outcome data from successive trials of the experiment can be used to estimate the probability of each of the outcomes. Figure 2 portrays the general structure of a statistical experiment.

Several assumptions underlie the validity of inferences from a statistical

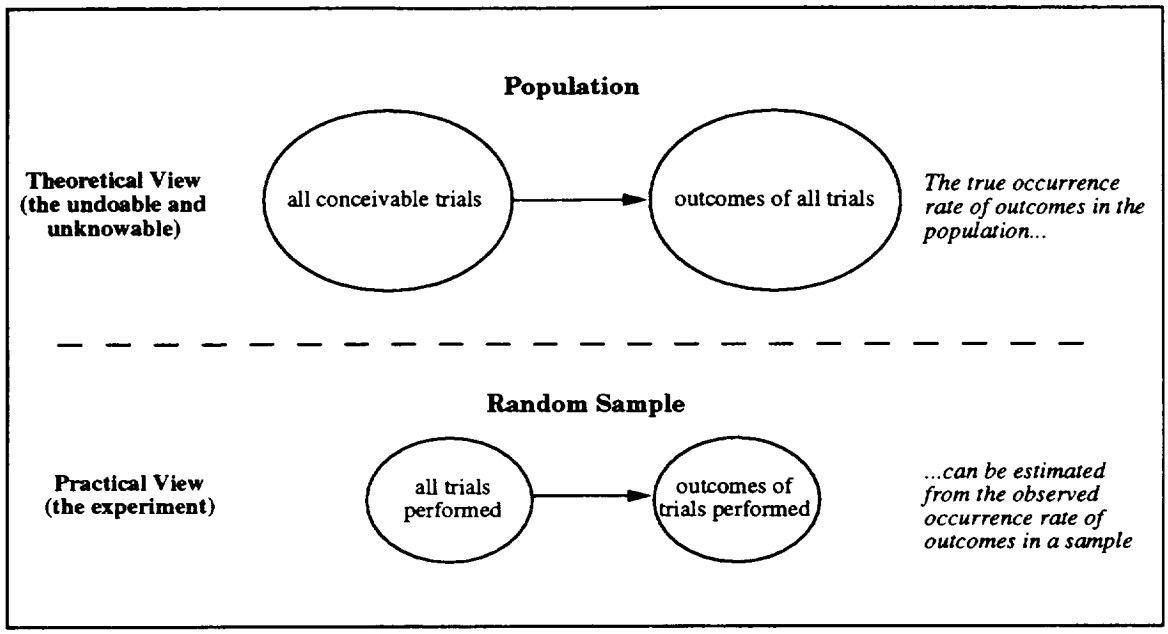


Figure 2. Structure of a Statistical Experiment

experiment, however. The assumptions are as follows.

(1) Each trial is performed under the same conditions.

(2) There is one outcome per trial.

(3) All outcomes are possible in each trial.

(4) Trials are independent.

The implications of these assumptions for the testing protocol must be understood. Proper experimental control in statistical certification testing is essential to the validity of the claims that result.

Meeting the Assumptions of a Statistical Experiment in Statistical Testing of Software

In statistical testing, a trial is ordinarily considered to be a test case. A test case generated from the usage model is a complete usage scenario beginning with some appropriate initial event (e.g., invocation, switchhook up, power on) and ending with some appropriate final event (e.g., termination, switchhook down, power off). Other definitions of a trial are possible, however, such as a single transaction or some other set of transactions. The certifier defines a trial in a manner that is appropriate for the application, and must do so in conjunction with the form of generalization the certifier wants to make about the population.

A statistical test case results in one outcome from a specified set of possible outcomes. The possible outcomes of a test case, for example, may be defined as {success, failure}. Under another design, the possible outcomes might be {success, cosmetic failure, serious nonblocking failure, blocking failure, crash}. Another design still may entail outcomes of {0 failures, 1 failure, 2 failures, ...10 or more

failures}. The challenges in experimental control grow with the complexity of the design since more granular judgments are required.

Regardless of the design of the statistical experiment---i.e., the definition of a trial and the specified set of possible outcomes---the foregoing assumptions about a statistical experiment must be met in the way trials are conducted and evaluated.

The implications of each of the foregoing assumptions is considered next. In the following discussion, a trial will be regarded as a test case that has been randomly generated from the usage model, and the possible outcomes of the test case will be regarded as success and failure.

Assumption 1: Each trial is performed under the same conditions.

What "conditions" are relevant to the conduct of a test case? The entities associated with a test case are, at a minimum,

- the software,
- the input,
- the system environment,
- the basis for evaluation of performance, and
- the tester (human or automated).

The software and the basis for evaluation of performance are entities that can be held constant; the input, the system environment and the tester are not amenable to complete control.

Software. The software used in testing will not change unless it is deliberately modified and recompiled. If it is changed in any way, *the statistical experiment must begin anew*. One may not amass data over several versions of software and treat them as a simple statistical experiment. Such data may be applied to reliability *growth* models that predict growth as a function of performance history and *changes* in the software, but may not be used to estimate parameters of a specific version of the software. Testing of each version of the software is a separate statistical experiment.

Input. To the extent that input varies with classes of usage---e.g., novice vs. expert, literary vs. mathematical subject matter, new vs. mature database---separate statistical experiments may be desirable. Otherwise, input (regardless of its origin in the system under test or another source) may be directly incorporated in the usage model structure and randomized via the usage probability distribution (e.g., percentage access of short and long files). The latter strategy effectively removes input from the set of conditions to which Assumption 1 applies by making it part of the trial rather than part of the background. This strategy also eliminates the distortion that could result from tester bias toward the shortest test cases, the "easiest" ones, the most subjectively interesting ones, etc.

System Environment. The system environment is perhaps the most illusive of the conditions to be controlled. Variability of background will be a feature of the real operational environment, however, so the experimental task is to simulate a test environment with variability that is typical of the actual environment. Concurrent activity, system load, interrupt schedules, etc., make for a constantly changing background. Again, key variables may be directly incorporated in the usage model structure and randomized via the usage probability distribution.

Basis for Evaluation. The basis for evaluation of a test case may be the specification, an independent "oracle," or both. It is not uncommon for a specification to change at any stage of

development, including testing. Consistent evaluation criteria must be applied within a testing experiment, however. Behavior that is regarded as correct (or incorrect) in one test case must be evaluated the same way in any other test case applied to that version of the software.

Tester. A given human tester may vary in the way he or she conducts and evaluates test cases, and the performance of any two testers may vary. Training, alertness, motivation, perception, and any number of other variables may affect the performance of human testers. While complete control over these factors is impossible, most of the variability can be eliminated through

- coordination of all test activities by a chief certification engineer,
- thorough tester training,
- explicit policies about test materials, session length, and data collection,
- documented guidance about issues on which the "test script" is not explicit,
- periodic "recalibration" of testers through paired performance of test cases with the chief certification engineer, and
- timely communication among testing team members with regard to observations and decisions that may affect test judgment.

Assumption 2: There is one outcome per trial.

If the specified set of outcomes (i.e., elementary events) is {success, failure}, then the outcome of a test case is either one success or one failure; it is not both, not two successes, *and not two (or more) failures.* A success is a test case in which the software performs correctly on all inputs in the test case; otherwise, the test case is counted as a failure.

In the strictest sense, then, counting of successes and failures is a simple matter. The number of successes plus the number of failures equals the number of test cases run.

The implication of one-outcome-per-trial is that a test case must be counted as a failure as soon as a failure on any input occurs. This is an unpopular policy, however, because a minor but unavoidable failure that occurs early in every test case will drive the measured reliability of the version to zero even though the software does most everything correctly.

An organization using statistical certification testing must develop a testing policy that accommodates the assumption of one-outcome-per-trial, yet allows testing to proceed in the presence of minor failures. Policy options may be politically difficult (e.g., counting every failure, with the result that status reports show declining reliability) or scientifically suspect (e.g., not counting recurrences of a failure, such that a correct fix and independence of failures must be assumed). Policies each have their advantages and disadvantages. A reasoned policy must be reached and used, however, so that the implications for the integrity of the statistical experiment are understood.

Assumption 3: All outcomes are possible in each trial.

All possible scenarios of usage must be candidates for selection in each trial, such that all the ways the software could succeed and all the ways it could fail are potentially observable.

In addition, this assumption implies that testing must not proceed in the presence of "blocking" failures. If an input is unreachable due to a blocking failure that

is "not counted" upon recurrence, then success or failure that would result from the input cannot be observed. The detection of a blocking failure is grounds for stopping the testing process and creating a new version of the software.

Assumption 4: Trials are independent.

Trials are independent if the outcome of one trial has absolutely no connection with the probability of the outcome of any subsequent trial. For software, trials (i.e., test cases) are independent if the success or failure of one test case has no bearing on the success or failure of any subsequent test case.

It may be argued that this assumption cannot be met since programs build up state information over successive runs. Since state data is the encapsulation of input history, the input in one trial may result in a change in state, and the new state may increase the probability of exposing a program defect---i.e., producing a failure---in a subsequent trial.

The only certain way to avoid dependency between failures is to fix each fault and corresponding state data after a failure, and restart testing with the new version of the software.

Alternatively, it may be possible to either avoid or randomize state data. Two types of state information exist: internal variables and external files. Internal variables exist for the duration of an execution. A test case that ends in termination, therefore, will not carry over internal state data to the next run. External files persist from one execution to the next, of course, but it is often not necessary to use them in sequential runs; their use may be randomized. Test cases of word processing software, for example, may randomly access one of a number of files (e.g., no file; short and long files; narrative and equation-filled files; etc.) according to an expected usage distribution.

Regression testing is a common violation of the assumption of independent trials. If previously used test cases are run on a new version of the software, they should not be counted as new trials.

Independence of trials in statistical software testing is defensible, but requires a deliberate strategy---either fixing failures as they are found, avoiding the carryover of state data, or randomizing state data according to an expected usage distribution.

The Slippery Slope Revisited

ACME Software improved control over its software testing process by establishing a documented testing protocol and training the project team. Things were different in the next project.

Before testing began, the testing team reviewed the specification, the test script, and other reference materials in detail. The group executed the first several test cases together, with each person taking a turn as the tester. The group reconvened at several points in testing for brief "recalibration" sessions. John and Mary's evaluations of test cases were much more consistent this time.

As prescribed by the protocol, Jane took a short break after each testing hour to review her annotations on the test script, update the chief certification engineer on her progress, and confer with other testers. She was much more alert and attentive to detail as a result.

Joe now understood that the product reliability claim would only be valid if engineering changes were made in a controlled way. Everyone understood that any deviation from the protocol was to be discussed by the team so that the impact on the integrity of the testing process could be determined.

Michael and Bill both now understood the "selection bias" that could result from picking and choosing among test cases, subdividing test cases, or otherwise altering the randomly generated sample of test cases. They now executed test cases in the order in which the test cases were generated.

Testers recorded all choices and observations as notes on the test script. Anyone with points of uncertainty---such as new hires---could later go over the specifics with the chief certification engineer to ensure the correctness of evaluations.

The Engineering Practice of Statistical Reliability Certification

If test team members are aware of the threats to experimental integrity, they can approach the innumerable decisions that must be made during testing with an eye toward preserving the validity of results. Recommendations for control over the testing process in the foregoing discussion are summarized here.

Test Preparation

- Define a test case as a usage scenario that is a longer period than the software can retain internal state data (e.g., invocation-to-termination).
- Randomize external state data via the usage probability distribution.
- Define the system environment(s), and either establish different usage models for different environments or sustain the conditions in a given environment throughout testing.
- Train test staff to ensure a common understanding of all test materials and policies, and monitor performance to prevent "drift."

Test Case Execution and Evaluation

- Hold the specification and independent oracle constant for each version of the software that is tested.
- Assign each test case one outcome from the specified set of possible outcomes.
- Run test cases in the order in which they are generated. Do not pick and choose.
- If previously used test cases are rerun on a new version, they should be performed for peace-of-mind only and not counted as new random trials.
- If a "blocking" failure occurs, stop and create a new version.
- If a failure occurs which could conceivably cause a subsequent failure, stop and create a new version.
- Schedule regular communication between test team members for discussion of matters that may affect test judgment.

Surviving the Compromises of Everyday Practice

A sound testing strategy may be compromised in practice if the rationale for the strategy is not well understood, is not embodied in a documented process, or is not practiced as documented. Indeed, "the difference between theory and practice in practice is greater than the difference between theory and practice in theory."

The threats to validity in certification testing can largely be controlled through understanding the assumptions in a statistical experiment, establishing explicit policies to meet them, and monitoring adherence to the policies in practice. Such experimental control is necessary to sound

footing on the slippery slope of applied science.

References

1. Currit, P. Allen, Michael Dyer, and Harlan D. Mills. "Certifying the Reliability of Software." *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, January 1986.

2. Mills, H. D., M. Dyer, and R. C. Linger. "Cleanroom Software Engineering." *IEEE Software*, September, 1987, pp. 19-24.

3. Mills, H. D. and J. H. Poore. "Bringing Software Under Statistical Quality Control." *Quality Progress*, November 1988.

4. Cobb, R. H. and H. D. Mills. "Engineering Software Under Statistical Quality Control." *IEEE Software*, November 1990.

5. Musa, J.D., A. Iannino, and K. Okumoto. Software Reliability: Measurement, Prediction, Application. McGraw-Hill: New York, 1987.

6. Musa, J.D. and William W. Everett. "Software-Reliability Engineering: Technology for the 1990s." *IEEE Software*, November 1990.

7. Musa, John D. "Operational Profiles in Software-Reliability Engineering." *IEEE Software*, March 1993.

8. Whittaker, James A. and J.H. Poore. "Markov Analysis of Software Specifications." *Transactions on Software Engineering and Methodology*, January 1993.

9. Walton, Gwendolyn H., J.H. Poore and Carmen J. Trammell. "Software Usage Modeling." *Software Practice and Experience*, to appear.

10. Poore, J. H., Harlan D. Mills, and David Mutchler. "Planning and Certifying Software System Reliability." *IEEE Software*, January 1993.

EXPERIMENTAL CONTROL IN SOFTWARE RELIABILITY CERTIFICATION

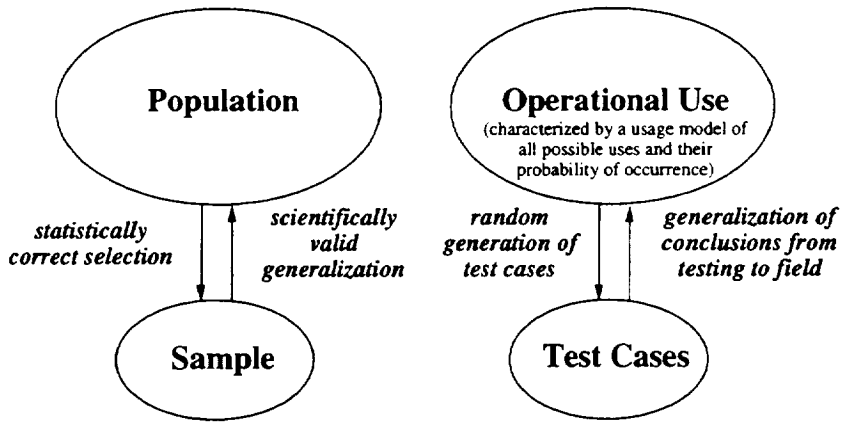
17th Annual Software Engineering Workshop
NASA/Goddard Space Flight Center

Carmen Trammell
University of Tennessee

UNIV. OF TENN. SOFTWARE ENGINEERING FOCUS: ADVANCES IN CLEANROOM PRACTICE

- Software Quality Research Laboratory
- Fifteen Cleanroom projects since 1988
- Student employees, high turnover
- Statistical testing (Mills and Musa) is used

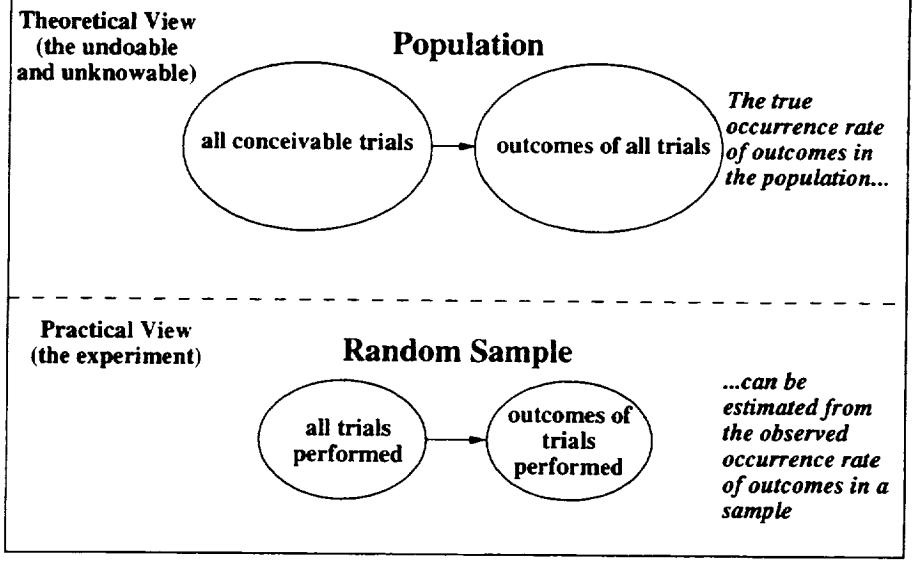
SOFTWARE TESTING AS A STATISTICAL EXPERIMENT



SYMPTOMS OF POOR TESTING PROCESS CONTROL

- **Delayed observation of failures**
- **Conflicting evaluations by testers**
- **Picking and choosing among test cases**
- **Unauthorized engineering changes**
- **Lack of communication by new testers**

STRUCTURE OF A STATISTICAL EXPERIMENT



ASSUMPTIONS IN A STATISTICAL EXPERIMENT

(1) **Each trial is performed under the same conditions.**

(2) **There is one outcome per trial.**

(3) **All outcomes are possible in each trial.**

(4) **Trials are independent.**

ASSUMPTION (1)
Each trial is performed under the same conditions.

- the software
- the input
- the system environment
- the basis for evaluation of performance
- the tester (human or automated)

ASSUMPTION (2)
There is one outcome per trial.

- the set of possible outcomes must be specified, e.g.,
  - { success, failure}
  - {no failures, minor failure, serious failure, crash}
  - {0 failures, 1 failure, ...n or more failures}
- recurrences of failures: to count or not to count?
  - counting recurrences results in reports of declining reliability
  - not counting recurrences requires judgments about independence of failures

ASSUMPTION (3)
All outcomes are possible in each trial.

- **all usage scenarios must be candidates for selection in each trial**
- **the outcome of each scenario must be observable... testing cannot proceed in the presence of blocking failures**

ASSUMPTION (4)
Trials are independent.

- **test cases must exceed the retention of internal state data**
- **external state data should be randomized**
- **regression tests must not be counted as new random trials**

LESSONS LEARNED ARE EMBODIED IN THE CURRENT PROTOCOL

Test Preparation

- Define a test case as a usage scenario that is a longer period than the software can retain internal state data (e.g., invocation-to-termination).
- Randomize external state data via the usage probability distribution.
- Define the system environment(s), and either establish different usage models for different environments or sustain the conditions in a given environment throughout testing.
- Train test staff to ensure a common understanding of all test materials and policies, and monitor performance to prevent "drift."

LESSONS LEARNED ARE EMBODIED IN THE CURRENT PROTOCOL

Test Case Execution and Evaluation

- Run test cases in the order in which they are generated.
- Hold the specification and oracle constant for each version
- Assign each test case one outcome from the set of possible outcomes.
- If test cases are rerun, do not count them as new trials.
- If a "blocking" failure occurs, stop and create a new version. If an observed failure could cause a subsequent failure, stop and create a new version.
- Schedule regular communication for discussion of matters that may affect test judgment.

GENERALIZED IMPLEMENTATION OF SOFTWARE SAFETY POLICIES†

John C. Knight
knight@virginia.edu

Kevin G. Wika
wika@virginia.edu

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

(804) 924-7605

An Abstract Submitted to:

Nineteenth Annual Software Engineering Workshop
Software Engineering Laboratory
Goddard Space Flight Center
Greenbelt, MD

†. Supported in part by the National Science Foundation under grant number CCR-9213427 and in part by NASA under grant number NAG1-1123-FDP.

Introduction

As part of a research program in the engineering of software for safety-critical systems, we are performing two case studies. The first case study, which is well underway, is a safety-critical medical application. The second, which is just starting, is a digital control system for a nuclear research reactor. Our goal is to use these case studies to permit us to obtain a better understanding of the issues facing developers of safety-critical systems, and to provide a vehicle for the assessment of research ideas.

The case studies are not based on the analysis of existing software development by others. Instead, we are attempting to create software for new and novel systems in a process that ultimately will involve all phases of the software lifecycle. In this abstract, we summarize our results to date in a small part of this project, namely the determination and classification of policies related to software safety that must be enforced to ensure safe operation. We hypothesize that this classification will permit a general approach to the implementation of a policy enforcement mechanism.

The Problem

The functionality demanded by modern applications, including safety-critical applications, frequently leads to software that is very large and complex. Functionality requirements have increased because of the many benefits of computer-based control and the availability of inexpensive yet powerful computing hardware. Hardware performance limits that formerly restricted software complexity are rarely reached because of the remarkable hardware performance now available.

Unfortunately, significant software defects tend to remain in such systems after deployment despite extensive effort on the part of the developers [2,6]. Building these systems to perform as desired is very difficult for a number of reasons. Even the best software development processes cannot ensure that faults are avoided completely during development. Similarly, fault detection techniques are imperfect. Research has shown, for example, that testing as an approach to verification cannot demonstrate sufficient levels of dependability because of the sheer number of tests that are required [1].

Even building very small, simple software systems that achieve the extreme dependability necessary for safety-critical applications has proven to be very challenging. Formal techniques have made substantial progress and have been applied to real systems in a number of cases, but their application to large, complex systems remains mostly impractical. The complexity of large systems involving characteristics such as real-time operation and distributed processing is likely to preclude any significant assurance that the systems meet desired dependability goals if traditional techniques are used in traditional ways.

A central question that arises is how to deal with a software system that is on the one hand safety critical and on the other hand large and complex, i.e., so large and complex as to preclude a complete attack on the problem of showing dependability using even the best available techniques. We outline an approach to this problem that we are pursuing in the next section.

Technical Approach

An approach that has been tried in many safety-critical systems is to isolate the problem of ensuring safe operation so that a small part of the software, often termed a *kernel*, is responsible. This is the approach we are following but we are attempting to develop a general, comprehensive approach to the problem by exploiting an analogy with security kernels.

Security kernels are used to enforce access-control policies in classified information systems. The idea of trying to exploit this technique to implement safety rather than security, i.e., the concept of a more general *safety kernel*, was proposed by Rushby [5,7], among others. The idea that Rushby suggested is different from other architectures described as safety kernels because certain essential safety policies are enforced regardless of the actions of the application software. This is in direct analogy with security kernels that enforce access control with a similar degree of generality. Other safety-kernel architectures that have been developed tend to provide a set of services that enforce required safety policies, *if used appropriately by the application*. This is a critical distinction.

The safety-kernel idea is of value if it is able to enforce a suitably large subset of the required safety policies. A major benefit would be gained if this safety-kernel approach could be implemented in a reusable manner, i.e., in such a way that the same safety kernel implementation could be used in a variety of applications. To evaluate the safety-kernel idea, assess its utility, and try to get some insight into generality that might be possible in an implementation we have analyzed the two case studies at our disposal. We report our results in the next section.

Empirical Results

We began this study by identifying the safety policies required by each application. We then examined the two sets to ascertain whether general classes of policies existed and whether the policies were similar in the two cases after application-dependent parameters were removed. We begin this section by summarizing the important details of the two applications and list examples of the safety policies they require. We then discuss the resulting structure of the policies and its implication on implementation and generality.

Magnetic Stereotaxis System

The first case study that we are engaged in is the *Magnetic Stereotaxis System* (MSS). This is an investigational device for performing human neurosurgery being developed in a joint effort between the Department of Physics at the University of Virginia and the Department of Neurosurgery at the University of Iowa [3,4]. The device operates by manipulating a small permanent magnet (known as a "seed") within the brain using an externally applied magnetic field. The patient is positioned at the center of six superconducting electromagnets. Under the direction of the computer, power supplies and current controllers regulate the electric current in the electromagnets thereby producing the magnetic field that acts on the seed. Along each axis perpendicular to the patient's body, an X-Ray source and camera produce fluoroscopic images for tracking the seed. By varying the magnitude and gradient of the external field, the seed can be moved along a non-linear path and positioned at a site requiring therapy, e.g., a tumor.

When the MSS is in operation, there are a large number of events that could lead to patient injury. The complete set is determined by a hazard analysis including the use of techniques such as system fault-tree analysis. Events that could lead to patient injury include failure of current controllers, X-Ray overdose, incorrect calculation of currents for a seed movement, and failure to respond promptly to an increase in seed velocity. Each of these could be the result of numerous different faults, and, in fact, the software could either initiate or prevent many of these failures. Such failures can be prevented irrespective of their cause and irrespective of the state of the equipment if safety policies such as the following (stated here informally) are enforced:

- If the seed moves faster than 2.0 mm/sec, the coil currents must be set to zero.
- If the vision system cannot locate the seed while it is being moved, the coil currents must be set to zero.
- The currents must be within 5.0 A of the value predicted by the coil control model.
- The current requested of a controller must be in the range -100 A to +100 A.
- Before moving the seed, a reversal check must be executed to ensure that the requested currents provide the desired direction within 5 degrees.
- An X-Ray device must be "off" for 0.2 sec before an "on" command is executed.
- The total X-Ray dose during an operation must be less than 100 millirem.

In the MSS system, a total of 42 safety policies have been identified. They are all similar in complexity and breadth to these examples.

University of Virginia Reactor

The target of the second case study is the nuclear research reactor currently operated by the University of Virginia. It is a 2 MW thermal, concrete-walled pool reactor. It was originally constructed in 1959 as a 1 MW system, and it was upgraded to 2 MW in 1973. Though only a research reactor rather than a power reactor, the issues raised are significant and can be related easily to the problems faced by full-scale reactor systems.

The system operates using 20 to 25 plate-type fuel assemblies placed on a rectangular grid plate. There are three scramable control rods, and one non-scramable regulating rod that can be put in automatic mode. The primary process variables that are measured are: 1) Gross output, by movable fission chamber; 2) Neutron flux, by ion chamber; 3) Start-up neutron flux and period, by BF_3 counter; 4) Core inlet and outlet temperatures, by thermocouples; 5) Primary system flow, by pressure gauge; 6) Control and regulating rod positions, by potentiometer; 7) Gross gamma-ray dose, by ion chamber; 8) Various limit set switches to monitor pool level, etc.

As with the MSS, there are a large number of events that could lead to a reactor accident with the potential to cause extensive damage. Some examples of events that could result in hazards include uncontrolled withdrawal of the reactor control rods, loss of water in the reactor pool, failure of a coolant pump, and high radiation levels outside of the reactor pool. Again as with the MSS, such failures can be prevented irrespective of their cause if safety policies such as the following (again stated informally) are enforced:

- The control rods must not be withdrawn at a rate faster than 1.5 mm/sec.
- When control parameters are adjusted, the state of the reactor must respond to reflect the control settings.
- The position of the regulating rod must be adjusted at least once per second based on the power output of the reactor.

If any of the following conditions is true, the control rods must be scrammed:

- A safety channel indicates a power greater than 125% of maximum power.
- The flow in the primary cooling system is below 3,400 liters/min (900 gals/min).
- The reactor inlet water temperature exceeds 105° F.
- The pool level falls below 19 ft. 3 1/4 in.
- The radiation at the reactor face exceeds 2 mR/hr.

A preliminary identification of the safety policies in this application revealed a total of 43 safety policies. As detailed requirements analysis proceeds, this number is likely to rise.

Once the initial sets of safety policies had been identified for the two applications, we focused on identifying common characteristics both within and between the two applications that might permit a logical organization of the two sets of safety policies. We were seeking insight into what might be a general case in order to permit us to begin consideration of a general, reusable, safety kernel. After examining a variety of possibilities, the characteristic that permitted the most complete and systematic classification of the policies was based on the origin and derivation of the safety policies.

Safety policies such as the examples above result from the system safety analysis, and specify safety requirements that must be met by the various system components. In a system safety analysis, a set of mishaps are identified along with hazards that could cause the particular mishap. Each hazard is in turn placed at the root of a system fault tree and the failure conditions that could result in the hazard are analyzed. The exact form of a fault tree depends on the hazard being considered and the details of the particular application. However, we have identified a canonical fault-tree pattern for computer-controlled devices, and we have been able to classify failure conditions according to their location and purpose with respect to the canonical fault tree. We are thus able to classify safety policies according to which type of condition the policy addresses, and this has yielded the following *general* categories of policies:

System operation	Device operation
Device failure	Device input from software
Software error	Failure response
Software input	Operator input to the software
Sensor input	Configuration or application data
Operator error	Operator information

Subsequent re-analysis of the two complete sets of safety policies from our two case studies has shown that the various policies fit into the taxonomy very well. Thus, although the applications are very different, their requirements for safe operation are remarkably similar in basic form and differ to a large extent only in application-specific detail. Though important, these details can be viewed as parameters that can be used to tailor a general implementation strategy, i.e., a general-purpose safety kernel operating in a manner analogous to a security kernel.

A safety kernel prototype is being developed that will enforce policies from the first six categories of policies identified above. These are policies that originate near the top of the canonical fault tree discussed above. They have been selected for enforcement because they are most closely associated with the operation of the application devices. It is the devices that actually cause a mishap, so it makes sense to enforce safety policies that are directly related to devices. Policies from the other classes were omitted because the benefits were not as great and the pragmatic issues of quality assurance, cost, and functional performance would be adversely affected by enforcing policies from these classes.

Conclusions

Based on the two systems we have been studying, it appears to be the case that a great deal of structure exists in the safety policies that have to be enforced. Given this situation, there seems to be a strong possibility that a reusable safety kernel operating independently of the application in a manner analogous to the operation of a security kernel can be built. Such a kernel would permit execution-time enforcement of selected safety policies for systems too complex to verify by traditional means.

Acknowledgments

This work was supported in part by the National Science Foundation under grant number CCR-9213427, and in part by NASA under grant number NAG1-1123-FDP.

References

1. Butler, R. W. and G. B. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," *IEEE Transactions on Software Engineering*, Vol. 19-1, pp. 3-12, Jan. 1993.

2. Garman, J. R., "The Bug Heard 'Round the World," ACM Software Engineering Notes Vol. 6-5, pp. 3-10, October 1981.

3. Gillies, G. T. et al, "Magnetic Manipulation Instrumentation for Medical Physics Research," *Review of Scientific Instruments*, Vol. 65-3, pp. 533 - 562, March 1994.

4. Grady, M. S. et al, "Preliminary Experimental Investigation of *in vivo* Magnetic Manipulation: Results and Potential Application in Hyperthermia," *Medical Physics* Vol. 16-2, pp. 263 - 272, Mar/Apr. 1989.

5. Leveson, N. G., T. J. Shimeall, J. L. Stolzy and J. C. Thomas, "Design for Safe Software," in *Proceedings AIAA Space Sciences Meeting*, Reno, Nevada, 1983.

6. Neumann, P.G., Editor, "Risks to the Public". Software Engineering Notes.

7. Rushby J., "Kernels for Safety?," in *Safe and Secure Computing Systems*, T. Anderson Ed., Blackwell Scientific Publications, 1989, pp. 210-220.

GENERALIZED IMPLEMENTATION OF SOFTWARE SAFETY POLICIES*

John C. Knight Kevin G. Wika

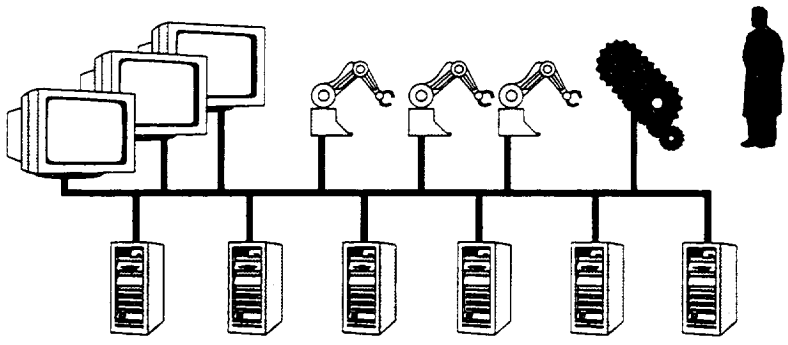
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

* Work sponsored in part by NASA, the NSF, the NRC & Motorola Inc

UVA
Department of Computer Science

THE PROBLEM WE FACE

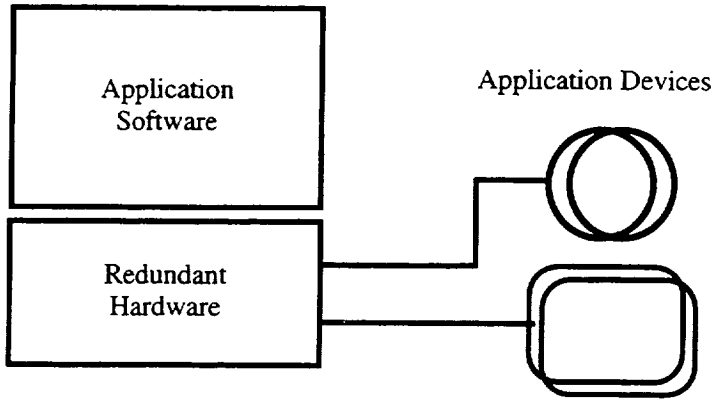
- Software Is *Large And Complex* In Many Safety-critical Systems:



- Huge Subsystems, E.g. System Services, Windowing, The Application, Etc.
- How Do We Build Safety-critical Software That Is:
  - Dependable?
  - Cost-effective?

UVA
Department of Computer Science

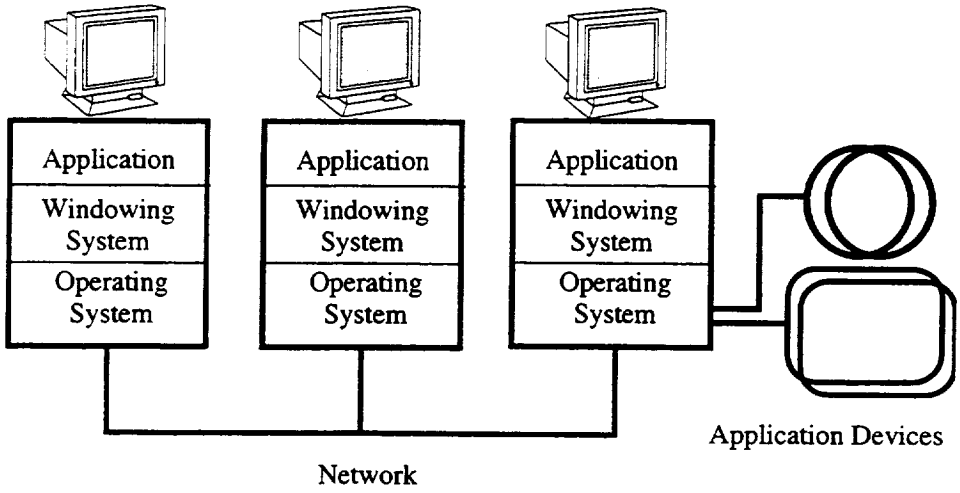
NAIVE SYSTEM ARCHITECTURE



- Keep It Simple, S*****

UVA
Department of Computer Science

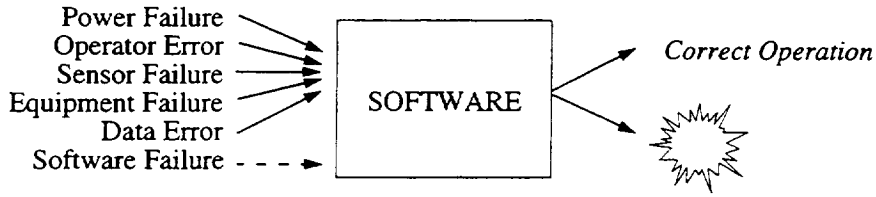
REALISTIC APPLICATION ARCHITECTURE



- Diverse Hardware, Network, High-performance Displays
- Extensive, Diverse And Unreliable Software, Perhaps Off-the-shelf

UVA
Department of Computer Science

SAFETY REQUIREMENTS AND SAFETY POLICIES



- Safety Requirements Can Often Be Expressed As *Safety Policies*
- Safety Policies — Policies That "Software" Must Enforce To Avoid Hazard
- Policies Such As The Following (From A Nuclear Reactor):

  If the flow in the primary cooling system is below 3,400 liters/ minute, a scram must occur.

  The source range must be indicating at least 2 counts/second before a safety rod can be withdrawn.

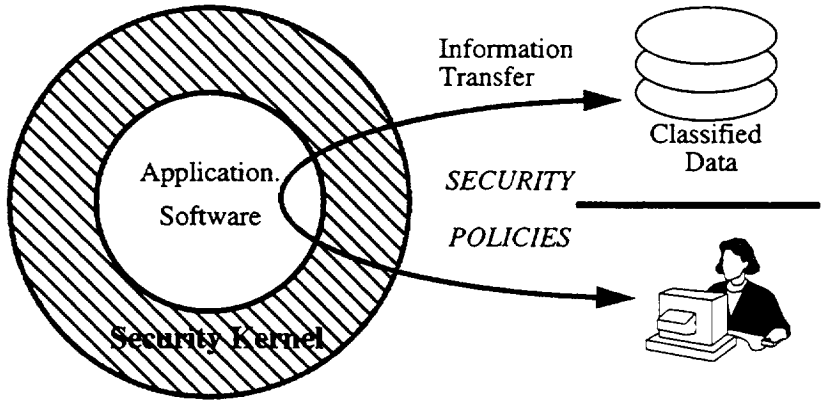
How Do We Ensure Enforcement Of Safety Policies?

UVA
Department of Computer Science

NASA GSFC/SEL - 94 - Slide 5 (© John C. Knight 1994)

SECURITY KERNEL CONCEPT

- Concept Is That Security Kernel Controls Access To All Information
- Kernel Enforces A Set Of *Security* Policies Irrespective Of Application Software's Actions:



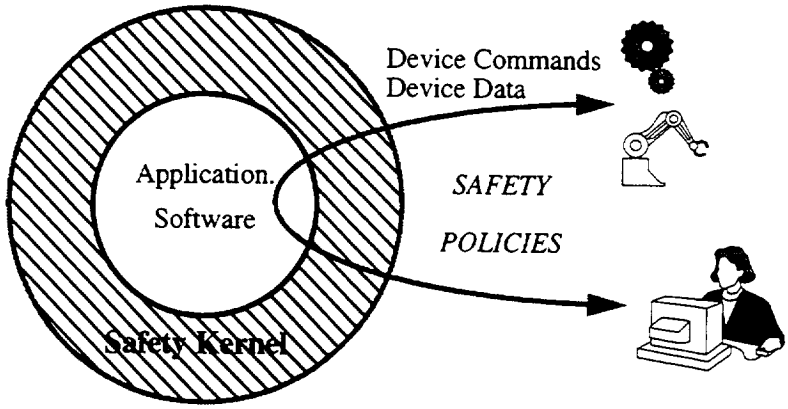
- Might A Similar Approach Work For Safety (Rushby, 1989)?

UVA
Department of Computer Science

NASA GSFC/SEL - 94 - Slide 6 (© John C. Knight 1994)

SAFETY KERNEL CONCEPT

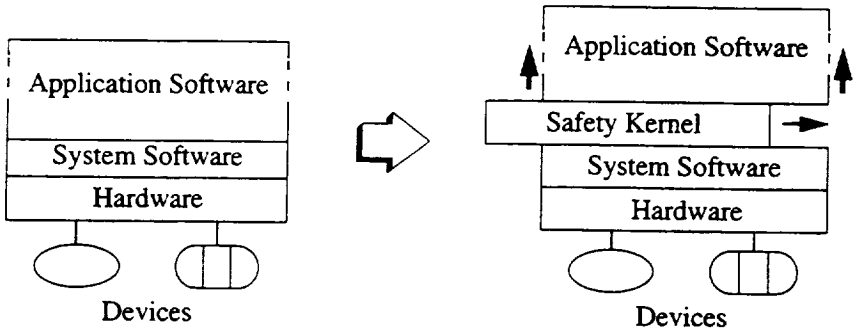
- Concept Is That Safety Kernel Controls Access To All Devices
- Kernel Enforces A Set Of *Safety* Policies Irrespective Of Application Software's Actions:



- A Similar Approach Appears To Work For Safety

UVA
Department of Computer Science

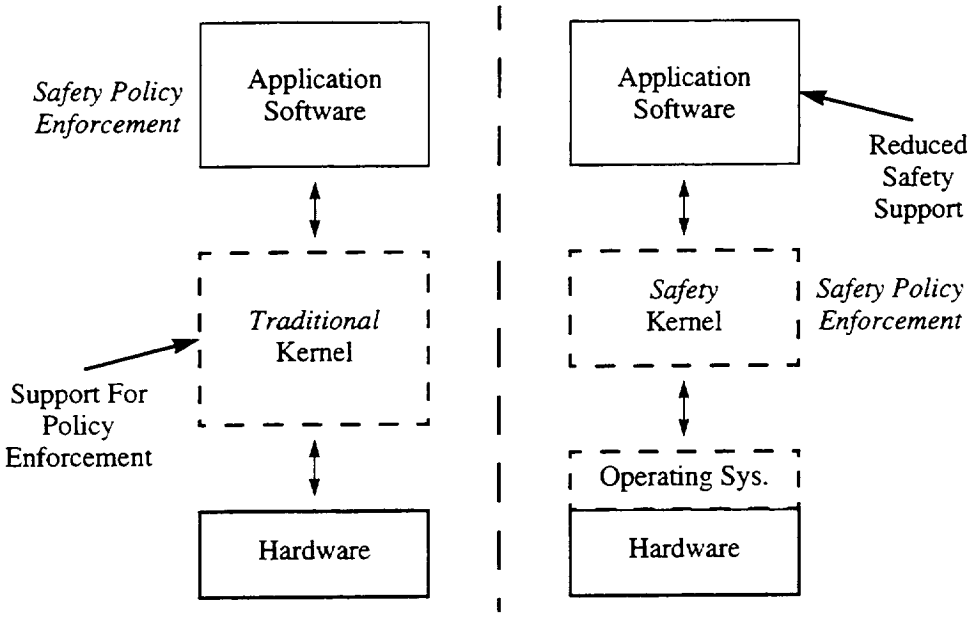
SAFETY KERNEL



- Policy Enforcement Given To Smallest, Simplest Kernel Possible
- Kernel Controls Access To All Devices Thereby Controlling Effect Of Software
- Policy Enforcement:
  - Certain Important Policies Entirely Enforced By Kernel
  - Enforcement Support For Other Policies

UVA
Department of Computer Science

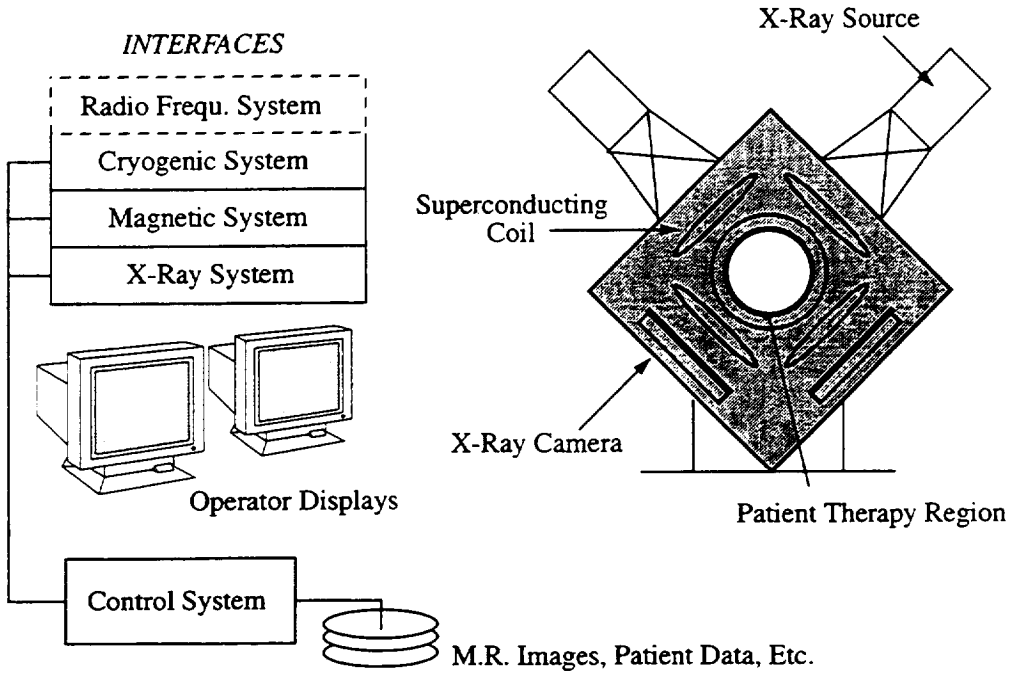
"TRADITIONAL" KERNEL vs. SAFETY KERNEL



UVA
Department of Computer Science

NASA GSFC/SEL - 94 - Slide 9 (© John C. Knight 1994)

CASE STUDY - MAGNETIC STEREOTAXIS SYSTEM



UVA
Department of Computer Science

NASA GSFC/SEL - 94 - Slide 10 (© John C. Knight 1994)

SOME OF THE MSS SAFETY POLICIES

If the seed moves faster than 2.0 mm/sec., the coil currents must be set to zero.

The coil currents must be within 5.0 amps of the predicted value.

The coil current requested by the application must be within the range -100 amps to 100 amps.

An X-ray source must be "off" for 0.2 seconds before an "on" command is executed.

The total X-ray dose during an operation must be less than 100 millirem.

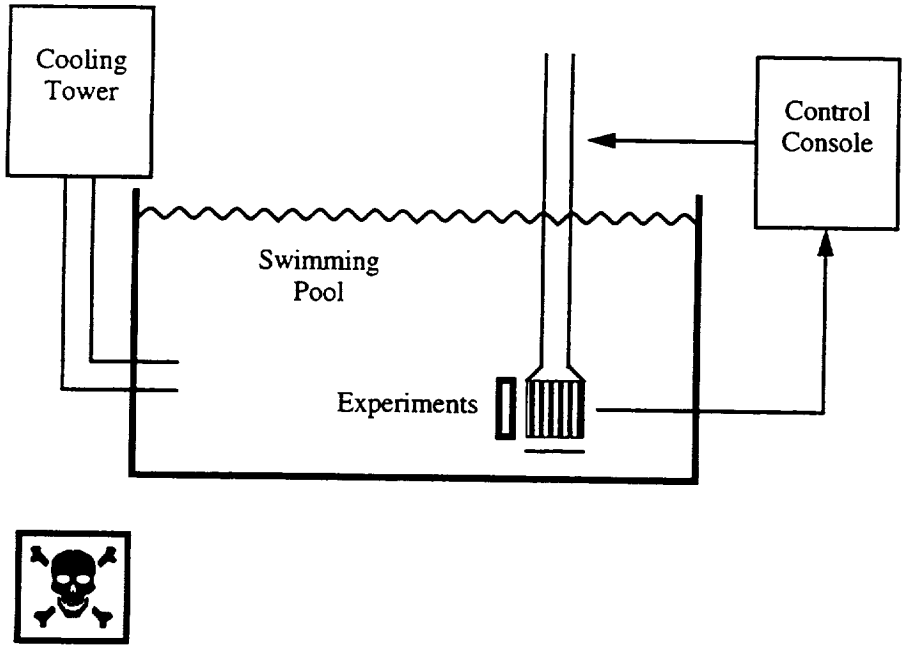
Before moving the seed, a reversal check must be executed on the requested currents to compare the predicted force with the desired force.

And so on.....

UVA
Department of Computer Science

NASA GSFC/SEL - 94 - Slide 11 (© John C. Knight 1994)

CASE STUDY - UVA RESEARCH REACTOR



UVA
Department of Computer Science

NASA GSFC/SEL - 94 - Slide 12 (© John C. Knight 1994)

SOME OF THE REACTOR SAFETY POLICIES

The control rods must not be withdrawn at a rate faster than 1.5 mm/sec.

The position of the regulating control rod must be adjusted at least once per second based on the power of the reactor.

The control rods must be scrammed if a safety channel indicates a power level greater than 125% of the authorized maximum.

The control rods must be scrammed if the pool water level falls below 19' 3.25".

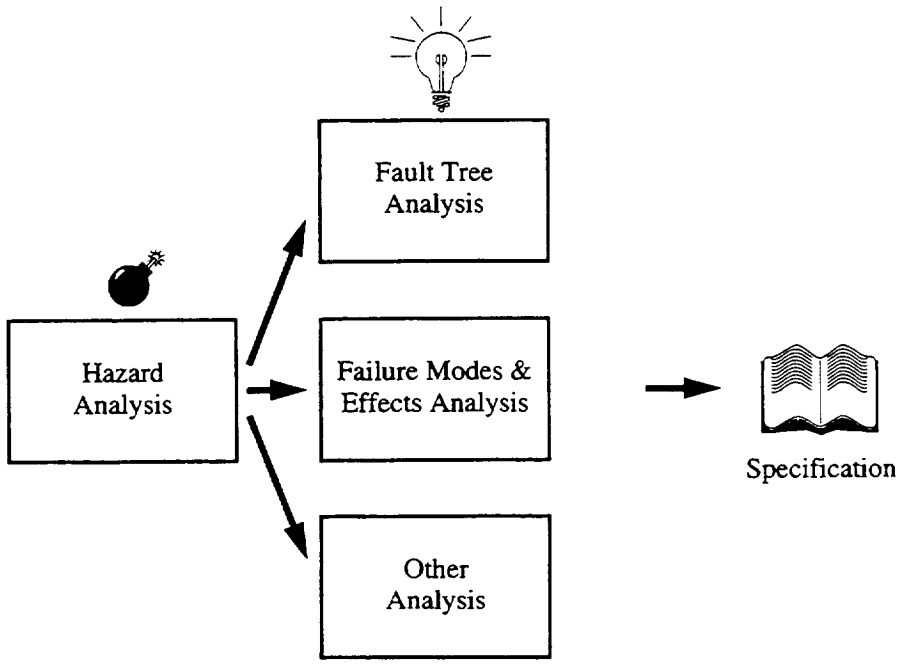
The control rods must be scrammed if the inlet water temperature exceeds 105° F

And so on.....

UVA
Department of Computer Science

NASA GSFC/SEL - 94 - Slide 13 (© John C. Knight 1994)

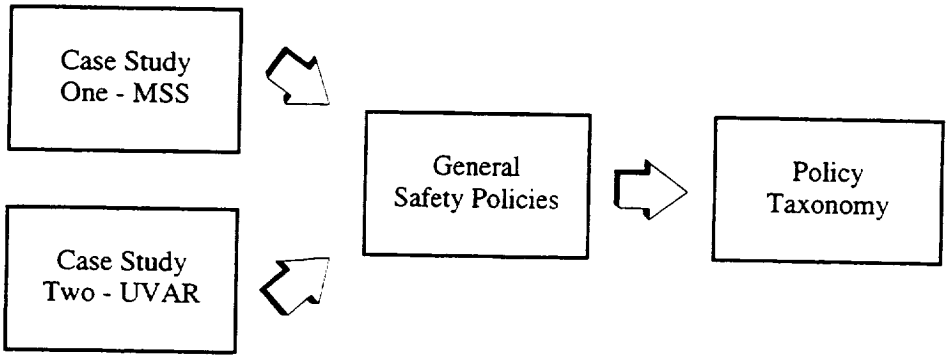
SAFETY POLICY DEVELOPMENT



UVA
Department of Computer Science

NASA GSFC/SEL - 94 - Slide 14 (© John C. Knight 1994)

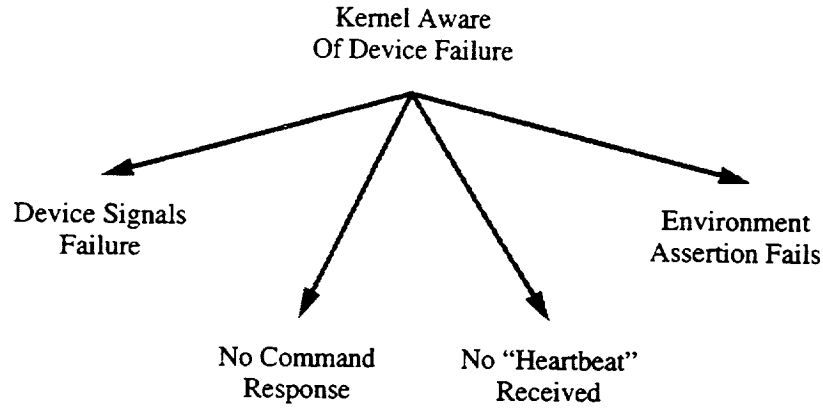
TAXONOMY - GENERAL SAFETY POLICIES



NASA GSFC/SEL - 94 - Slide 15 (© John C. Knight 1994)

UVA
Department of Computer Science

EXAMPLE - DEVICE FAILURE DETECTION



- Captures Essential Content Of "All" Device Failure Detection Policies
- Parameterized Implementation In Reusable Safety Kernel
- Follows From Generalized System Fault Trees

NASA GSFC/SEL - 94 - Slide 16 (© John C. Knight 1994)

UVA
Department of Computer Science

CONCLUSIONS

- Systems Are Getting Very Complex:
  - Simple Software Structures Unrealistic
  - Users Need "Gee Whiz" Features
- No Hope Of Verifying Everything Required:
  - Far Too Much Software
  - Off-the-shelf (Untrusted) Software Might Be Included
- Safety Kernel Analogy With Security Kernel Seems Viable
- Safety Policies Examined From Two Very Different Applications:
  - Taxonomy Suggested By Similarity Of Policies
  - General System Fault Tree Patterns
- General-purpose Safety Kernel For Variety Of Applications:
  - Seems Feasible
  - Significant Technical Issues In Implementation
  - Prototype Kernel Being Developed - Will Be Applied To Two Case Studies

UVA
Department of Computer Science

Session 6: Measurement

A Quantitative Comparison of Corrective and Perfective Maintenance
Joel Henry, East Tennessee State University

Does Software Design Complexity Affect Maintenance Effort?
Christopher Lott, University of Kaiserslautern

Profile of Software Engineering Within NASA
Craig Sinclair, Science Applications International Corporation

A Quantitative Comparison of Corrective and Perfective Maintenance

Joel Henry and James Cain
Department of Computer and Information Sciences
East Tennessee State University

Summary: *This paper presents a quantitative comparison of corrective and perfective software maintenance activities. The comparison utilizes basic data collected throughout the maintenance process. The data collected are extensive and allow the impact of both types of maintenance to be quantitatively evaluated and compared. Basic statistical techniques test relationships between and among process and product data. The results show interesting similarities and important differences in both process and product characteristics.*

1. INTRODUCTION

Most large software systems have long lifetimes during which the software undergoes significant change. Software maintenance is defined as the set of activities performed to change a software product after the software product is delivered to the customer (Pressman, 1987). These activities, plus the tools and methods used to maintain software are referred to as the maintenance process. Changes to existing software include adding functionality to the software, correcting defects discovered in the software system, adapting the software to changes in the environment, and changing the software to support future maintenance or operation. The variety of changes made to software and the fact that most maintenance personnel were not involved in the development effort add significantly to the difficulties encountered while performing software maintenance.

In recent years the software process (including both development and maintenance) has received a great deal of attention (Humphrey et al., 1987) (Humphrey, 1989) (Bollinger et al., 1991) because the process used to develop and maintain software significantly impacts the cost, quality and timeliness of software products. The impact is so significant that software process improvement is seen as the most important approach to software product improvement (Humphrey, 1989).

While software development typically refers to the creation of new software, software maintenance is performed for a variety of reasons. The four types of software maintenance activities are:

1. Corrective - changes made to correct defects in software
2. Adaptive - changes needed to adapt existing software to a changing environment
3. Perfective - enhancements to software which provide additional functionality or modify existing functionality
4. Preventative - changes which improve future maintainability, reliability or support future enhancements

The tasks employed during maintenance are very similar to those applied during development: specify, design, code, and test. Thus, the first step in maintenance is to obtain a written specification of the functionality to be added. The written specification is given by changes and additions to the documentation specifying the functionality of the existing software. In principle the written specification is given completely and is never changed during the ensuing maintenance effort. In practice, however, these specifications are corrected and refined throughout the maintenance process. The changing of functional specifications during maintenance and development is referred to as requirements volatility. Requirements volatility has been cited as the leading problem in a field study of software managers (Thayer et al., 1982). Changing requirements adversely affects the design, coding and testing of software. An acute need exists to quantitatively assess the maintenance process and the impact of requirements volatility on both the maintenance process and the software product.

The focus of this paper is a comparison of corrective and perfective maintenance activities driven by changes to the specification documents of existing software. This comparison attempts to answer three general questions:

1. What similarities exist between corrective and perfective maintenance characteristics?
2. What differences exist between corrective and perfective maintenance characteristics?
3. What do these similarities and differences suggest about the nature of perfective and corrective maintenance?

This paper describes a portion of the results of a three-year study conducted at a large commercial software organization to assess the maintenance process and the impact of requirements volatility on the

maintenance process. The portion of the assessment described here illustrates similarities and differences between corrective maintenance and perfective maintenance.

While this paper describes the results obtained within a single large organization, the results may be used by other organizations. These results indicate organizations should manage corrective and perfective maintenance differently.

The remainder of this paper is divided into two sections. Section 2 presents analysis results in five distinct areas. Section 3 outlines conclusions and the direction of future work.

2. ASSESSMENT RESULTS

Five significant results are described in the following subsections. Each subsection discusses the focus of the analysis, the data used in the analysis, and the statistical results. A maximum P-value of 0.05 and the minimum R^2 value of 0.75 were established as criteria for asserting relationships existed. This maximum P-value represents a 5% chance of mistakenly assuming a relationship exists. The minimum R^2 can be viewed as explaining 75%of the variability of the predicted variable.

2.1 CORRECTIVE AND PERFECTIVE SIMILARITIES

2.1.1 PRODUCTIVITY

Software maintenance productivity is of particular interest when examining corrective and perfective maintenance activities. We compared the productivity of both types of activities using corrective and perfective activity measures. Productivity is measured in SLOCs (source lines of code) per day and changed SLOCs per day.

Our initial examination showed only a 5.6% difference in productivity, with perfective maintenance being slightly more productive. Requirements volatility, tracked by specification changes occurring during design, code, and test, showed only an 8.5% difference. Again, perfective maintenance productivity was slightly higher.

The Mann-Whitney test, which statistically tests the differences in the sample means, was applied in order to test the hypothesis that corrective and perfective maintenance items are similar. The Mann-

Whitney test produced a P-value of 0.9833 which is not less than the previously established maximum P-value of 0.05. The P-value of 0.9833 supports acceptance of the hypothesis that the productivities of corrective items and perfective items are not statistically different.

2.1.2 SIGNIFICANT IMPACT ON PRODUCTIVITY

The previous section strongly supports the assertion that productivity of corrective maintenance and perfective maintenance is not statistically different. However, we noted differences between corrective and perfective product impact, as shown in Table 1. Perfective maintenance impact is greater in terms of SLOCs changed and modules changed than corrective maintenance. SLOCs changed per module appear similar. We investigated which of these three factors influenced productivity the most. We found the most significant factor influencing productiviy is SLOCs per module.

	CORRECTIVE TOTAL SLOCS	CORRECTIVE MODULES CHGD	PERFECTIVE TOTAL SLOCS	PERFECTIVE MODULES CHGD
MEAN	33.1905	1.7541	150.8511	3.0459
STD DEV	55.3804	1.7763	517.6439	3.6676
MEDIAN	10.5000	1.0000	23.5000	2.0000

Table 1. Basic Statistics for Corrective and Perfective Characteristics

	CORRECTIVE PRODUCTIVITY	PERFECTIVE PRODUCTIVITY
SLOCS per MODULE	0.951	0.788

Table 2. Linear Correlations of Product Impact vs Productivity

Table 2 gives the linear correlations for productivity with SLOCs changed per module for both corrective and perfective maintenance. The linear correlations for corrective and perfective are both

above the 0.75 threshold. These correlations suggest corrective and perfective maintenance productivity are significantly influenced by the distribution of change across modules.

4.2 CORRECTIVE AND PERFECTIVE DIFFERENCES

4.2.1 PRODUCT IMPACT

This section describes the significant differences between corrective and perfective maintenance. The characteristics compared include size of the change (measured in SLOCS), implementation effort(measured in person days), and distribution of change(measured in modules changed). We again applied the Mann-Whitney test, testing the hypothesis that the size and distribution of change are similar for both types of maintenance.

The results of the Mann-Whitney tests for modules changed and size of change produced P-values of 0.0170 and 0.0012, both significantly less than the maximum P-value of 0.05. These P-values support rejection of the hypotheses that modules changed and size for corrective maintenance are similar to corresponding measures for perfective maintenance. Thus, there are more lines of code, and are more modules changed for perfective maintenance than for corrective maintenance.

4.2.2 PRODUCT IMPACT ON QUALITY

Thus far, analysis has focused on corrective and perfective characteristics within the maintenance process, prior to delivery to the customer. This subsection examines the product impact of corrective and perfective maintenance activities on software quality.

We obtained defect data gathered prior to delivery and following product delivery. These defects have different levels of severity and are of great importance to the customer. Defect data (pre-delivery and post-delivery) and product impact data were analyzed using rank correlations to determine, statistically, their relationships.

	CORRECTIVE CHANGED SLOCs	PERFECTIVE CHANGED SLOCs
PRE-DELIVERY DEFECTS	0.3214	0.9702
POST-DELIVERY DEFECTS	0.2143	0.8884

Table 3. Rank Correlations of Defects And Changed SLOCs

Table 3 presents the rank correlations between the corrective and perfective changed SLOCs and the number of pre-delivery and post-delivery defects detected. The number of perfective changed SLOCs has a much stronger positive correlation to both types of defects than the number of corrective changed SLOCs. These results suggest that as the number of perfective changed SLOCs increases, the number of pre-delivery and post-delivery defects also increases.

4.2.3 PROCESS IMPACT ON QUALITY

This subsection investigates the impact of productivity on the number of pre-delivery and post-delivery defects. This is an important area because the customer is not only interested in software maintenance being performed in a cost-effective, timely fashion, but also in the quality of the delivered software. In order to investigate the relationship between corrective and perfective productivity, rank correlations will again be used.

	CORRECTIVE PRODUCTIVITY	PERFECTIVE PRODUCTIVITY
PRE-DELIVERY DEFECTS	- 0.8214	0.4545
POST-DELIVERY DEFECTS	- 0.8214	0.5775

Table 4. Rank Correlations of Defects and Productivity

Table 4 presents rank correlations between productivity and quality for corrective and perfective maintenance. Perfective productivity has weak correlation with the number of pre-delivery and post-

delivery defects detected, while corrective productivity has a very strong negative correlation with the number of pre-delivery and post-delivery defects detected. This implies that as corrective maintenance productivity increases, the number of defects increases.

3. CONCLUSIONS

The results of this investigation suggest several interesting, and perhaps provocative, characteristics of software maintenance. Viewing the similarities, differences, and statistical relationships between perfective and corrective maintenance confirms a previously advanced "rule of thumb", questions another such rule, and leads to the proposal of a new rule.

Requirements volatility analysis led to the discovery of some important differences between perfective and corrective. The size of change and distribution of change to the product differed significantly between perfective and corrective maintenance; perfective maintenance resulted in larger and more distributed change to the software product than corrective maintenance. However, productivity did not show a significant statistical difference because the average change per software module remained roughly the same for both types of maintenance. These results confirm the old rule: the more local the change to the software product, the easier the maintenance effort.

Analysis of the impact of perfective and corrective maintenance on the quality of the delivered software product provides two interesting results. First, strong positive rank correlation exists between the impact of perfective maintenance and the number of post-delivery defects detected in the software. This correlation suggests that as the impact of perfective maintenance increases the number of post-delivery defects also increases. Second, a strong negative correlation exists between the impact of corrective maintenance productivity and the number of pre-delivery and post-delivery defects. This correlation suggests that as the impact of corrective maintenance increases the number of post-delivery defects decreases. This result questions an old rule: fixing errors inserts new errors into software.

Our results suggest a new rule: as the impact of changes to the software product caused by corrections to the requirements document increase, the number of pre-delivery and post-delivery defects decreases. Obviously a realistic limit to this rule exists. The number of pre-delivery and post-delivery defects could not be eliminated by maximizing the impact of corrective maintenance.

These results illustrate two additional points. First, neither the size of the change nor the distribution of the change, taken individually, influence productivity. It is the combination of these factors which significantly impact the productivity of both perfective and corrective maintenance activities. Second, perfective and corrective maintenance differ significantly in both the impact on the software product and the impact on the number of defects. These two types of maintenance differ to the extent that they should be managed and assessed separately.

REFERENCES

Bollinger, T.B. and McGowen, C., "A Critical Look at Software Capability Evaluations," *IEEE Software*, July 1991.

Humphrey, W.S. and Sweet, W.L. "A Method for Assessing the Software Engineering Capability of Contractors," Software Engineering Institute, Carnegie Mellon University, September 1987.

Humphrey, W.S., *Managing the Software Process*, Addison-Wesley, 1989.

Jablonski, J., R., *Implementing Total Quality Management: An Overview*, Pfeiffer, 1991.

Pressman, R. S., *Software Engineering: A Practitioners Approach*, McGraw-Hill, 1987.

Thayer, R. H., Pyster, P. and Wood, R. C., "Validating Solutions to Major Problems in Software Engineering Project Management," *IEEE Computer*, August 1982.

A Quantitative Comparison of Corrective and Perfective Maintenance

Software Engineering Workshop
December 1, 1994

Joel Henry
Jim Cain

East Tennessee State University
Department of Computer and Information Sciences

Overview

- Introduction
- Process
- Data collection
- Quantitative comparison
  - Similarities
  - Differences
- Conclusions

Introduction

- Focus

  - assessment of corrective and perfective maintenance activities driven by changes to the specification documents

- Purpose

  - quantitative comparison of maintenance process and product impact

Process Terminology

- Items

  - Upgrade
  - Corrective

- Specification Changes (SCs)

  - Upgrade
  - Corrective

- Miscellaneous terms

  - SLOCs
  - Modules

Data Collection

- WHAT :
  - Process and product data
  - Corrective and perfective maintenance data
- HOW:
  - Item, specification change, and computer program change numbers
  - Validation performed by multiple groups
- WHERE:
  - Storage in a single, central, tightly controlled database

SIMILARITIES: PRODUCTIVITY

- Corrective Items vs. Perfective Items
  - Basic statistics showed only a 5.6% difference in SLOCS per person day
- Corrective SCs vs. Perfective SCs
  - Basic statistics showed only a 8.5% difference in SLOCS per person day
- Mann-Whitney Test showed no statistical difference in productivities

SIMILARITIES: SIGNIFICANT FACTOR

	CORRECTIVE ITEM SLOCS per PERSON DAY	PERFECTIVE ITEM SLOCS per PERSON DAY
SLOCS per Module	0.951	0.788

- Coorelations of corrective items and perfective items with SLOCs per module

DIFFERENCES: SIGNIFICANT FACTOR

	CORRECTIVE CHANGED SLOCs	PERFECTIVE CHANGED SLOCs
PRE-DELIVERY DEFECTS	0.3214	0.9702
POST-DELIVERY DEFECTS	0.2143	0.8884

- Corrective changed SLOCs show weak coorelation to pre-delivery and post-delivery defects

- Perfective changed SLOCs show significant coorelation to pre-delivery and post-delivery defects

DIFFERENCES: PRODUCTIVITY/DEFECT RELATIONSHIP

	CORRECTIVE PRODUCTIVITY	PERFECTIVE PRODUCTIVITY
PRE-DELIVERY DEFECTS	- 0.8214	0.4545
POST-DELIVERY DEFECTS	- 0.8214	0.5775

- Productivity of perfective maintenance shows weak coorelation with both pre-delivery defects and post-delivery defects
- Productivity of corrective maintenance shows a negative coorelation with both pre-delivery and post-delivery defects

Conclusions

- Productivity similar
- Change per module similar
- Process and product impact on quality differ

Does Software Design Complexity Affect Maintenance Effort?

Andreas Epping*
Coopers & Lybrand
Consulting GmbH
New-York-Ring 13
22297 Hamburg, Germany

Christopher M. Lott
Software Technology Transfer Initiative
Department of Computer Science
University of Kaiserslautern
67653 Kaiserslautern, Germany

19th Annual Software Engineering Workshop, 30 Nov–1 Dec 1994

Abstract

The design complexity of a software system may be characterized within a refinement level (e.g., data flow among modules), or between refinement levels (e.g., traceability between the specification and the design). We analyzed an existing set of data from NASA's Software Engineering Laboratory to test whether changing software modules with high design complexity requires more personnel effort than changing modules with low design complexity. By analyzing variables singly, we identified strong correlations between software design complexity and change effort for error corrections performed during the maintenance phase. By analyzing variables in combination, we found patterns which identify modules in which error corrections were costly to perform during the acceptance test phase.

1 Introduction

Software systems seldom remain unchanged after their initial development and delivery. A system may be extended to fulfill new specifications or may be repaired to remove faults. These changes, as well as many others, are performed during a period of time called the maintenance phase.

Some authors see software design complexity as a highly important factor affecting the costs of software development and maintenance [Rom87, CA88]. We performed a study to test the hypothesis that changes to modules with high software design complexity require more personnel effort than changes to modules with low complexity. We define software design complexity in terms of several different factors, and test the hypothesis by investigating how the complexity factors affect the costs of changing the software.

If we can determine the impact of the complexity factors on maintenance effort, we can develop guidelines which will help reduce the costs of maintenance by recognizing troublesome situations early. In response to these situations, the developers may decide to reduce the software design complexity of the systems themselves, to develop tools that support maintenance of complex modules, to write documentation that helps the developers manage the complexity better, or simply to re-allocate resources to reflect the situation. Our results might even be used to justify an expensive, controlled experiment to test the hypothesis more rigorously.

In the case study presented here, we used an existing set of data to investigate the impact of software design complexity on the effort required to implement changes during the acceptance test and maintenance phases. We studied two FORTRAN systems from NASA's Software Engineering Laboratory (SEL). The independent variables of the design complexity included a mapping to the specification, global data bindings, and control flow relationships. The dependent variables on maintainability were gathered by the SEL and include the necessary effort for isolating and implementing changes.

This paper extends work first presented in [Epp94]. Section 2 gives the design of the case study, Section 3 discusses our complexity and effort metrics, and Section 4 explains the context of the study. Section 5 states the results for the maintenance and acceptance

*At the time this study was performed, Epping was a student in the Department of Computer Science, University of Kaiserslautern.

test data, and sketches related work. Finally, Section 6 summarizes lessons for the SEL, the researchers, and the software-engineering community.

2 Designing the Study

This study, which was motivated in part by [Rom87], began by refining the original hypothesis into two, closely related hypotheses:

Hypothesis 1: Changing modules that implement many specifications requires more effort than changing modules that implement few specifications.

Hypothesis 2: Changing modules that are tightly coupled to each other via data and control-flow relationships requires more effort than changing modules that are loosely coupled to each other.

2.1 Design

The case study to test our hypotheses was designed using the Goal/Question/Metric Paradigm [BW84, BR88]. Our G/Q/M goal was to analyze two FORTRAN systems for the purpose of characterizing them with respect to the influence of design complexity on the maintainability of modules, from the point of view of the researchers within the context of the SEL. We analyzed vertical design complexity (traceability to specifications) and horizontal design complexity (coupling among modules). We defined maintainability in terms of change isolation effort, change implementation effort, and the number of modules changed (locality of the change). Using these definitions, we refined the goal into a set of questions, and in turn refined the questions into a set of metrics. Figure 1 diagrams the relationship of the goal and the following sets of questions and metrics.

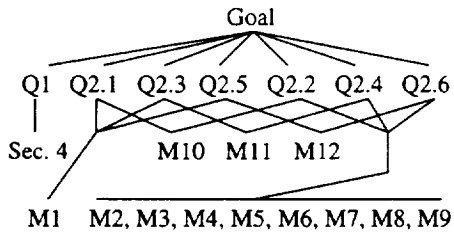


Figure 1: Goal, questions, and metrics

Q1: What are the characteristics of the software systems, the environment, the processes followed, and the personnel? Answers are given in Section 4.

Q2.1/2.2: Is the vertical/horizontal design complexity of modules affected by changes with high *isolation effort* greater than modules affected by changes with low effort?

Q2.3/2.4: Is the vertical/horizontal design complexity of modules affected by changes with high *implementation effort* greater than modules affected by changes with low effort?

Q2.5/2.6: Is the vertical/horizontal design complexity of modules affected by changes that touched a large *number of modules* greater than modules affected by changes that touched few modules?

Answers to questions Q2.x will be developed using the following design complexity and change effort metrics, which are discussed in detail in Section 3:

M1: The number of specifications a module fulfills, either directly or indirectly.

M2: Number of common blocks used in a module.

M3: Number of global variables visible in a module.

M4: Number of global variables used in a module.

M5: Ratio of used:visible global variables.

M6: Number of potential data bindings in a module.

M7: Number of used data bindings in a module.

M8: Measure of fan-in for a module.

M9: Measure of fan-out for a module.

M10: Isolation effort per module per change.

M11: Implementation effort per module per change.

M12: Number of modules affected by a change.

2.2 Available data

Although we would like to assume that all changes are similar in size, this may not be so for enhancements, which range from trivial to extensive. However, we can assume similarity in the size of changes for error corrections.

Table 1 shows the count of data points from the acceptance test and maintenance phases (error corrections are a subset of all changes). Although our original

Phase	Change types	
	Error Corrections	All Changes
Acceptance test	302	508
Maintenance	17	33

Table 1: Data points according to category

goal was to focus on maintenance changes, the limited data encouraged us to include acceptance-test changes. However, interpretation of that data is difficult owing to the different environments, as discussed in Section 4.

2.3 Analysis and threats to validity

The study tests our hypotheses by checking for relationships between the independent variables concerning software design complexity and the dependent variables concerning change isolation effort, change implementation effort, and number of modules changed. The appropriate statistical approach for univariate analysis is a correlation analysis. As will be explained in Section 3, both the isolation and implementation effort metrics lie on an ordinal scale, so we must use a correlation technique which does not require ratio or interval-scale data. We planned to compute Spearman rank-correlation coefficients with respect to single complexity measures of the modules and the maintainability measures.

Based on the notion that a combination of independent variables might better explain high change effort than only a single variable, we planned to analyze multiple variables in combination using a machine-learning technique called Optimized Set Reduction (OSR) [BTH93, BBH93]. OSR finds patterns in the independent (explanatory) variables which reliably predict values of a single dependent variable. The OSR approach is insensitive to the scale of the data, but requires a large data set, ideally several hundred points. We planned to apply the OSR technique to the full data vectors; i.e., consider all explanatory variables together.

If we can find strong correlations between design complexity values and change effort values, or can find patterns of large design complexity values that reliably predict which modules are expensive to change, we will have confirmed our hypotheses for this data set.

There were at least two threats to internal validity. First, the nature of a case study meant that we were not able to control or even measure the factors that influenced the SEL personnel during their day-to-day activities. Second, individual differences may be responsible for some variation (i.e., noise) in the data.

One significant threat to external validity is the specialization of the software-system design used by the SEL. These results may not be applicable to other FORTRAN systems.

3 Complexity and Maintainability

Curtis refines the concept of software complexity into algorithmic and psychological complexity [Cur80]. Algorithmic (or computational) complexity characterizes the run-time performance of an algorithm. Psychological complexity affects the performance of programmers trying to understand or modify a code module. We measured two aspects of psychological complexity, namely the vertical design complexity (the relationship between specifications and modules) and the horizontal design complexity (the relationship between modules). A module is a file with a single subroutine. These relationships are illustrated in Figure 2.

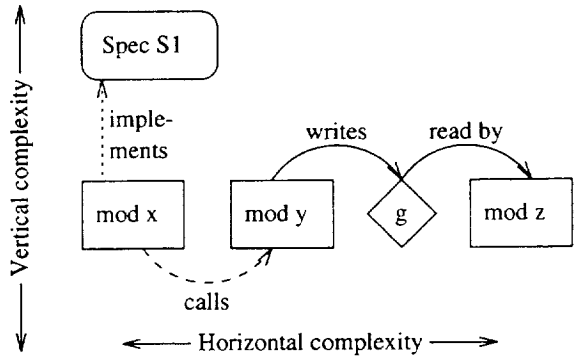


Figure 2: Vertical and horizontal design complexity

3.1 Vertical complexity: the relationship between specifications and modules

The vertical complexity of a module x is the number of specifications the module helps implement. To measure vertical complexity, we count how many specifications a module implements directly (mentioned in the documentation) or indirectly (invoked by another module that implements the specification directly or indirectly). An example is shown in Figure 2, where module x helps implement specification S1 directly and calls module y, meaning that module y helps implement S1 indirectly.

3.2 Horizontal complexity: the relationship between modules

The horizontal complexity of a module x is characterized by the number of connections between that module and other modules. An example is shown in Figure 2, where module y writes data into a global variable g, that is read in turn by module z. We analyzed the source code to gather data for the following metrics:

- Number of *COMMON blocks* which are referenced in a module.
- Number of *visible global variables*; i.e., the variables defined in the referenced COMMON blocks.
- Number of *used global variables*; i.e., the visible global variables that were also used in the code.
- *Ratio* of used global variables to visible global variables.
- For modules p and q, and a variable x within the static scope of both p and q, a *potential data binding* is defined as an ordered triple (p, q, x) [HB85].
- Again using p, q, and x, a *used data binding* is a potential data binding where p and q either read a value from or assign a value to x [HB85].
- The *fan-in* measure of a module is the number of other modules which call the module.
- The *fan-out* measure of a module is the number of other modules which the module calls.

3.3 Maintainability

Maintainability is an abstract concept that cannot be assessed directly but may be defined using attributes of the software that can be measured. We use change effort as our metric for maintainability.

Changes. The SEL distinguishes between three types of changes. An *error correction* repairs faults in the software. An *enhancement* implements changes for extended specifications. An *adaptation* makes provisions for alterations in the system's environment. For us, the error corrections were of primary interest.

Effort data. The analyses presented here are based on a four-step model of the change activity that guides data collection. In step one. the developers/maintainers become aware of the need for a change. Step two involves isolating the modules to be changed. In step three, they plan and implement the change. Finally, in step four they test the changed code. The change effort data that was available to us were limited to the following, routinely collected items [Nat91b]:

- Isolation effort: the effort to determine which modules must be changed (step two).
- Implementation effort: the effort to plan, implement, and test the change (steps three and four)
- Locality: the number of components affected by a change.

Effort expended during the maintenance phase is collected as a point on an ordinal scale, namely "less than one hour," "one hour to one day," "one day to one week," "one week to one month," and "greater than one month." Effort expended during the acceptance test phase is collected using the ordinal scale of "less than one hour," "one hour to one day," "one day to three days," and "more than three days."

4 Context of the Study

The study was conducted on two projects developed by the Flight Dynamics Division (FDD) of NASA's Goddard Space Flight Center. Data about the FDD's projects are gathered by the Software Engineering Laboratory (SEL), a cooperative effort of NASA's FDD, Computer Sciences Corporation, and the University of Maryland. The SEL was founded and began collecting data about the FDD's development activities in 1976. Data collection from maintenance activities began in 1988 [RUV92].

4.1 FDD Staff

The staff who performed the changes were familiar with both the application domain (ground-support software for satellites), which were similar for both systems, and the solution domain (FORTRAN), which was identical for both systems.

4.2 Activities in the acceptance test phase

During the acceptance test phase, the original developers exercise the system to detect failures and repair faults as needed [Nat91a]. Enhancements and adaptations may also be made to the software during this phase owing to new requirements.

4.3 Activities in the maintenance phase

During the maintenance phase, a team of software engineers who were not the original developers tests the software using simulators and modifies the systems as needed [Nat91a]. These engineers are experts in their application domain, but not necessarily highly familiar with the software systems. The maintenance phase essentially ends when satellites are launched; in any case, no data are collected following the launch.

4.4 The software systems

Project 1 and Project 2 (names have been changed) are ground-support software systems that were coded in FORTRAN. Both were single-mission systems.[1] Their sizes were approximately 130 and 180 KSLOC (carriage returns). These systems determine the exact position of a satellite with respect to other planetary bodies using data sent by the satellite. The systems do not run continuously, they are not subject to real-time constraints, and they are not required to meet highly stringent reliability requirements. For both projects, the software architecture and document standards are highly similar and specific to the FDD environment.

4.4.1 Specifics of Project 1

Project 1 consists of 582 modules. Of those, 23 modules are assembler modules, with a range of 6–3100 SLOC (carriage returns). The other 559 modules are FORTRAN modules (range 2–3200 SLOC). The system consists of 15 subsystems.

Changes in acceptance test. The developers processed 179 change requests during acceptance testing. Those change requests directly affected 163 unique modules, but owing to multiple changes to the same modules, there were 306 changes to code modules. Of the 163 changed modules, 32 modules were not available to us, or were assembler modules that were not analyzed. Therefore 48 changes to individual modules and 33 change requests total could not be analyzed.

Project 1 was in development (design, code, and test activities) for approximately 28 calendar months. Of those 28 months, the acceptance test phase lasted approximately 5 months.

Changes in maintenance. The single maintainer processed 15 change requests during maintenance. Of those, 5 were corrections, 9 were enhancements and 1 was an adaptation. Those change requests directly affected 28 unique modules, but because of multiple changes to the same modules, there were 37 changes to code modules. The assembler modules were not considered (5 change requests, 2 modules).

The maintenance phase for Project 1 began in 1988. Because of launch delays, it lasted about 33 months. The level of effort was extremely low for much of that time.

4.4.2 Specifics of Project 2

Project 2 consists of 816 modules. Of those, 31 modules are assembler modules (range 6–7300 SLOC). In addition to the 747 FORTRAN modules (range 3–2800 SLOC), there are 38 data files (range 9–400 SLOC). The system consists of 30 subsystems.

Changes in acceptance test. The developers processed 413 change requests during acceptance testing. Those change requests directly affected 346 unique modules, but because of multiple changes to the same modules, there were 850 changes to code modules. Of the 346 changed modules, 119 modules were not available to us, or were assembler modules which were not analyzed. Therefore 238 changes to individual modules and 136 change requests total could not be analyzed.

Project 2 was also in development for approximately 28 calendar months. Of those 28 months, the acceptance test phase lasted approximately 7 months.

Changes in maintenance. The four maintainers processed 25 change requests during maintenance. Of those, 12 were corrections, 12 were enhancements, and 1 was an adaptation. Those change requests directly

[1] A single-mission system is expected to cost 2% of development costs per year in maintenance until it is taken out of service, while a multi-mission system is expected to cost 10% [PS93].

affected 55 unique modules, but because of multiple changes to the same modules, there were 67 changes to code modules. Fortunately for our analysis, the assembler modules were not changed.

The maintenance phase for Project 2 began in 1988 and lasted about 19 months.

5 Results

After discussing some problems with the data, we present results from analyzing the maintenance and acceptance test data and sketch results from related work.

5.1 Data difficulties

We encountered some difficulties while trying to collect the data for the metrics defined in Section 2. In all fairness to the SEL, their data-collection forms were not designed to support such a detailed investigation, and we could not change data collection after the fact, so some problems were to be expected.

First, collecting data for metric M1 depended both on the modularity of the specification and the traceability of the specification to the code. At one extreme of modularity, the whole project can be seen as one single specification, while at the other extreme, every condition such as "x > 0" can be also seen as a specification. We began by using the system description document, in which a system is divided into 40–70 subspecifications. Even with this coarse level of modularity, it was not possible to map the modules to the subspecifications with any hope of accuracy because there was no document containing this information. We resolved this difficulty by simplifying the problem. Because the subsystems (Projects 1 and 2 had 15 and 30, respectively) were easily identifiable both in the requirements document and in the code, we essentially labeled each subsystem a "specification." Then we traced modules back to subsystems by analyzing the calling structure of the code.

The change effort data presented a second problem. In the SEL environment, a change activity occurs in response to a change request, and may affect many modules. The effort data are collected for each change activity, but no data for the change effort *per module* are collected. Because it is impossible to determine from the data how much change effort was expended on individual modules, we could not obtain values for metrics M10, M11, and M12 as originally planned. We resolved this difficulty by using an average for each change, namely the average of the complexity measures that were collected from the modules affected by that change. All analyses therefore are focused on *changes* rather than modules. However, by averaging, we reduced the range in complexity values, possibly losing significant differences.

Finally, we concluded that significant differences in effort were hidden by the ordinal scale of the effort data. For example, a maintenance change that required 9 hours of implementation effort is quite different from one that required 39 hours, but both are classified identically as "one day to one week."

5.2 Results from the maintenance data

5.2.1 Vertical complexity measures

First we tested hypothesis 1 using maintenance data, subject to the caveats discussed in Section 5.1.

Data collection process. We built a prototype tool that extracted the module calling trees from the FORTRAN code for each subsystem. This information told us which modules were part of a particular subsystem. While collecting these data, we found that not all of the modules changed are executable modules, and therefore are not in the call tree. Measures of change effort were obtained by querying the SEL database [Nat90] and by examining the data-collection forms completed by the maintainers after making the changes.

Results from univariate analyses. For Project 1, 19 modules that were changed were found in the call tree. Of those 19 executable modules, only 3 supported multiple subsystems; i.e., helped implement more than one specification. For Project 2, 32 modules that were changed were found in the call tree. Of those 32 executable modules, only 1 supported multiple subsystems. This left us with 4 data points for changed modules which supported multiple subsystems. None of the 4 modules participated in changes with above-average isolation or implementation effort.

Results from multivariate analyses. The OSR technique requires a large set of data to be effective. Because the maintenance data set was too small to be used, we have no multivariate results.

Interpretation. We could not support hypothesis 1; the answer to questions 2.1, 2.3, and 2.5 was "not for these data." Although our analysis found many modules that supported more than one subsystem, few of those modules were changed. We later learned that many of the modules which are widely reused are utility functions or so-called "institutional software." This term refers to modules that are reused repeatedly from project to project and are rarely changed.

We also learned that subsystems are designed mostly in isolation from one another, with the result that modules are not reused widely across subsystems. Although our definition of a "specification" was arguably too coarse, we could not refine the traceability further without a detailed familiarity with the systems.

An interesting result was that for Project 1, 12 of the 19 changed executable modules were from a single subsystem. No comparable, frequently changed subsystem was identified in Project 2, although the changes were clustered in 5 of the 30 subsystems.

5.2.2 Horizontal complexity measures

Next we tested hypothesis 2 using maintenance data.

Data collection process. We built a prototype tool which counted the use of common blocks and common-block variables in the FORTRAN code, and reused the calling-tree information from the analysis of vertical complexity for the measures of fan-in and fan-out. After loading all the resulting data into a database system, it computed the necessary complexity values. Recall that module complexity values were averaged on a *per change* basis as explained in Section 5.1. Effort data were obtained as discussed in Section 5.2.1.

Results from univariate analyses. Figure 3 uses data about error corrections from the maintenance phase to plot isolation effort against the average number of used common blocks (metric M2) in the modules affected by each change. This figure shows a trend towards higher effort when the average number of common blocks is also high. Thus encouraged, we computed correlations for the change data from the maintenance phase.

Table 2 shows the Spearman rank-correlation coefficient values for the relationships between all independent and dependent variables for *all changes* during maintenance; Table 3 shows only the coefficient values for *error corrections*. The correlations were computed

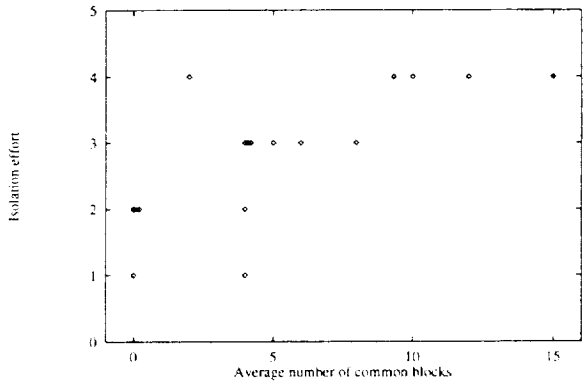


Figure 3: Data for error corrections in maintenance

as explained in Section 2.3. An approximation of the .05 cutoff (a 5% chance of obtaining the numbers by chance) is given in both tables to help judge the significance of the results.

Results from multivariate analyses. As mentioned previously, we had too few data points to apply OSR to the maintenance data.

Interpretation. When considering all changes during maintenance, all measures of global variables correlated positively (some significantly) with isolation effort. The counts of used globals and actual data bindings showed the most significant correlation of all measures; in an absolute sense the correlation is weak (approximately 0.60). These results support the idea that global variables make a program difficult to understand, although this conjecture was not supported by [LZ84] (see also Section 5.4). We found no significant correlation between complexity measures and implementation effort, nor between complexity measures and the number of modules changed. The measures of control-flow complexity were not helpful. To summarize the results for all changes, we can support hypothesis 2 in some respects: the answer to question 2.2 (isolation effort) is a qualified yes for some of the measures, but the answer to questions 2.4 (implementation effort) and 2.6 (locality) is "not for these data."

When considering just the error corrections during maintenance, the measures of global variables correlate positively and much more strongly with the isolation effort than previously. Both the counts of used globals

Dependent variables (averages per change)	Independent variables		
	Isolation effort	Implem'n effort	Modules changed
M2: Common blocks	**.415**	.088	-.376
M3: Visible global vars	**.575**	.207	-.303
M4: Used globals vars	**.628**	.228	-.198
M5: Ratio used:visible globals	**.534**	.303	.105
M6: Potential data bindings	**.528**	.193	-.330
M7: Used data bindings	**.599**	.214	-.294
M8: Fan-in	-.268	-.010	.067
M9: Fan-out	.322	.181	-.363

N = 33, critical r (.05) t approximation = .343

Table 2: Spearman rank-correlation coefficients for *all changes* during *maintenance*

Dependent variables (averages per change)	Independent variables		
	Isolation effort	Implem'n effort	Modules changed
M2: Common blocks	**.738**	.403	-.169
M3: Visible global vars	**.785**	**.511**	-.143
M4: Used global vars	**.799**	**.511**	.000
M5: Ratio used:visible globals	**.619**	**.493**	.164
M6: Potential data bindings	**.770**	**.511**	-.214
M7: Used data bindings	**.813**	**.511**	-.102
M8: Fan-in	-.406	-.208	-.096
M9: Fan-out	**.610**	**.545**	-.143

N = 17, critical r (.05) t approximation = .482

Table 3: Spearman rank-correlation coefficients for *error corrections* during *maintenance*

and actual data bindings again showed the most significant correlations, in this case fairly strong in an absolute sense (approximately 0.80). We also found correlations with implementation effort; some were significant but again weak in an absolute sense (approximately 0.50). Fan-out correlated positively weakly with both measures of effort. No measures correlated with the number of affected modules. To summarize the results for the error corrections, we can support hypothesis 2 strongly; the answers to questions 2.2, 2.4, and 2.6 are a reasonable yes, a weak yes, and another "not for these data."

Finally, we found it interesting that the number of changed modules frequently correlated *negatively*, although weakly, with the complexity values. We are unable to explain this result.

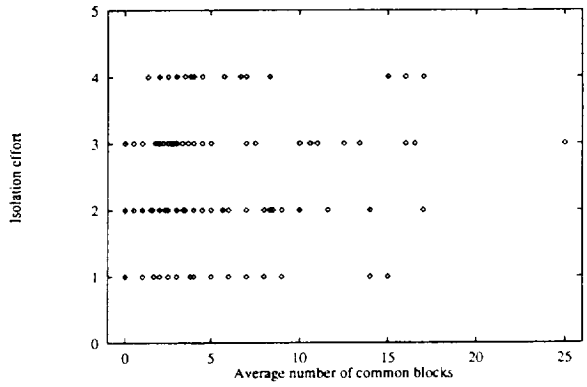


Figure 4: Data for error corrections in acceptance test

5.3 Results from the acceptance test data

As mentioned earlier, we extended the scope of the study to include data from the acceptance test phase. The results must be interpreted carefully, because the measures of the source code were computed using the code as it existed at the end of the maintenance phase. A version of the code from the end of the acceptance test phase was not available.

5.3.1 Vertical complexity measures

Due to the problems discussed in Sections 5.1 and 5.2.1, we did not test hypothesis 1 using acceptance test data.

5.3.2 Horizontal complexity measures

Finally, we tested hypothesis 2 using the acceptance test data.

Data collection process. The measures M2 to M9 were computed from the source code as of the end of the maintenance phase. Again, module complexity values were averaged on a *per change* basis as explained in Section 5.1. Measures of change effort were obtained by querying the SEL database [Nat90].

Results of univariate analyses. Figure 4 uses data about error corrections from the acceptance test phase to plot the isolation effort against the average number of common blocks in the modules affected by each change. Plots of isolation and implementation effort against other independent variables were similarly random, which discouraged us from computing univariate correlations.

Results of multivariate analyses. Because we had data for several hundred changes in the acceptance test phase, we were able to apply the OSR technique [BTH93, BBH93]. Based on the results achieved when working with the maintenance data, we restricted the data set to the error corrections. All analyses took the approach of trying to identify whether the error corrections (changes) would be inexpensive or expensive, where inexpensive was defined as requiring one day or less (the lower two values on the ordinal scale) and expensive was defined as requiring more than one day (the upper two values). The technique found reliable patterns when using isolation effort as the dependent variable, but found no reliable results when using implementation effort or locality as the dependent variable.

All results are expressed as OSR patterns. Patterns provide interpretable models where the impact of each predicate can be easily evaluated [BTH93]. An OSR pattern is a set of one or more predicates, where predicates have the form ($EV_i \in EVclass_{ij}$), meaning that a particular explanatory (independent) variable EV_i belongs to part of its value domain, i.e., $EVclass_{ij}$. Taken as a whole, the pattern predicts whether the value of the dependent variable will be in the high-cost or the low-cost class. For each pattern, we state the reliability of the prediction (a measure of pattern accuracy), and the significance level of the reliability (whether the pattern is based on a sufficiently large set of data to be trusted). The OSR technique found reliable and significant pat-

terns which predict low and high isolation effort. We present patterns which had high reliability values (> 0.8) and low reliability significance values (< 0.05).

Pattern L1:
Fan-in $\in$ 26-100% AND
fan-out $\in$ 0-50% $\Rightarrow$ low
(reliability 0.85, rel. sig. 0.011)

Pattern L1 suggests that modules with medium to high fan-in values and low fan-out values were easy to change (predicts low isolation effort). This pattern may indicate leaf modules (such as library subroutines) which are called frequently but call few other modules.

Pattern L2:
Used var $\in$ 0-12% OR
used db $\in$ 0-11% $\Rightarrow$ low
(reliability 0.92, rel. sig. 0.001)

Pattern L2 suggests that modules with low numbers of used variables or low numbers of used data bindings were easy to change (predicts low isolation effort).

Pattern H1:
Fan-in $\in$ 8-26% AND
(used db $\in$ 20-100% OR
used var $\in$ 20-100%) $\Rightarrow$ high
(reliability 1.00, rel. sig. 0.000)

Pattern H1 suggests that if a module is called by a relatively low number of other modules, and additionally has many used data bindings or many used variables, then that module was expensive to change (predicts high isolation effort).

Pattern H2:
Ratio used:visible $\in$ 63-100% AND
(vis var $\in$ 34-100% OR
used db $\in$ 30-100%) $\Rightarrow$ high
(reliability 1.00, rel. sig. 0.001)

Pattern H2 suggests that if a module has a high ratio of used to visible global variables, and additionally has many visible variables or many used data bindings, then that module was expensive to change (predicts high isolation effort).

Pattern H3:
Fan-out $\in$ 42–100%
AND used db $\in$ 59-100% $\Rightarrow$ high
(reliability 1.00, rel. sig. 0.007)

Pattern H3 suggests that modules which call many other modules and have many data bindings to other modules were expensive to change (predicts high isolation effort).

Interpretation. The univariate analyses were not helpful, but the OSR analysis identified some patterns that reliably characterize modules which participated in error corrections with both low and high isolation effort. All of the patterns support hypothesis 2. We have not established a causal relationship between the patterns and isolation effort, no statistical analysis technique does so, but we have identified a set of patterns that may be suitable for further investigation.

5.4 Results from related studies

We summarize the results of previous studies and experiments that analyzed the effects of design complexity on various dependent variables. Note that comparisons with related work are dangerous owing to different definitions of both independent and dependent variables.

Lohse and Zweben [LZ84] ran a controlled experiment to examine the effects of data coupling (data flow among modules) via global variables versus formal parameters, in the context of performing maintenance changes (enhancements) to two software systems. The primary dependent variable was the time required to implement the enhancement. They found no significant differences attributable to the use of global variables versus formal parameters.

Card et al. [CCA86] performed a case study on five SEL FORTRAN systems to examine the impact of various design practices on the dependent variables fault rate and cost in the context of development. They found no correlation with the percentage of referenced variables in COMMON blocks but a positive correlation with the number of descendants (fan-out). The percentage of unreferenced variables from COMMON blocks correlated with faults, but not with cost.

Rombach [Rom87] ran a controlled experiment to examine the effects of various programming-language constructs on isolation effort, implementation effort, and locality in the context of performing maintenance changes (enhancements) to two software systems. Complexity was measured in terms of information flow, which includes both data bindings and control flow between modules. He found a correlation of both isolation effort and locality with external complexity, but no

correlation of implementation effort with external complexity. Our results support his with respect to isolation and implementation effort, but not locality.

Card and Agresti [CA88] performed a case study on SEL FORTRAN systems to test for a relationship between a combined complexity measure and either productivity (lines of code delivered per unit of time) or fault rate in the context of development. Their combined measure of local complexity (e.g., cyclomatic complexity) and structural complexity (e.g., module fan-out) correlated well with productivity and number of faults. Because their study does not separate local (internal) complexity from structural (external) complexity, we cannot compare results.

6 Conclusion and Lessons Learned

The data from the two SEL systems support our hypothesis 2, so we can answer in the affirmative that horizontal design complexity appears to affect maintenance effort (isolation effort for error corrections). However, we have only demonstrated a possible relationship. We cannot establish causation using a case study.

Next we summarize the results of the study in terms of what the SEL can learn, what we learned, and what the software-engineering community can learn. Our analyses, which we primarily see as pointers for further investigation, found a number of relationships between software design complexity and maintenance effort that might help the SEL predict maintenance effort. Univariate analysis showed that the metrics "used globals" and "used data bindings" correlated strongly with the isolation effort for error corrections performed during the maintenance phase. Data for other metrics relating to the definition and use of global variables also correlated with isolation effort, but much less strongly with implementation effort. The measure of fan-out was also somewhat helpful in explaining high isolation effort. Multivariate analysis of acceptance test data using OSR found a number of patterns which were strong indicators of both low and high isolation effort in this data set. Future studies could be performed using other SEL systems to test whether the relationships and patterns which we found hold for more than just the two systems that we analyzed.

We gained a better understanding of the data required for thoroughly testing our hypotheses. First, to measure vertical complexity, both the modularity of the specification and its traceability to the code must be addressed. To solve the latter problem, a traceability matrix could be constructed in which the rows represent individual code modules and the columns represent units of the specification. A mark in the matrix means that the module of that row implements the unit of specification of that column. To build such a matrix, the modularity of the specification is critical, but beyond the scope of this paper. Second, to measure the effort required for a change, we need to collect the isolation and implementation effort on a per-module basis whenever possible. A minor change to the SEL's data-collection forms could be to collect an estimate of the percentage of the total effort required by each module. However, some effort, such as the effort to test the changed modules together, cannot be allocated to individual modules. Third, the simplest and most helpful change to the SEL's data collection forms (from our point of view) would be the use of a ratio scale such as days or hours for collecting effort data instead of the ordinal scales currently in use. This would allow us to distinguish more precisely between different changes as well as to compare effort data between the maintenance and acceptance test phases.

Finally, we believe that an empirical investigation such as this one uncovers more challenging questions than it answers. Future work might include replicating our study by analyzing the designs of other SEL software systems or systems from other software development organizations. Our data might also be used as a basis for planning and running a controlled experiment such as the one discussed in [Rom87] to test our hypotheses more rigorously. In a controlled experiment, programmers (subjects) might implement changes of similar sizes in modules that have low, medium, and high software design complexities. This would allow the researchers to control for many effects as well as to measure the effort required on a per-module basis to implement changes. Such an experiment would offer stronger evidence for refuting or accepting our hypotheses than any case study.

7 Acknowledgements

We would like to thank Lionel Briand and Alfred Bröckers for help with the analyses, Dieter Rombach for suggesting the hypotheses, Jon Valett for answering our questions, and most importantly, the SEL for trusting us with their systems and data.

References

[BBH93] Lionel C. Briand, Victor R. Basili, and Christopher J. Hetmanski. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transactions on Software Engineering*, 19(11):1028–1044, November 1993.

[BR88] Victor R. Basili and H. Dieter Rombach. The TAME Project: Towards improvement–oriented software environments. *IEEE Transactions on Software Engineering*, SE-14(6):758–773, June 1988.

[BTH93] Lionel C. Briand, William M. Thomas, and Christopher J. Hetmanski. Modeling and managing risk early in software development. In *Proceedings of the* 15^{th} *International Conference on Software Engineering*, pages 55–65. IEEE, May 1993.

[BW84] Victor R. Basili and David M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, SE-10(6):728–738, November 1984.

[CA88] David N. Card and William W. Agresti. Measuring software design complexity. *Journal of Systems and Software*, pages 185–197, June 1988.

[CCA86] David N. Card, Victor E. Church, and William W. Agresti. An empirical study of software design practices. *IEEE Transactions on Software Engineering*, SE-12(2):264–271, February 1986.

[Cur80] Bill Curtis. Measurement and experimentation in software engineering. *Proceedings of the IEEE*, 68(9):1144–1157, September 1980.

[Epp94] Andreas Epping. An empirical investigation of the impact of the structure of two software systems on their maintainability (in German). Master's thesis, Department of Informatics, University of Kaiserslautern, 67653 Kaiserslautern, Germany, April 1994.

[HB85] David H. Hutchens and Victor R. Basili. System structure analysis: clustering with data bindings. *IEEE Transactions on Software Engineering*, SE-11(8):749–757, August 1985.

[LZ84] John B. Lohse and Stuart H. Zweben. Experimental evaluation of software design principles: An investigation into the effect of module coupling on system modifiability. *Journal of Systems and Software*, 4(4):301–308, November 1984.

[Nat90] National Aeronautics and Space Administration. Software Engineering Laboratory (SEL) Database Organization and User's Guide, Revision 1. Technical Report SEI-89-101, NASA Goddard Space Flight Center, Greenbelt MD 20771, February 1990.

[Nat91a] National Aeronautics and Space Administration. Manager's handbook for software development. Technical Report SEL-84-101, NASA Goddard Space Flight Center, Greenbelt MD 20771, 1991.

[Nat91b] National Aeronautics and Space Administration. Software engineering laboratory (SEL) relationships, models, and management rules. Technical Report SEL-91-001, NASA Goddard Space Flight Center, Greenbelt MD 20771, February 1991.

[PS93] Rose Pajerski and Donald Smith. Recent SEL experiments and studies. In *Proceedings of the* 18^{th} *Annual Software Engineering Workshop*, pages 81–94. NASA Goddard Space Flight Center, Greenbelt MD 20771, 1993.

[Rom87] H. Dieter Rombach. A controlled experiment on the impact of software structure on maintainability. *IEEE Transactions on Software Engineering*, SE-13(3):344–354, March 1987.

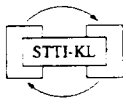
[RUV92] H. Dieter Rombach, Bradford T. Ulery, and Jon Valett. Toward full life cycle control: Adding maintenance measurement to the SEL. *Journal of Systems and Software*, 18(2):125–138, May 1992.

Does Software Design Complexity Affect Maintenance Effort?

A study of existing NASA/SEL data

Andreas Epping, Uni-KL (M.S. thesis)
Christopher Lott, STTI-KL

19th GSFC Software Engineering Workshop
1 December 1994



Overview

- Problem and hypotheses
- Vertical and horizontal design complexity
- Study design, independent and dependent variables
- Results for maintenance data
- Results for acceptance test data
- Conclusions and lessons learned

Problem and hypotheses

It is generally believed that software design complexity affects error rate, change effort etc.

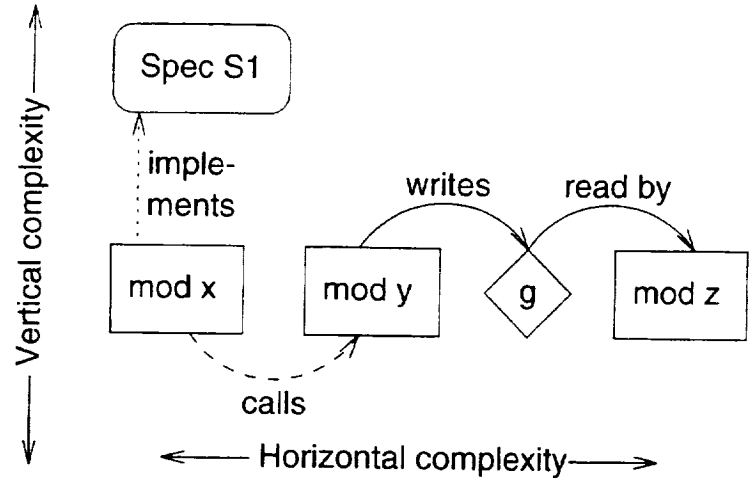
Supporting studies include: Card et al., TSE 86; Rombach, TSE 87; Card & Agresti, JSS 88; Briand et al., CSM 93.

We used existing SEL data to test two related hypotheses:

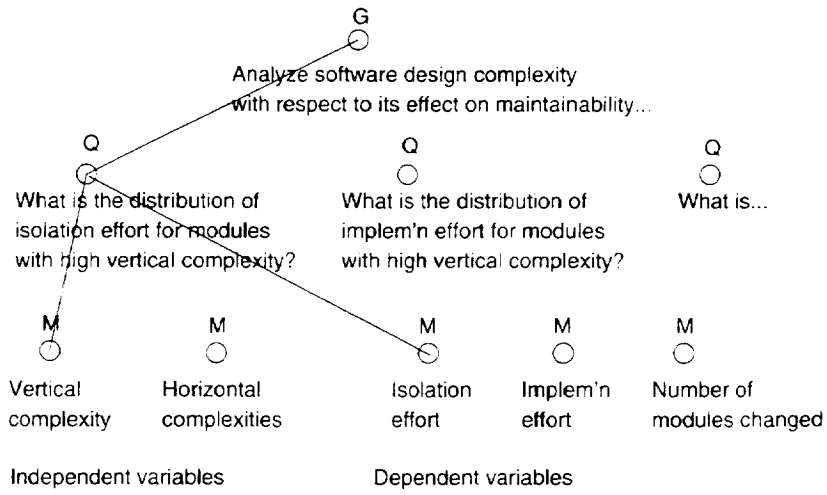
Hypothesis 1: Module implements many specifications (vertical complexity) ⇒ maintenance effort will be high

Hypothesis 2: Module is tightly coupled to others (horizontal complexity) ⇒ maintenance effort will be high

Terminology: Design complexity



Design via G/Q/M



Design: variables

- Independent variables (newly gathered):
  - Vertical design complexity (1 measure)
    - Number of specifications which a module implements (in)directly
    - Problems: modularity of the specification and traceability to code
  - Horizontal design complexity (8 measures)
    - Number of COMMON blocks referenced in a module, ...
    - Minor problem: limited to static metrics derived from the code
- Dependent variables (existing data):
  - Maintainability (3 measures)
    - Isolation effort, Implementation effort, Number of modules changed
    - Problems: collected per change, not per module; ordinal scale for effort

Results for error corrections in maintenance

Unable to test hypothesis 1 (vertical complexity).

Results for hypothesis 2 (horizontal complexity) using 17 data points:

- Significant and strong correlations found with isolation effort

  Example: 0.785 for count of visible global variables (.05 cutoff: .482)

- Significant but weak correlations found with implementation effort

  Example: 0.511 for count of visible global variables (.05 cutoff: .482)

- No significant correlations found with locality

  Example: -.303 for count of visible global variables (.05 cutoff: .482)

Results for error corrections in acceptance test

Unable to test hypothesis 1 (vertical complexity).

Results for hypothesis 2 (horizontal complexity) using 302 data points:

- Analyzed variables in combination using Optimized Set Reduction (OSR)

- Found reliable patterns for complexity values which predict isolation effort:
  - Fan-in in 26-100% of value range AND fan-out in 0-50% ⇒ low iso. eff. (Reliability 0.85, reliability significance 0.011)
  - Fan-out in 42-100% AND used data bindings in 59-100% ⇒ high iso. eff. (Reliability 1.00, reliability significance 0.007)

Related work

Lohse & Zweben 1984 (JSS):
Controlled experiment to compare coupling via globals vs. formal parameters.
Results showed no significant difference; is not directly comparable.

Card et al. 1986 (TSE):
Case study of influence of software design practices on cost and fault rate.
Fan-out was highly influential; the influence was not as large in our study.

Rombach 1987 (TSE):
Controlled experiment to analyze influence of complexity on maint. effort.
Isolation effort affected more than impl'n effort; supported by our study.

Card & Agresti 1988 (JSS):
Case study of influence of complexity on productivity and fault rate.
Different definition of complexity makes comparison impossible.

Conclusions and lessons learned

- Cannot test Hypothesis 1 (vertical complexity) using existing SEL data.
- Can support Hypothesis 2 (horizontal complexity) using existing SEL data:
  - Univariate analysis of horizontal complexity measures (i.e., coupling) identified modules that are likely to cause changes to be expensive.
  - OSR identified patterns in complexity (coupling) data likely to increase isolation effort, but found no reliable patterns for implementation effort.
- Lessons for the SEL:
  - Correlations and patterns help predict maintenance effort.
  - We are not confident enough to recommend complexity (coupling) limits.
  - Need effort data drawn from a ratio scale ("days").

Profile of Software Engineering Within the National Aeronautics and Space Administration (NASA)

19th Annual Software Engineering Workshop

Craig C. Sinclair, Science Applications International Corporation (SAIC)
Kellyann F. Jeletic, NASA/Goddard Space Flight Center (GSFC)

ABSTRACT

This paper presents findings of baselining activities being performed to characterize software practices within the National Aeronautics and Space Administration. It describes how such baseline findings might be used to focus software process improvement activities. Finally, based on the findings to date, it presents specific recommendations in focusing future NASA software process improvement efforts. NOTE: The findings presented in this paper are based on data gathered and analyzed to date. As such, the quantitative data presented in this paper are preliminary in nature.

BACKGROUND

The NASA Software Engineering Program was established by the Office of Safety and Mission Assurance (Code Q) at NASA Headquarters in 1991 to focus on the increasingly large and important role of software within NASA. The primary goal of this program is to establish and implement a mechanism through which long-term, evolutionary software process improvement is instilled throughout the Agency.

NASA's Software Engineering Program embraces a three-phase approach to continuous software process improvement. The first and most crucial phase is *Understanding*. In this phase, an organization baselines its current software practices by characterizing the software product (e.g., size, cost, error rates) and the software processes (e.g., standards used, lifecycle followed, methodologies employed). During the Understanding phase, models are developed that characterize the organization's software development or maintenance process. Models are mathematical relationships that can be used to predict cost, schedule, defects, etc. Examples are the relationships between effort, code size, and productivity or the relationship between schedule duration and staff months. This in-depth understanding of software practices is gained within the context of a specific software domain and must precede any proposed change. In the second phase, *Assessing*, a software improvement goal is identified. Based on the specific local organizational goal, a process change is introduced and its impact to the software process and product is measured and analyzed. The results of the Assessing phase are then compared back to the baseline developed during the Understanding phase. In the third phase, *Packaging*, experiences gained and lessons learned are packaged and infused back into the organization for use on ongoing and subsequent projects. Forms of packaging typically include standards, tools, training, etc. This three-phase software process improvement approach (Figure 1) is iterative and continuous.

The importance of the Understanding phase cannot be emphasized enough. Before an organization can introduce a change, it must first establish a baseline with which to compare the measured results of the change. This baseline must be domain-specific and the software goals of the organization must be clearly understood. Continual baselining is necessary not only because people, technology, and activities change, but also because identifying, designing, implementing, and measuring any change requires an in-depth understanding and monitoring of the particular process on which the change is focused. This implies that understanding and change are closely coupled, necessarily iterative, and never-ending. Continual, ongoing understanding and incremental change underlie any process improvement activity.

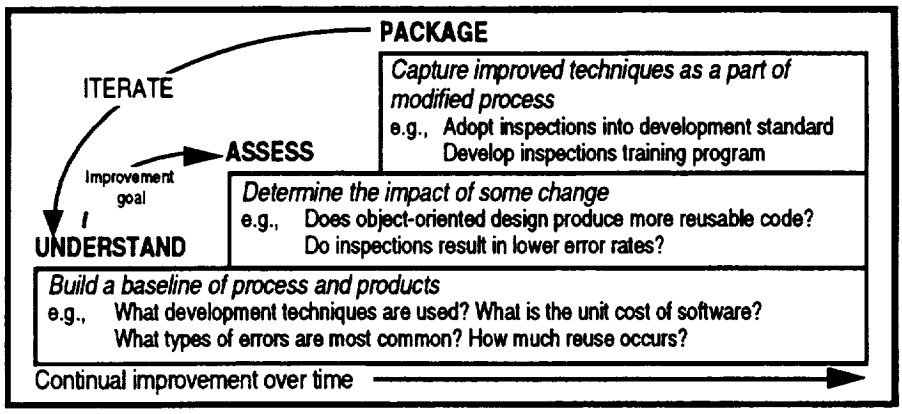


Figure 1. Three-Phase Approach to Software Process Improvement

This paper addresses the Understanding phase, that is, the baselining of NASA software. Since the baselining activities focus on a global organizational level, that is, NASA as a whole, the difference between applying the process improvement approach at the global level rather than at a local organizational level must first be addressed.

SOFTWARE PROCESS IMPROVEMENT AT A GLOBAL LEVEL

The steps in the software process improvement approach are applied differently at the global and local organizational levels. Figure 2 illustrates the differences between the local and global approaches. The *Understanding* phase is predominantly the same at both levels; basic characteristics of software process and product are captured. At a local level, models are also developed, e.g., cost and reliability models, to help engineer the process on ongoing and future projects. At a global organizational level, models can only be very general relationships, such as the percentage of application software in each of the identified software domains of an organization.

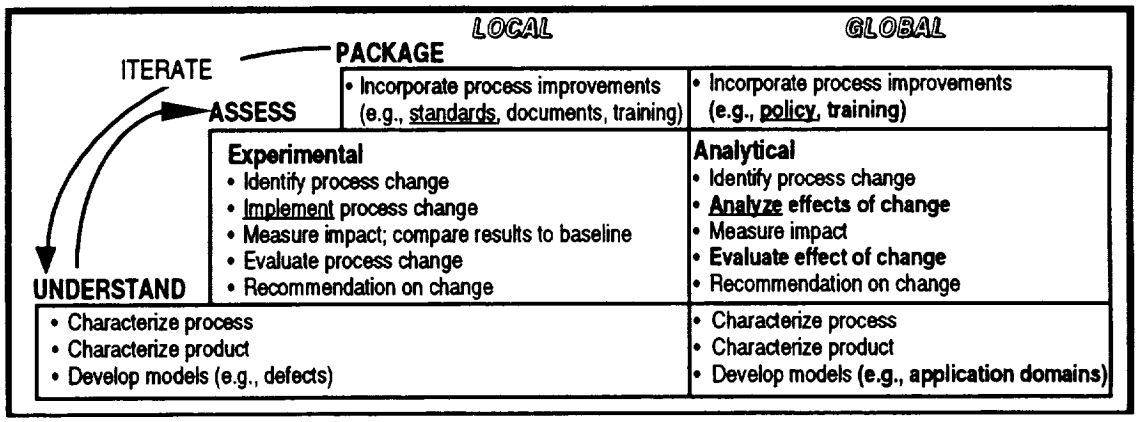


Figure 2. Local versus Global Software Process Improvement Approach

It is in the *Assessing* phase where most differences occur. At the local level, the Assessing phase is experimental in nature. Specific technologies are introduced to try to attain some local goal (e.g., inspections might be introduced to reduce error rates). The results of these experiments are then compared to the baseline from the Understanding phase to determine what impact the change has had. At a global level, the Assessing phase is analytical rather than experimental. Process changes are identified and the effects of the change(s) are analyzed and evaluated. Recommendations are then made at an organizational level. For example, a potential process change is identified such as code reuse. Analysis and evaluation of the effects of increased reuse in an organization is accomplished by determining which software domains would benefit from reuse, measuring via

survey the amount of reuse that currently takes place in those domains, and projecting the potential development time and cost savings that could be achieved by instituting a focused reuse program. Finally, specific recommendations are developed for the organization that stimulate the local implementation of code reuse.

The third phase, *Packaging*, is also similar at both levels. Changes that result in identified improvements are packaged and infused back into the organization's process. There are differences in the types of packages produced at both levels. At the local level packages might include experience-driven standards, guidebooks, training, and tools. Packages at the global level might include a high level training plan or a policy requiring software process improvement activities for various software domains and organizational levels. The global approach is intended to stimulate local implementations so each individual organization can attain its local goals and improve its products and processes. NASA will benefit, as a whole, as local benefits are attained in software organizations throughout the Agency.

BASELINING NASA'S SOFTWARE

As the critical first step toward continual software process improvement, NASA has recently begun the Understanding phase and has baselined its software products and processes. The Mission Operations and Data Systems Directorate (Code 500) at the Goddard Space Flight Center (GSFC) was first characterized to prototype and refine the steps necessary to construct such a baseline [Reference 1]. With the experiences gained during the Code 500 efforts, a single NASA Field Center, GSFC, was then baselined [Reference 2]. Lessons learned were again factored into the process and, finally, NASA as a whole was baselined to determine current Agency software practices. Since the NASA-wide data collection and analysis are not yet complete, this paper presents findings to date. The final NASA baseline, the *Profile of Software at the National Aeronautics and Space Administration*, is nearly complete and is targeted for completion in early 1995 [Reference 3].

During fiscal year 1993 (FY93), NASA software and software engineering practices were examined to gain a basic understanding of the Agency's software products and processes. The objective of the NASA baseline was to understand the Agency's software and software processes. There is no intent to judge right or wrong; it merely presents a snapshot in time of software within NASA. The baseline includes all software developed or maintained by NASA civil servants or under contract to NASA. It does not include commercial-off-the-shelf (COTS) software such as operating systems, network software, or database management systems. It also does not include COTS application packages such as word processing packages, spreadsheet software, graphics packages, or other similar tools hosted on workstations and personal computers.

To produce the baseline, software product and process data were gathered from seven NASA Field Centers[1] and the Jet Propulsion Laboratory. Data and insight gathering were performed using four approaches:

(1) *Surveys* administered in person to a representative set of civil servants and support contractors from across the NASA community

(2) *Roundtable* discussions consisting of a structured group interview process

(3) One-on-one *interviews* with management and technical personnel

(4) *Reviews* of organizational and project data (e.g., budgets, policies, software process development documentation)

Reference 4 provides additional details on the baselining approach.

[1]Data were collected from the following NASA Field Centers: Ames Research Center, GSFC, Johnson Space Center, Kennedy Space Center, Langley Research Center, Lewis Research Center, and Marshall Space Flight Center

The remainder of this paper focuses on the findings of the NASA baseline and how they might be used. The baseline will help NASA management understand the scope and extent of software work within the Agency. It will also assist managers in focusing future resources to improve NASA's software products and processes. The baseline can also be assessed to identify candidate areas for improvement. As the baseline findings are presented, examples are given as to how they might be used. Finally, recommendations are proposed for focusing future process improvement efforts.

NASA'S SOFTWARE PRODUCT BASELINE

This section presents results from the analyses performed on the product data gathered throughout the NASA Centers. This section summarizes a selected set of the software product baseline data that can be found in the draft *Profile of Software at the National Aeronautics and Space Administration* [Reference 3]. Examples of additional software product data detailed in the document include the amount of operational software per NASA Field Center, the size of the software domains at the Centers, allocation of resources to the life-cycle phases, and other measures.

The software product baseline characterizes the attributes of the software itself. This paper addresses several questions pertaining to NASA's software product:

- What *classes* of software exist?
- *How much* software exists?
- How much of NASA's *resources* are invested in software?
- What *languages* are used?

These product characteristics are discussed below.

SOFTWARE CLASSES

Six classes (domains) of software were identified throughout NASA. It was necessary to define separate software domains within NASA, since the development and maintenance practices, the management approach, and the purposes of the software in various domains are distinctly different. Hence the software improvement goals for varying domains are generally different. The definitions of the six NASA software domains are given below.

- *Flight/embedded* -- embedded software for on-board spacecraft or aircraft or ground command and control applications (e.g., robotics)
- *Mission ground support* -- software usually not embedded; operates on the ground in preparation for or in direct support of space and aircraft flight missions (e.g., flight dynamics, control center, command processing software, and software for crew or controller training)
- *General support* -- software that supports the development of flight and ground operational software (e.g., engineering models, simulations, engineering analyses, prototypes, wind tunnel analyses, test aids, and tools)
- *Science analysis* -- software used for science product generation, processing and handling of ancillary data, science data archiving, and general science analysis
- *Research* -- software supporting various studies in software, systems, engineering, management, and/or assurance (e.g., software tools, prototyping, models, environments, and new techniques)
- *Administrative information resources management (IRM)* -- software supporting administrative applications (e.g., personnel, payroll, and benefits software)

Figure 3 shows the distribution of these domains for operational software. Mission ground support and administrative/IRM software were found to be the largest and most prevalent software domains within NASA, accounting for over 60 percent of all NASA software. General support software was the next largest software domain, accounting for almost 20 percent of NASA software. The science analysis, research, and flight/embedded software domains were much smaller in size.

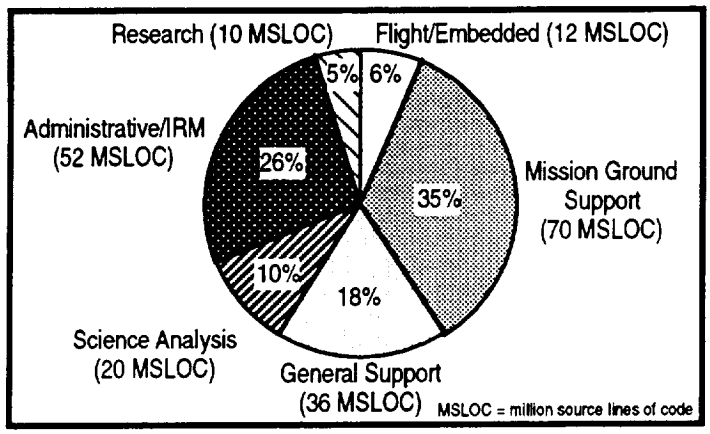


Figure 3 - Operational Software by Domain

How might such baseline information be used? The largest domains could indicate where software improvement efforts might most effectively be applied.

SOFTWARE QUANTITIES

During the baseline period, about 200 million source lines of code (SLOC) were in operational use. During that same period, NASA developed about 6 million SLOC (MSLOC). In terms of lines of operational code, almost 122 million SLOC within NASA is mission ground support (70 MSLOC) software and administrative/IRM (52 MSLOC) software. As mentioned in the previous section, focusing an effective software improvement program in these software domains has the potential of reaping enormous cost benefits. This type of data can be used to assist NASA management in seeing where they should focus their resources to improve software products and processes.

SOFTWARE RESOURCES

Figures 4 and 5 show the amount of resources invested in software in dollars and manpower, respectively. As these figures indicate, NASA has a significant investment in software. More than $1 billion of NASA's total $14 billion budget is spent on the development and maintenance of software (Figure 4). Most of NASA's software budget is spent on contractors, nearly 80 percent of NASA's software work is contracted out to industry. Software staffing accounted for more than 10 percent of NASA's total work force (Figure 5). This includes all civil servants and contractors who spend the majority of their time managing, developing, verifying, maintaining, and/or assuring software. These data can be used to help senior managers at NASA to understand the scope and extent of NASA's investment of manpower and budget in software.

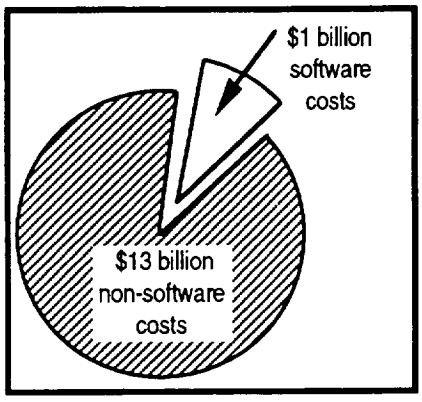


Figure 4 - Software Versus Total Costs

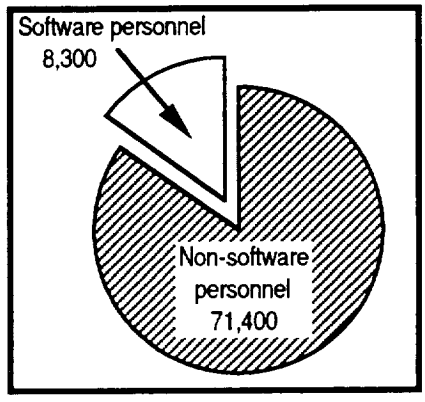


Figure 5 - Software Versus Total Staffing

SOFTWARE LANGUAGES

Figure 6 compares the preferences in software languages being used in current development efforts across NASA with those used in existing software now being maintained. Several trends are apparent. FORTRAN usage has remained relatively constant. Usage of both Cobol and other languages (e.g., Assembler, Jovial, Pascal), has decreased significantly, presumably replaced by the large increase in C/C++ usage. The usage of both C/C++ and Ada have increased dramatically. This implies that there is a significant trend toward C/C++ across NASA. Another trend is the lack of substantial movement toward Ada despite a decade of attention within NASA and advocacy from the Department of Defense. Although Ada use has increased, the magnitude of the increase is small compared to the intensity of past advocacy. It appears that Ada is not "catching on" within NASA culture and that C/C++ are becoming the languages of choice.

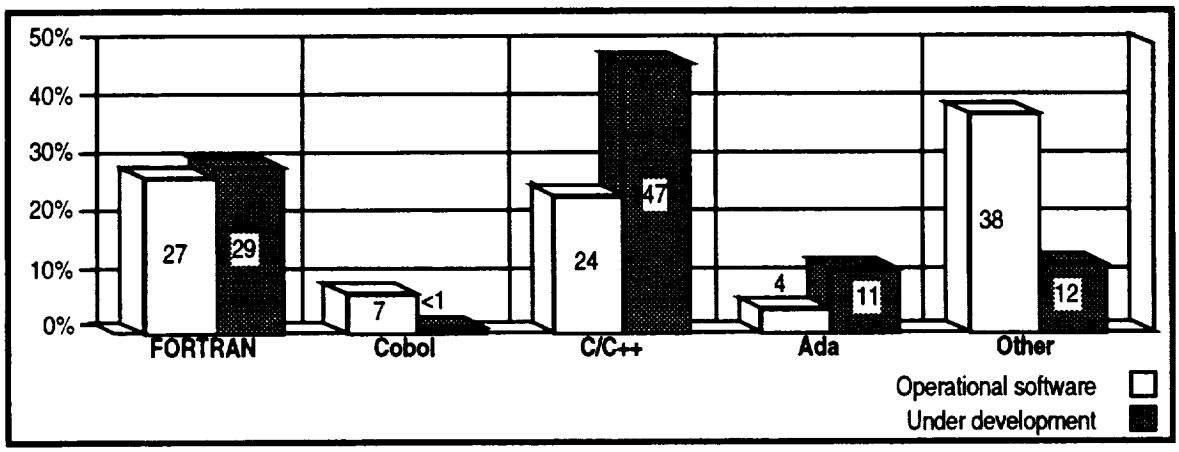


Figure 6. Language Preferences and Trends

Data such as the language preferences and trends might be used to focus training activities, not only toward language training, but also toward methodologies appropriate to specific languages.

NASA'S SOFTWARE PROCESS BASELINE

This section presents results from the analyses performed on the process data gathered throughout the NASA Centers. It summarizes a selected set of the software process baseline data that can be found in Reference 3. Examples of additional software process data detailed in the document include management experience, documentation standards, development tools, training, and other processes.

The software process baseline characterizes the attributes of the software practices. This paper addresses several questions pertaining to NASA's software process:

- What software *standards* are used and are helpful?
- How are *requirements managed*?
- How much and what type of *reuse* occurs?
- What are the Agency's practices with respect to software *metrics (measures)*?
- What *development methodologies* are used?

These process characteristics are discussed below.

SOFTWARE STANDARDS

A software standard refers to any mutually agreed upon specification for a software product or a software process within a software development or maintenance project. Examples of software standards related to software products are coding standards, language standards, and error rate specifications. Examples of software standards related to software processes are specifications of

software development standards, software configuration standards, and software methodologies. Almost all the written, baselined software standards within NASA are in the form of software development standards. This is a type of process standard that consists of one or more of the following: software life-cycle phases and their activities, software review requirements, and document format and content requirements. Though software standards exist at various levels within NASA organizations, there is relative little usage of software standards by NASA personnel. On the contrary, standards usage is widespread among NASA's support contractors, which is significant considering that they are responsible for nearly 80 percent of NASA's software work.

One resounding sentiment throughout the Agency was that the most used and useful software standards are typically defined at the project level. Software standards defined and imposed from higher organizational levels were widely ignored. Another observation supported by the process data was that the awareness of software standards baselined at higher organizational levels was relatively low. In fact, there was a clear trend that indicated that the higher up in the organizational chain the standard is baselined, the less likely the project software staff know of its existence.

When software standards do exist, they do not enjoy a high level of use and do not appear to be used by the majority within an organization. This observation appeared to be true at all organizational levels. However, when software standards are used, they are generally perceived as helpful. So even though software standards do not have an overall high level of use, those that do use them generally perceive them to be helpful. Finally, even when software standards exist and are used, they are not enforced by the organizational level at which they are baselined.

This information can be used to provide specific focus in developing and facilitating the effective use of software standards within the Agency.

REQUIREMENTS MANAGEMENT

Software requirements represent an agreement between NASA and its contractors as to what will be produced and how it will perform. These "agreements" form the basis for the software size, schedule, budget, and staffing levels. If the software requirements are not clearly defined before the onset of design, schedule slips, code growth, and cost overruns are often the result. Management of software requirements is especially important for NASA civil servants since over 80 percent of the software projects at NASA are developed or maintained by contractors.

A widespread finding throughout NASA was that unstable requirements were perceived as the major cause of software schedule slips, cost overruns, and code size growth problems. Unstable requirements were interpreted to mean not only changing requirements, but also missing and/or misinterpreted requirements. A related finding was that most of the NASA engineers and managers surveyed claimed that software requirements were generally not stable by the onset of preliminary design.

SOFTWARE REUSE

Software reuse is the establishment, population, and use of a repository of well-defined, thoroughly tested software artifacts. Software artifacts that can be reused include not only code, but software requirements specifications, designs, test plans, documentation standards, etc.

Throughout NASA, most focus on reuse is at the code level. On average, about 15 percent of code is reused from one project to another, however, there is considerable variance in reuse levels between Centers. The level of reuse was also observed to widely vary between projects within a given Field Center. In NASA overall, there was little in the way of defined approaches for handling software reuse.

SOFTWARE MEASUREMENT

Software measures are quantitative data that specify a set of characteristics about the software product or its processes. Software measures can be used to aid in the management of software

projects, help in the estimation of new projects, define and model an organization's software characteristics, and guide improvements of software and its processes.

The collection and utilization of software measures varied from non-existent to a few robust programs. Overall, relatively few NASA organizations collected software measures. Of those organizations surveyed that did collect software measures, less than half used the data to analyze and provide information back to the project. Overall, there was little evidence of the collection and use of measures throughout NASA.

DEVELOPMENT METHODOLOGIES

Figure 7 shows the relative awareness, training, and usage of several software development methodologies. Since structured analysis and Computer-Aided Software Engineering (CASE) tools have been around for a long time, it is not surprising that they are well known and widely used. There is a lot of awareness about object-oriented technologies, but usage is moderate. Some newer technologies, e.g., Cleanroom, are much less known and used. With the exception of CASE, one can also see a rather close link between the level of training and the level of usage. CASE is not a surprising exception since, as with other tools, people tend to jump in and use them rather than take courses or delve through documentation. One might surmise by the link with training and usage that NASA may be investing in "just in time" training.

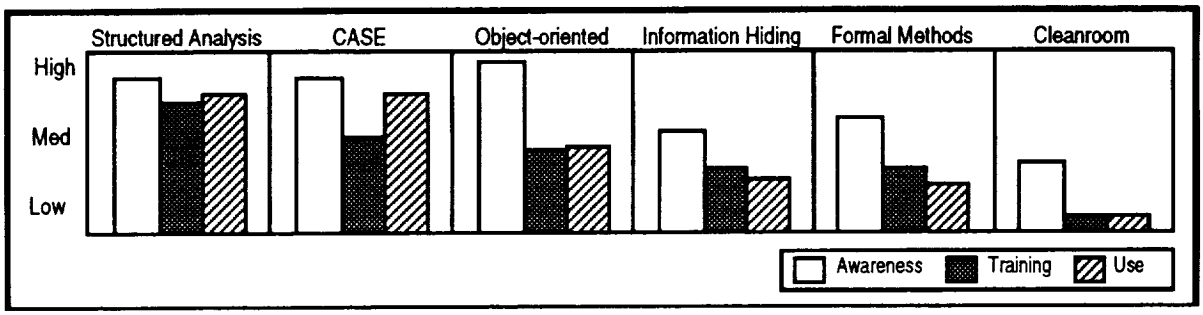


Figure 7. Development Methodologies

APPLYING THE FINDINGS

As previously indicated, the NASA baseline can be used to identify candidate areas for improvement and to develop specific recommendations for implementation of software improvement within NASA. These software improvement recommendations must not consist of rigid NASA-wide requirements imposed from above onto NASA projects. Rather, the software improvement recommendations at the higher levels of organization within NASA need to be top level policy and funding assistance, designed to stimulate and facilitate the development of local implementations of software improvement methods. If the goal is to bring software improvement into the projects, the projects must be given proper incentives and allowed to tailor software improvement implementation to their specific goals and domains. The following are two examples of how the NASA software baseline findings could be assessed and utilized.

Software Reuse

Recall that, on average, about 15 percent of the code is reused from one project to another. Throughout NASA, there is little or no emphasis on reusing anything but code. Overall, there are few defined approaches to reuse and only a few NASA organizations utilize software reuse as part of their software development process.

There are some NASA organizations who focus on more than just the reuse of code (e.g., reuse of code and architecture). These organizations have seen 75 to 80 percent reductions in both the time and cost to develop software. NASA might be able to leverage these few robust programs to assist the adoption of software reuse by other NASA organizations.

Applying proven NASA-developed solutions to the same software domains of other NASA organizations will give a much higher probability for success within the NASA culture.

Software Measurement
Recall that there is little evidence of collection and use of software measures throughout NASA. Collection and use ranged from non-existent to a few robust programs.

Software measurement is critical for project management and for the success of any software process improvement effort. Without measurement, change and improvement cannot be demonstrated. Here also, NASA might be able to leverage the few robust measurement programs to assist in the adoption of measurement by other NASA organizations. As in the case of reuse, applying domain-consistent, NASA-developed solutions to projects has the best chance for acceptance in the NASA culture.

In both examples, NASA and Center level policies could be put in place to encourage the reuse and software measurement programs by the projects. The existing positive examples of projects using reuse and software measurement could be packaged in a way that could be useful for other projects. In some cases, appropriate NASA and Center funding assistance could be applied to get the programs started. The projects themselves should then be responsible for setting their own project-specific goals, tailoring the packaged software improvement processes, and implementing them in a way that contributes positively to their projects.

Other baseline findings can be examined to extract similar observations and to make recommendations for improvement. In analyzing the baseline, software domains and organizational levels must be considered. First, consider software domains. Examining reuse in domains that perform repeated tasks, e.g., mission ground support software, would probably be more beneficial than examining reuse in the area of research software where most software is one-of-a-kind. Similarly, research software might not require much in the area of software measurement. When analyzing the baseline, identifying areas for improvement, applying the findings, and implementing changes, software domains must be considered.

Organizational levels also play a key role in analyzing and applying the findings. Higher organizational levels (e.g., NASA and Center level) should focus on encouraging local implementations via policy and funding assistance. Local projects should determine their own goals and devise an implementation of the software improvement area that fits their experience and domains.

RECOMMENDATIONS

Based on the findings to date, some recommendations can be made. First, since a significant portion of NASA's resources (both manpower and budget) is spent on software, each NASA Center and significant software organization should establish a software baseline.

Second, since project level standards are the most used and useful, NASA should focus on project and domain level standards, NOT on NASA-level standards.

Finally, NASA should assess the existing baseline to identify areas for software improvement. Based on the assessment, recommendations should be developed. At the very least, these recommendations should focus on software reuse and software measurement.

SUMMARY

This initial baseline of NASA software provides the answers to basic questions about NASA's software practices. It can provide insight for NASA to focus on potential areas of improvement. It also provides a snapshot in time to be used for future comparisons as changes are introduced and NASA's software process evolves.

This baseline is a first step toward continual software process improvement. It also must be the first of many baselines. As the Agency's process evolves, this baseline must be reexamined and updated to accurately characterize NASA's software practices at that point in time. Maintaining a baseline is critical to retain an ongoing understanding of NASA's software process and products. Without such understanding, improvements cannot be identified and continual software process improvement cannot be attained.

REFERENCES

1. *Profile of Software Within Code 500 at the Goddard Space Flight Center*, Hall, D. and McGarry, F., NASA-RPT-001-94, April 1994.

2. *Profile of Software at the Goddard Space Flight Center*, Hall, D., Sinclair, C., and McGarry, F., NASA-RPT-002-94, June 1994.

3. *Profile of Software Within the National Aeronautics and Space Administration*, Hall, D., Sinclair, C., Siegel, B., and Pajerski, R., NASA-RPT-xxx-95, Draft, January 1995.

4. *Profile of NASA Software Engineering: Lessons Learned from Building the Baseline*, Hall, D. and McGarry, F., Eighteenth Annual Software Engineering Workshop, SEL-93-003, December 1993.

ACRONYMS AND ABBREVIATIONS

CASE	Computer-Aided Software Engineering
Code 500	Mission Operations and Data Systems Directorate (at GSFC)
Code Q	Office of Safety and Mission Assurance (at NASA/Headquarters)
COTS	commercial-off-the-shelf
FY93	fiscal year 1993
GSFC	Goddard Space Flight Center
IRM	Information Resources Management
MSLOC	million source lines of code
NASA	National Aeronautics and Space Administration
SAIC	Science Applications International Corporation
SLOC	source lines of code

Profile of Software Engineering Within NASA

Craig C. Sinclair, SAIC
Kellyann Jeletic, NASA/GSFC

19th Annual Software Engineering Workshop
12/1/94

GOALS

Overall Goal:

- *Apply Software Engineering Laboratory (SEL) software process improvement approach to NASA as a whole*
- *Instill continual software process improvement throughout NASA*
- *Build specific recommendations for software process improvement within NASA*

Study Goal:

- *Establish the baseline of software and software engineering practices throughout NASA*

PURPOSE OF THE BASELINE

- To help NASA management understand the scope and extent of the software work within NASA
- To assist NASA management to see where they should focus future $$$ to improve software products and processes
- To assess the baseline for identification of candidate areas for software improvement

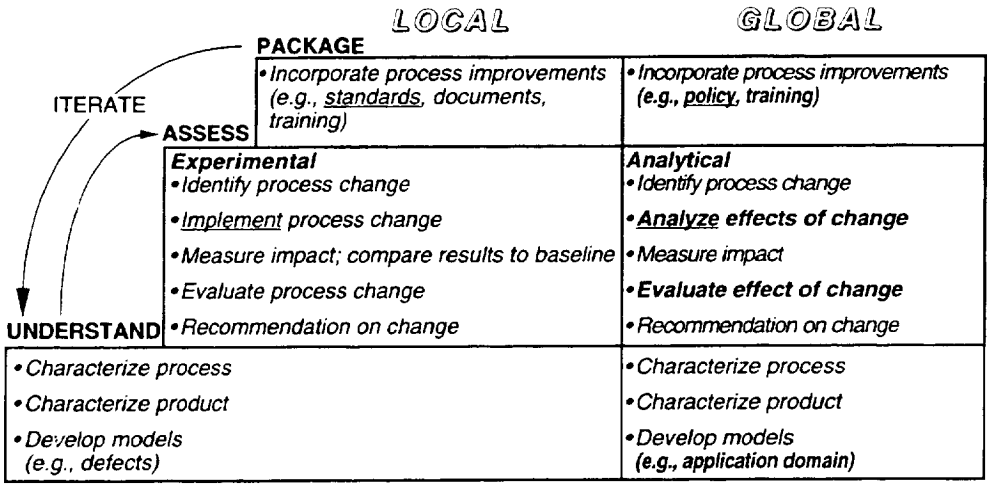
APPROACH

SEL Software Process Improvement Approach:

1) Understand (Baseline)
2) Assess
3) Package

There are some differences when applying the SEL approach to a ***global*** organizational level compared to a ***local*** organizational level

APPROACH - LOCAL VS. GLOBAL



Local ---> Experimental
Global ---> Analytical

ESTABLISH THE BASELINE

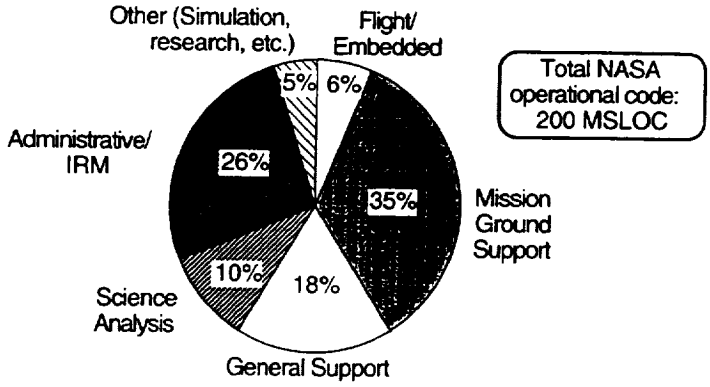
- Captured snapshot of FY93 attributes of:
  - NASA software (the product)
  - NASA's software engineering practices (the process)
- Data gathering methodology
  - Surveys, administered in person
  - Roundtable discussions
  - One on one interviews
  - Review of project documentation

Basic objective is to understand, not to judge right or wrong

- Next few charts describe the NASA Baseline
- Then we show how the baseline might be used

NASA SOFTWARE
PRODUCT CHARACTERISTICS

Amount of Software and Software Domains

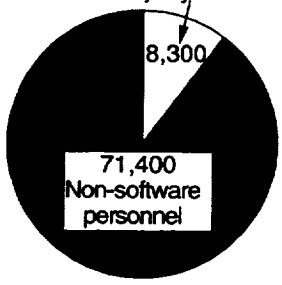


Largest software domains indicate where software improvement efforts could be focused

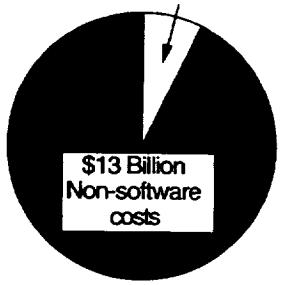
NASA SOFTWARE
PRODUCT CHARACTERISTICS

Software Staffing and Cost

More than 10% of NASA's 80,000 civil servants and support contractors were involved with software the majority of the time



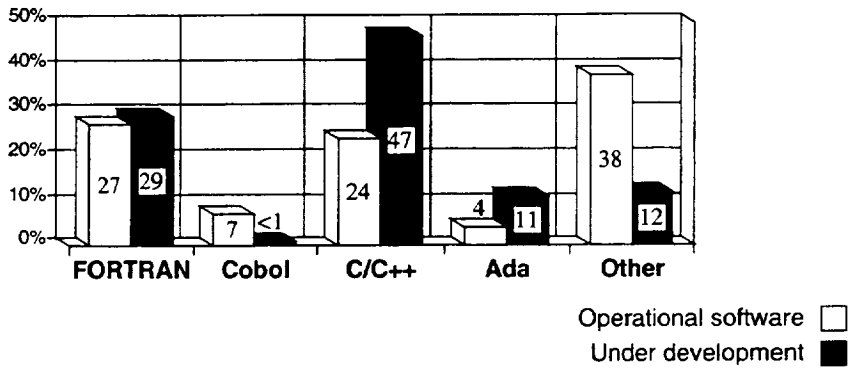
About 80% of NASA's software work was contracted to industry



NASA has a significant investment of manpower and budget in software

NASA SOFTWARE
PRODUCT CHARACTERISTICS

Language Preferences and Trends



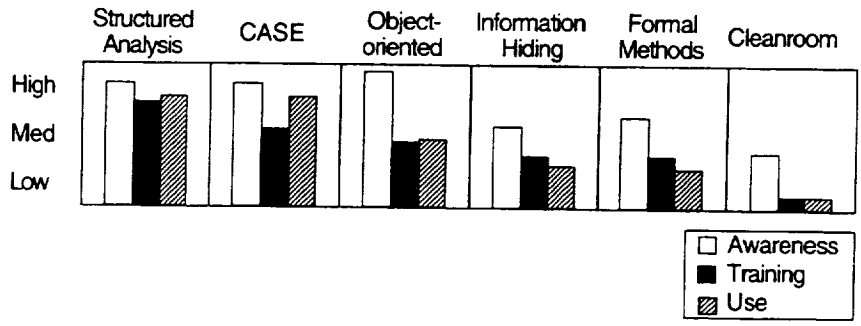
Findings may be used to focus training activities

NASA SOFTWARE
PROCESS CHARACTERISTICS

- ***Software Standards***
  - Project level were found to be most used and useful
  - Relative little usage by NASA personnel; widespread among contractors
- ***Requirements Management***
  - Unstable requirements are the biggest cause of schedule, budget, and code size growth problems
  - In general, requirements are not stable by preliminary design
- ***Software Reuse***
  - On average, about 15% of code is reused from one NASA project to another
  - Most focus is on code reuse; considerable variance in levels between Centers
- ***Software Metrics***
  - Little evidence of collection and use throughout NASA as a whole
  - Collection and use varied from non-existent to a few robust programs

NASA SOFTWARE *PROCESS CHARACTERISTICS*

Development Methodologies



NASA may be investing in "just in time" training

HOW BASELINE CAN BE USED

- To assess the baseline for **identification of candidate areas for software improvement**
- To **develop specific recommendations for implementation** of software improvement within NASA
- To **stimulate local implementation** of software improvement recommendations (bottom-up)

ASSESSING THE BASELINE
EXAMPLE 1

Measurement

- **Findings**
  - Collection and use varied from non-existent to a few robust programs
  - Little evidence of collection and use throughout NASA as a whole
- **Observations**
  - Software metrics need to be used for project management and to determine success of software improvement efforts
  - NASA could leverage the few robust metrics programs to assist the adoption of metrics by other NASA organizations

ASSESSING THE BASELINE
EXAMPLE 2

Reuse

- **Findings**
  - On average, about 15% of code is reused from one project to another
  - A few NASA organizations utilize software reuse as a normal part of their software development process
  - Overall, there were few defined approaches to reuse
- **Observations**
  - Organizations with software reuse (architecture and code) have made 75 - 80% reductions in cycle time and development cost
  - NASA could leverage the few robust programs to assist the adoption of software reuse by other NASA organizations

APPLYING THE FINDINGS

- Findings must be analyzed in terms of **software domains**
  - Science analysis software may not require much in the way of a metrics program
  - Software reuse may be most useful in domains that perform repeated tasks, such as mission ground support versus research software
- Findings must be analyzed in terms of the **organizational levels**
  - NASA-wide: top level policies
  - Center-wide: center level policies
  - Local organizations: implementation

BASELINING NASA SOFTWARE *RECOMMENDATIONS*

- **Each NASA Center and significant organization should baseline**, since more than 10% of NASA's budget is spent on software related activities.
- **NASA should focus on project level and domain standards,** NOT on NASA-level standards, since project standards were found to be the most used and useful.
- **NASA should assess the existing baseline to identify areas for software improvement.** Recommendations should be developed, including at least:
  - Software reuse
  - Software measurement

Appendix A: Workshop Attendees

Aalto, Juha-Markus, Nokia Telecommunications
Aaronson, Kenneth, Computer Sciences Corporation
Abrahamson, Shane, HQ AFC4A
Amad, Shahla, Hughes STX Corp.
Angevine, Jim, ALTA Systems, Inc.
Armentrout, Terry, Computer Sciences Corporation
Averill, Edward L., Edward Averill & Associates
Ayers, Everett, ARINC Research Corp.

Bailey, John, Software Metrics, Inc.
Barski, Renata M., AlliedSignal Technical Services Corp.
Basili, Victor R., University of Maryland
Bassman, Mitchell J., Computer Sciences Corporation
Beall, Shelley, Social Security Administration
Bean, Paul S., DSD Labs
Beierschmitt, Michael J., Loral AeroSys
Beifeld, David, Unisys Corp.
Bellamy, David, MITRE Corp.
Belvins, Brian E., Naval Surface Warfare Center
Bender, Jeff, University of Maryland
Beswick, Charlie A., Jet Propulsion Laboratory
Blagmon, Lowell E., Naval Center For Cost Analysis
Blankenship, Donald D., U.S. Air Force
Blatt, Carol, U.S. Census Bureau
Bobowiec, Paul W., COMPTEK Federal Systems
Bohn, Ilsa, SECON
Boland, Dillard, Computer Sciences Corporation
Boloix, Germinal, Ecole Polytechnique de Montreal
Bond, Jack, DoD
Borger, Mark W., Software Engineering Institute
Borkowski, Karen, SECON
Bowers, Allan, Loral AeroSys
Boyd, Andrew, U.S. Air Force
Bozoki, George J., Lockheed Missiles & Space Co., Inc.
Branson, Jim, American Systems Corp.
Bredeson, Mimi, Space Telescope Science Institute
Britt, Joan J., MITRE Corp.
Brown, Cindy, Computer Sciences Corporation
Brown, Richard A., Boeing Information Services
Bunch, Aleda, Social Security Administration
Burke, Karen, Loral Federal Systems
Burkhart, Michael R., IIT Research Institute
Burr, Carol, Computer Sciences Corporation
Busby, Mary, Loral Federal Systems Group
Buseman, Bill, EER Systems Corp.

Caldiera, Gianluigi, University of Maryland
Callahan, Jack, NASA/WVU IV&V Facility
Canfield, Roger A., DoD Joint Spectrum Center
Carlin, Catherine M., Veterans Benefits Administration
Carlisle, Candace, NASA/GSFC
Carlson, Randall, NSWCDD
Carmichael, Kevin R., NASA/LeRC
Carmody, Cora L., PRC, Inc.
Celentano, Al, Social Security Administration
Centafont, Noreen, DoD
Chapman, Robert, EG&G WASC, Inc.
Chen, Lily Y., AlliedSignal Technical Services Corp.
Chiverella, Ron, Pennsylvania Blue Shield
Chu, Richard, Loral AeroSys
Clamons, Jim, Harris Corp.
Clark, Carole A., Veterans Benefits Administration
Clark, James D., Naval Surface Warfare Center
Cochrane, Gail, TRW
Cohen, Judy, Jet Propulsion Laboratory
Condon, Steven E., Computer Sciences Corporation
Cooke, Terrence, Eagle Systems, Inc.
Coon, Richard W., Computer Sciences Corporation
Corbin, Regina, Social Security Administration
Cover, Donna, Computer Sciences Corporation
Cowan, Marcia, Loral AeroSys
Coyne, Edward J., SETA Corp.
Crawford, Art, Mantech Services Corp.
Crowder, Grace, University of Maryland-Baltimore Co.
Cuesta, Ernesto, Computer Sciences Corporation

Daku, Walter, Unisys Corp.
Daniele, Carl J., NASA/LeRC
Deutsch, Michael S., Hughes Applied Info. Systems, Inc.
Devasirvatham, Josiah, CARDS/D.N. American
DiNunno, Donn, Computer Sciences Corporation
Diaczun, Paul, Tidewater Consultants, Inc.
Dickson, Charles H., USDA
Diskin, David H., Defense Information Systems Agency
Dixson, Paul E., Loral AeroSys
Doland, Jerry T., Computer Sciences Corporation
Dolphin, Leonard, ALTA Systems, Inc.
Donnelly, Laurie M., AlliedSignal Technical Services Corp.
Dotseth, Margaret, Computer Sciences Corporation
Dowen, Andrew Z., Jet Propulsion Laboratory

Drake, Anthony M., Computer Sciences Corporation
DuBard, James E., Computer Sciences Corporation
Dudash, Ed, Naval Surface Warfare Center
Duncan, Scott P., BELLCORE
Dyer, Michael, DYCON Systems

Edelson, Robert, Jet Propulsion Laboratory
Eichmann, David, University of Houston-Clear Lake
Elliott, Geoffrey, Raytheon Engineers & Contractors
Ellis, Walter J., Software Process & Metrics
Esker, Linda J., Computer Sciences Corporation
Estep, James, WVHTC Foundation
Ettore de Cesare, Luciano, University of Maryland
Evans, Calvin Wayne, DoD

Farrell-Presley, Mary Ann, Applied Systems Technology, Inc.
Fedor, Gregory A., ADF, Inc.
Ferguson, Frances, Stanford Telecommunications, Inc.
Fernandes, Vernon, Computer Sciences Corporation
Finley, Doug, Unisys Corp.
Flora, Jackie, Social Security Administration
Flynn, Nick, Mantech Services Corp.
Forsythe, Ron, NASA/Wallops Flight Facility
Frahm, Mary J., Computer Sciences Corporation
Froehlich, Donna, IIT Research Institute
Futcher, Joseph M., Naval Surface Warfare Center

Gaddis, John B., CARDS/DSD Laboratories
Gallagher, Barbara F., DoD
Gallo, Al, Unisys Corp.
Gantzer, Don
Garlick, Teri, Pennsylvania Blue Shield
Getzen, Phil, DIA
Gill, David C., TECHSOFT, Inc.
Gluck, Raymond M., FCDSSA NSWC PHD ECO
Godfrey, Sally, NASA/GSFC
Golden, John R., Information Technologies
Goon, Linda, QA Consultant
Gosnell, Arthur B., U.S. Army Missile Command
Gover, Gary, Veterans Benefits Administration
Graffman, Ira, Hughes STX Corp.
Grano, Vince, Hughes STX Corp.
Grant, Jr., Ralph D., New Technology, Inc.
Grech, Neill, Computer Sciences Corporation
Green, David S., Computer Sciences Corporation
Green, Leonard, DoD
Green, Scott, NASA/GSFC
Greene, James S., U.S. Air Force
Gregory, Judith A., NASA/MSFC
Gregory, Shawna C., MITRE Corp.
Griffin, Brenda, McDonnell Douglas Space Systems Co.
Gwennet, Lance, Systems Research & Application Corp.
Gwynn, Thomas R., Computer Sciences Corporation

Haislip, William, U.S. Army
Hall, Angela, Computer Sciences Corporation
Hall, Charley, SECON
Hall, Dana L., SAIC
Hall, Ken R., Computer Sciences Corporation
Han, Cecilia, Jet Propulsion Laboratory
Hannon, Nanci, DISA/JIEQ/TBEC
Hanson, Pauline, U.S. Census Bureau
Harris, Barbara A., USDA
Hartzler, Ellen
Heintzelman, Clinton L., U.S. Air Force
Heller, Gerard H., Computer Sciences Corporation
Hendrzak, Gary, Booz, Allen & Hamilton, Inc.
Henry, Joel, East Tennessee State University
Herbsleb, Jim, Software Engineering Institute
Heuser, Wolfgang, Daimlerbenz, Research
Higgins, Herman A., DoD
Hoffmann, Kenneth, Ryan Computer Systems, Inc.
Holloway, Mel, Joint Warfare Analysis Center
Holmes, Barbara, CRM
Holmes, Joseph A., IRS
Hopkins, Clifford R., DoD
Hoppe, William C., U.S. Army/ISSC
Howard, William H., Unisys Corp.
Hultgren, Ed, DoD
Hung, Chaw-Kwei, Jet Propulsion Laboratory

Ingram, Darryl R., Maryland Tectrix, Inc.

Jefferson, Pat, IRS
Jeletic, Jim, NASA/GSFC
Jeletic, Kellyann, NASA/GSFC
Johnson, Pat, NASA/GSFC
Johnson, Temp, Hughes-STX
Jones, Christopher C., IIT Research Institute
Jones, Linda J., TRW
Jones, Lori M., Tidewater Consultants, Inc.
Jordan, Gary, Unisys Corp.
Jordano, Tony J., SAIC

Kalin, Jay, Loral AeroSys
Karlsson, Even-Andre, Q-Labs
Keeler, Kristi L., Computer Sciences Corporation
Kelly, John C., Jet Propulsion Laboratory
Kemp, Kathryn M., Office of Safety & Mission Assurance

Kester, Rush W., DC SIGAda
Kierk, Isabella K., NASA/JPL
Kim, Robert D., Computer Sciences Corporation
Kim, Yong-Mi, University of Maryland
Kistler, David M., Computer Sciences Corporation
Klein, Jr., Gerald A., QSS Group, Inc.
Kleptin, Laurie, Unisys Corp.
Knight, Colette A., NSWC
Knight, John C., University of Virginia
Kontio, Jyrki, University of Maryland
Kotov, Alexei, Oregon Graduate Institute
Kraft, Steve, NASA/GSFC
Kronisch, Mark, U.S. Census Bureau
Kuhle, Sherry, Booz, Allen & Hamilton
Kuhne, Fran, Social Security Administration
Kushner, Todd R., CTA, Inc.
Kyser, Frances, Nichols Research Corp.

LaPorte, Claude Y., Oerlikon Aerospace
Laitenberger, Oliver
Lamia, Walter, Software Engineering Institute
Landis, Linda C., Computer Sciences Corporation
Lane, Allan C., AlliedSignal Technical Services Corp.
Langston, James H., Computer Sciences Corporation
Lawrence, Raymond L., Loral AeroSys
Lay, Barbara N., Motorola, Inc.
Lehman, Meir M., Imperial College of Science
Levesque, Michael, Jet Propulsion Laboratory
Levinson, Laurie H., NASA/LeRC
Levitt, Dave S., Computer Sciences Corporation
Leydorf, Steven M., IIT Research Institute
Li, Rorry, ORACLE
Libson, Ted, Boeing Information Services, Inc.
Liebermann, Roxanne, U.S. Census Bureau
Liebrecht, Paula L., Computer Sciences Corporation
Lindsay, Scott, Government Systems, Inc.
Lindsey, Brad, IIT Research Institute
Lipsett, Bill, IRS
Liu, Jean C., Computer Sciences Corporation
Livingston, Karen, IIT Research Institute
Loesh, Bob E., Software Engineering Sciences Corporation
Lott, Christopher M., University of Kaiserlautern
Loy, Patrick H., Loy Consulting, Inc.
Lubash, Steven, ITT Avionics
Lucas, Janice P., Financial Management Service
Luczak, Ray W., Computer Sciences Corporation
Lutz, Robyn R., Iowa State University
Luu, Kequan, NASA/GSFC
Lynnes, Chris, NASA/GSFC
Lyons, Howarette P., AFPCA/GADB

Mabry, Bobbie L., DoD
Mahmud, Maruf, IRS
Marciniak, John J., Kaman Sciences Corporation
Marijarvi, Jukka, Nokia Cellular Systems
Martinez, Bill, Loral Federal Systems
Masters, Wen C., Jet Propulsion Laboratory
Maury, Jesse, Omitron, Inc.
Mazzola, Ray, Loral AeroSys
McCafferty, Brian, XonTech, Inc.
McConnell, Andrew, ASSET
McCreary, Faith A., Jet Propulsion Laboratory
McGarry, Frank E., Computer Sciences Corporation
McGrane, Janet K., U.S. Census Bureau
McHenry, Ron, Hughes STX Corp.
McIlwraith, Isabel, IRS
McKay, Judith A., U.S. Census Bureau
McNeill, Justin F., Jet Propulsion Laboratory
McSharry, Maureen, Computer Sciences Corporation
Melo, Walcelio L., University of Maryland
Mendonca, Manoel G., University of Maryland
Methia, Linda, NASA/HQ
Miller, Dave, COMSO, Inc.
Mills, Marilyn K., Computer Sciences Corporation
Minninger, John R., DoD
Moleski, Laura, CRM
Moore, Paula, NOAA/SPOx3
Moore, Robin W., Air Force Pentagon Communication Agency
Morgan, Pam, IIT Research Institute
Morusiewicz, Linda M., Computer Sciences Corporation
Moseley, Patricia, DoD
Murphy, Thomas, Siemans Corporate Research, Inc.
Myers, Philip I., Computer Sciences Corporation

Narrow, Bernie, AlliedSignal Technical Services Corp.
Neilan, Hester, Jet Propulsion Laboratory
Nestlerode, Howard, Unisys Corp.
Neuman, Harriet J., FAA
New, Ronald, NOAA
Nichols, Dan, CARDS/EWA
Nickerson, Brenda, Loral Federal Systems Group
Niemela, Mary, SECON
Nokovich, Sandra L., U.S. Census Bureau
Norcio, Tony F., University of Maryland-Baltimore Co.
Norton, William F., FAA
Nusenoff, Ron, Loral Software Productivity Lab

O'Brien, Robert L., Unisys Corp.
O'Donnell, Charlie, ECA, Inc.
O'Neill, Don, Software Engineering Consultant
Obenski, Dave, DoD
Offer, Regina W., AFPCA/6ADB
Ohlmacher, Jane A., Social Security Administration

Page, Gerald T., Computer Sciences Corporation
Pailen, William, Pailen-Johnson Associates, Inc.
Pajerski, Rose, NASA/GSFC
Paletar, Teresa L., Naval Air Warfare Center
Panlilio-Yap, Nikki M., Loral Federal Systems Group
Parker-Gates, Linda, Software Productivity Consortium
Passaretti, Gennaro, Consorzio SOFTIN
Patton, K. Kay, Computer Sciences Corporation
Pavnica, Paul, Treasury/Fincen
Peeples, Ron L., Intermetrics
Perry, Howard, Computer Sciences Corporation
Peterman, David, Texas Instruments
Petro, Jim, EWA, Inc.
Pettengill, Nathan, Martin Marietta
Phan, Quyen, BTG, Inc.
Polk, Bryant, Systems Research & Applications Corp.
Porter, Adam A., University of Maryland
Pottinger, David L., SAIC
Powers, Larry T., Unisys Corp.
Pressley, Coretta T., DoD
Provenza, Clint, Booz, Allen & Hamilton, Inc.

Quann, Eileen S., Fastrak Training, Inc.

Radley, Charles, Raytheon - EBASCO
Ramsey, Jack, Pennsylvania Blue Shield
Raney, Dale L., TRW
Redding, John L., Defense Information Systems Agency
Reeb, Jim, U.S. Army MICOM
Reed, James J., Karman Sciences Corporation
Regardie, Myrna L., Computer Sciences Corporation
Reitzel, Morris, DoD
Rifkin, Stan, Master Systems, Inc.
Riihinen, Jaakko, Nokia Cellular Systems
Risser, Gary E., Veterans Benefits Administration
Rizer, Stephani, NAWC-AD
Rodenas, Albe, ALTA Systems, Inc.
Rohr, John A., Jet Propulsion Laboratory
Rohrer, Amos M., SYSCON Corp.
Rosenberg, Linda H., Unisys Corp.
Roy, Dan M., STP&P
Russell, Wayne M., GTE
Rymer, John, Loral Federal Systems Group

Sabolish, George J., NASA IV&V Facility
Saisi, Robert O., DSD Laboratories, Inc.
Samadi, Shahin, NASA/GSFC
Samuels, George, Social Security Administration
Santiago, Richard, Jet Propulsion Laboratory
Satyen, Uma D., MITRE Corp.
Sawanobori, Tina, Computer Sciences Corporation
Schellhase, Ronald J., Computer Sciences Corporation
Schilling, Mark, Informatics, Inc.
Schmidt, Evan, EWA
Schrom, Roberta, Hughes Training, Inc.
Schuler, Pat M., NASA/LaRC
Schwartz, Benjamin L., Consultant
Schwarz, Henry, NASA/KSC
Scott, Donna, PRC, Inc.
Scott, Rhonda M., Loral Federal Systems Group
Seaman, Carolyn B., University of Maryland
Seiber, Dwayne, OAO Corp.
Seidewitz, Ed, NASA/GSFC
Sharma, Jagdish, NOAA
Sharma, Khem, Computer Sciences Corporation
Sheckler, John D., AlliedSignal Technical Services Corp.
Sheppard, Sylvia B., NASA/GSFC
Shi-Hung Hsueh, Bryan, University of Maryland
Short, Cathy, IRS
Singer, Carl A., BELLCORE
Six, Richard E., DoD
Siy, Harvey, University of Maryland
Slaton, Gordon, U.S. Air Force
Slonim, Jacob, Centre for Advanced Studies - IBM
Smith, Donald, NASA/GSFC
Smith, George F., Space & Naval Warfare Systems Command
Smith, Sharon, Loral Federal Systems Group
Smith, Vivian A., FAA
Sohmer, Robert, RAM Engineering Associates
Solomon, Carl A., Hughes-STX
Sova, Don, NASA/HQ
Squire, Jon S., Westinghouse
Squires, Burton E., Orion Scientific, Inc.
Ssemwogerere, Joe, Hughes STX Corp.
Stamboulis, Peter, SECON
Stang, David, ADF, Inc.
Stark, Michael, NASA/GSFC
Staub, Jane B., Tidewater Consultants, Inc.
Steger, Warren L., Computer Sciences Corporation
Steinberg, Sandee, Computer Sciences Corporation
Stoddard, Robert W., Texas Instruments

Summa, Robert L., Computer Sciences Corporation
Suryani Jamin Tung, Angela, AlliedSignal Technical Services Corp.
Szulewski, Paul A., C.S. Draper Labs, Inc.

Tasaki, Keiji, NASA/GSFC
Tesoriero, Roseanne, University of Maryland
Thomas, William, MITRE Corp.
Thomason, Clarke, Pailen-Johnson Associates, Inc.
Thompson, Sid, Unisys Corp.
Thornton, Thomas H., Jet Propulsion Laboratory
Tichenor, Charley, IRS
Trammell, Carmen J., University of Tennessee
Trible, Sue, GDE Systems, Inc.
Turcheck, Susan, Loral Federal Systems
Twombly, Mark A., Computer Based Systems, Inc.

Ulery, Bradford T., MITRE Corp.

Valdivia, Aaron, IIT Research Institute
Valett, Jon, NASA/GSFC
Van Verth, Patricia B., Canisius College
VanMeter, Charlene L., DoD
Vaughan, Frank R., RAM Engineering Associates
Vause, David G., Loral Federal Systems Group
Vernacchio, Al, NASA/GSFC
Viehman, Ralph, NASA/GSFC
Voigt, David, AlliedSignal Technical Services Corp.

Wald, Lawrence, NASA/LeRC
Waligora, Sharon R., Computer Sciences Corporation
Wallace, Dolores, NIST
Waterman, Robert E., Unisys Corp.
Webb, Hattie R., Naval Surface Warfare Center
Weiss, Sandy L., IIT Research Institute
Wenneson, Gregory J., Sterling Software, Inc.
West, Tim, DIA
Weston, William N., NASA/GSFC
Weszka, Joan, Loral Federal Systems Group
Wetherholt, Martha S., NASA/LeRC
Whisenand, Tom, Social Security Administration
White, Gilbert, NASA/HQ
Whittlesey, Raquel S., ADF, Inc.
Wiggers, Grace E., Computer Sciences Corporation
Wika, Kevin G., University of Virginia
Wilkins, Lori H., New-Bold Enterprises
Willey, Allan L., Motorola, Inc.
Williams, Bonnie, Computer Sciences Corporation
Wilson, Robert K., Jet Propulsion Laboratory
Wingfield, Charles G., DoD
Wolfish, Hannah K., Social Security Administration
Wood, Dick, Computer Sciences Corporation
Wu, Sabina L., IIT Research Institute

Yakstis, Louis C., Computer Sciences Corporation
Yassini, Siamak, NASA/HQ
York, Lee, Veterans Benefits Administration
Youman, Charles, SETA Corp.
Young, Andy, Young Engineering Services, Inc.
Young, Jeff, Lawrence Livermore National Lab
Yu, Phil, TRW

Zaveler, Saul, U.S. Air Force
Zeitvogel, Barney, SECON
Zelkowitz, Marv, University of Maryland
Zimet, Beth, Computer Sciences Corporation

Appendix B: Standard Bibliography of SEL Literature

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, *Proceedings From the First Summer Software Engineering Workshop,* August 1976

SEL-77-002, *Proceedings From the Second Summer Software Engineering Workshop,* September 1977

SEL-78-005, *Proceedings From the Third Summer Software Engineering Workshop,* September 1978

SEL-78-006, *GSFC Software Engineering Research Requirements Analysis Study,* P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, *Applicability of the Rayleigh Curve to the SEL Environment,* T. E. Mapp, December 1978

SEL-78-302, *FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3),* W. J. Decker, W. A. Taylor, et al., July 1986

SEL-79-002, *The Software Engineering Laboratory: Relationship Equations,* K. Freburger and V. R. Basili, May 1979

SEL-79-004, *Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment,* C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, *Proceedings From the Fourth Summer Software Engineering Workshop,* November 1979

SEL-80-002, *Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation,* W. J. Decker and C. E. Goorevich, May 1980

SEL-80-005, *A Study of the Musa Reliability Model,* A. M. Miller, November 1980

SEL-80-006, *Proceedings From the Fifth Annual Software Engineering Workshop,* November 1980

SEL-80-007, *An Appraisal of Selected Cost/Resource Estimation Models for Software Systems,* J. F. Cook and F. E. McGarry, December 1980

SEL-80-008, *Tutorial on Models and Metrics for Software Management and Engineering*, V. R. Basili, 1980

SEL-81-011, *Evaluating Software Development by Analysis of Change Data*, D. M. Weiss, November 1981

SEL-81-012, *The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems*, G. O. Picasso, December 1981

SEL-81-013, *Proceedings of the Sixth Annual Software Engineering Workshop*, December 1981

SEL-81-014, *Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL)*, A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, *Guide to Data Collection*, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-110, *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-305, *Recommended Approach to Software Development*, L. Landis, S. Waligora, F. E. McGarry, et al., June 1992

SEL-81-305SP1, *Ada Developers' Supplement to the Recommended Approach*, R. Kester and L. Landis, November 1993

SEL-82-001, *Evaluation of Management Measures of Software Development*, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, *Collected Software Engineering Papers: Volume 1*, July 1982

SEL-82-007, *Proceedings of the Seventh Annual Software Engineering Workshop*, December 1982

SEL-82-008, *Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory*, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, *FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1)*, W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, *Glossary of Software Engineering Laboratory Terms*, T. A. Babst, M. G. Rohleder, and F. E. McGarry, October 1983

SEL-82-1306, *Annotated Bibliography of Software Engineering Laboratory Literature*, D. Kistler, J. Bristow, and D. Smith, November 1994

SEL-83-001, *An Approach to Software Cost Estimation*, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, *Measures and Metrics for Software Development*, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983

SEL-83-007, *Proceedings of the Eighth Annual Software Engineering Workshop*, November 1983

SEL-83-106, *Monitoring Software Development Through Dynamic Variables (Revision 1)*, C. W. Doerflinger, November 1989

SEL-84-003, *Investigation of Specification Measures for the Software Engineering Laboratory (SEL)*, W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, *Proceedings of the Ninth Annual Software Engineering Workshop*, November 1984

SEL-84-101, *Manager's Handbook for Software Development (Revision 1)*, L. Landis, F. E. McGarry, S. Waligora, et al., November 1990

SEL-85-001, *A Comparison of Software Verification Techniques*, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, *Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team*, R. Murphy and M. Stark, October 1985

SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985

SEL-85-004, *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics*, R. W. Selby, Jr., and V. R. Basili, May 1985

SEL-85-005, *Software Verification and Testing*, D. N. Card, E. Edwards, F. McGarry, and C. Antle, December 1985

SEL-85-006, *Proceedings of the Tenth Annual Software Engineering Workshop*, December 1985

SEL-86-001, *Programmer's Handbook for Flight Dynamics Software Development*, R. Wood and E. Edwards, March 1986

SEL-86-002, *General Object-Oriented Software Development*, E. Seidewitz and M. Stark, August 1986

SEL-86-003, *Flight Dynamics System Software Development Environment (FDS/SDE) Tutorial*, J. Buell and P. Myers, July 1986

SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986

SEL-86-005, *Measuring Software Design*, D. N. Card et al., November 1986

SEL-86-006, *Proceedings of the Eleventh Annual Software Engineering Workshop*, December 1986

SEL-87-001, *Product Assurance Policies and Procedures for Flight Dynamics Software Development*, S. Perry et al., March 1987

SEL-87-002, *Ada® Style Guide (Version 1.1)*, E. Seidewitz et al., May 1987

SEL-87-003, *Guidelines for Applying the Composite Specification Model (CSM)*, W. W. Agresti, June 1987

SEL-87-004, *Assessing the Ada® Design Process and Its Implications: A Case Study*, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987

SEL-87-010, *Proceedings of the Twelfth Annual Software Engineering Workshop*, December 1987

SEL-88-001, *System Testing of a Production Ada Project: The GRODY Study*, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988

SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby and L. Esker, December 1988

SEL-88-004, *Proceedings of the Thirteenth Annual Software Engineering Workshop*, November 1988

SEL-88-005, *Proceedings of the First NASA Ada User's Symposium*, December 1988

SEL-89-002, *Implementation of a Production Ada Project: The GRODY Study*, S. Godfrey and C. Brophy, September 1989

SEL-89-004, *Evolution of Ada Technology in the Flight Dynamics Area: Implementation/Testing Phase Analysis*, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, *Lessons Learned in the Transition to Ada From FORTRAN at NASA/Goddard*, C. Brophy, November 1989

SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989

SEL-89-007, *Proceedings of the Fourteenth Annual Software Engineering Workshop*, November 1989

SEL-89-008, *Proceedings of the Second NASA Ada Users' Symposium*, November 1989

SEL-89-103, *Software Management Environment (SME) Concepts and Architecture (Revision 1)*, R. Hendrick, D. Kistler, and J. Valett, September 1992

SEL-89-301, *Software Engineering Laboratory (SEL) Database Organization and User's Guide (Revision 3)*, L. Morusiewicz, February 1995

SEL-90-001, *Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide*, M. Buhler, K. Pumphrey, and D. Spiegel, March 1990

SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis*, S. Green et al., March 1990

SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL)*, L. O. Jun and S. R. Valett, June 1990

SEL-90-004, *Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary*, T. McDermott and M. Stark, September 1990

SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990

SEL-90-006, *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November 1990

SEL-91-001, *Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules*, W. Decker, R. Hendrick, and J. Valett, February 1991

SEL-91-003, *Software Engineering Laboratory (SEL) Ada Performance Study Report*, E. W. Booth and M. E. Stark, July 1991

SEL-91-004, *Software Engineering Laboratory (SEL) Cleanroom Process Model*, S. Green, November 1991

SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991

SEL-91-006, *Proceedings of the Sixteenth Annual Software Engineering Workshop*, December 1991

SEL-91-102, *Software Engineering Laboratory (SEL) Data and Information Policy (Revision 1)*, F. McGarry, August 1991

SEL-92-001, *Software Management Environment (SME) Installation Guide*, D. Kistler and K. Jeletic, January 1992

SEL-92-002, *Data Collection Procedures for the Software Engineering Laboratory (SEL) Database*, G. Heller, J. Valett, and M. Wild, March 1992

SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992

SEL-92-004, *Proceedings of the Seventeenth Annual Software Engineering Workshop*, December 1992

SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993

SEL-93-002, *Cost and Schedule Estimation Study Report*, S. Condon, M. Regardie, M. Stark, et al., November 1993

SEL-93-003, *Proceedings of the Eighteenth Annual Software Engineering Workshop*, December 1993

SEL-94-001, *Software Management Environment (SME) Components and Algorithms*, R. Hendrick, D. Kistler, and J. Valett, February 1994

SEL-94-002, *Software Measurement Guidebook*, M. Bassman, F. McGarry, R. Pajerski, July 1994

SEL-94-003, *C Style Guide*, J. Doland and J. Valett, August 1994

SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994

SEL-94-005, *An Overview of the Software Engineering Laboratory*, F. McGarry, G. Page, V. Basili, et al., December 1994

SEL-94-006, *Proceedings of the Nineteenth Annual Software Engineering Workshop*, December 1994

SEL-RELATED LITERATURE

[10]Abd-El-Hafiz, S. K., V. R. Basili, and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components," *Proceedings of the IEEE Conference on Software Maintenance-1991 (CSM 91)*, October 1991

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," *Program Transformation and Programming Environments*. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[8]Bailey, J. W., and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability," *Proceedings of the Eighth Annual National Conference on Ada Technology*, March 1990

[10]Bailey, J. W., and V. R. Basili, "The Software-Cycle Model for Re-Engineering and Reuse," *Proceedings of the ACM Tri-Ada 91 Conference*, October 1991

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," *ASME Advances in Computer Technology*, January 1980, vol. 1

Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering*. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," *Proceedings of the First Pan-Pacific Computer Conference*, September 1985

[7]Basili, V. R., *Maintenance = Reuse-Oriented Software Development*, University of Maryland, Technical Report TR-2244, May 1989

[7]Basili, V. R., *Software Development: A Paradigm for the Future*, University of Maryland, Technical Report TR-2263, June 1989

[8]Basili, V. R., "Viewing Maintenance of Reuse-Oriented Software Development," *IEEE Software*, January 1990

[1]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[9]Basili, V. R., G. Caldiera, and G. Cantone, "A Reference Architecture for the Component Factory,"*ACM Transactions on Software Engineering and Methodology*, January 1992

[10]Basili, V., G. Caldiera, F. McGarry, et al., "The Software Engineering Laboratory—An Operational Software Experience Factory," *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE 92)*, May 1992

[1]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

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

[3]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," *Proceedings of the International Computer Software and Applications Conference*, October 1985

[4]Basili, V. R., and D. Patnaik, *A Study on Fault Prediction and Reliability Assessment in the SEL Environment*, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," *Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March 1981

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P—A Prototype Expert System for Software Engineering Management," *Proceedings of the IEEE/MITRE Expert Systems in Government Symposium*, October 1985

Basili, V. R., and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*. New York: IEEE Computer Society Press, 1979

[5]Basili, V. R., and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proceedings of the 9th International Conference on Software Engineering*, March 1987

[5]Basili, V. R., and H. D. Rombach, "TAME: Tailoring an Ada Measurement Environment," *Proceedings of the Joint Ada Conference*, March 1987

[5]Basili, V. R., and H. D. Rombach, "TAME: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

[6]Basili, V. R., and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, June 1988

[7]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment*, University of Maryland, Technical Report TR-2158, December 1988

[8]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes*, University of Maryland, Technical Report TR-2446, April 1990

[9]Basili, V. R., and H. D. Rombach, "Support for Comprehensive Reuse," *Software Engineering Journal*, September 1991

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[3]Basili, V. R., and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology," *Proceedings of the NATO Advanced Study Institute*, August 1985

[5]Basili, V. R., and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[9]Basili, V. R., and R. W. Selby, "Paradigms for Experimentation and Empirical Studies in Software Engineering," *Reliability Engineering and System Safety*, January 1991

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, July 1986

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983

[2]Basili, V. R., and D. M. Weiss, *A Methodology for Collecting Valid Software Engineering Data*, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," *Proceedings of the Second Software Life Cycle Management Workshop*, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," *Computers and Structures*, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," *Proceedings of the Third International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1978

Bassman, M. J., F. McGarry, and R. Pajerski, *Software Measurement Guidebook*, NASA-GB-001-94, Software Engineering Program, July 1994

[9]Booth, E. W., and M. E. Stark, "Designing Configurable Software: COMPASS Implementation Concepts," *Proceedings of Tri-Ada 1991*, October 1991

[10]Booth, E. W., and M. E. Stark, "Software Engineering Laboratory Ada Performance Study—Results and Implications," *Proceedings of the Fourth Annual NASA Ada User's Symposium*, April 1992

[10]Briand, L. C., and V. R. Basili, "A Classification Procedure for the Effective Management of Changes During the Maintenance Process," *Proceedings of the 1992 IEEE Conference on Software Maintenance (CSM 92)*, November 1992

[10]Briand, L. C., V. R. Basili, and C. J. Hetmanski, "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

[11]Briand, L. C., V. R. Basili, and C. J. Hetmanski, *Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components*, University of Maryland, Technical Report TR-3048, March 1993

[12]Briand, L. C., V. R. Basili, Y. Kim, and D. R. Squire, "A Change Analysis Process to Characterize Software Maintenance Projects", *Proceedings of the International Conference on Software Maintenance*, September 1994

[9]Briand, L. C., V. R. Basili, and W. M. Thomas, *A Pattern Recognition Approach for Software Engineering Data Analysis*, University of Maryland, Technical Report TR-2672, May 1991

[11]Briand, L. C., S. Morasca, and V. R. Basili, "Measuring and Assessing Maintainability at the End of High Level Design," *Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM 93)*, November 1993

[12]Briand, L., S. Morasca, and V. R. Basili, *Defining and Validationg High-Level Design Metrics*, University of Maryland, Technical Report TR-3301, June 1994

[11]Briand, L. C., W. M. Thomas, and C. J. Hetmanski, "Modeling and Managing Risk Early in Software Development," *Proceedings of the Fifteenth International Conference on Software Engineering (ICSE 93)*, May 1993

[5]Brophy, C. E., W. W. Agresti, and V. R. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," *Proceedings of the Joint Ada Conference*, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," *Proceedings of the Washington Ada Technical Conference*, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

[3]Card, D. N., "A Software Technology Evaluation Program," *Annais do XVIII Congresso Nacional de Informatica*, October 1985

[5]Card, D. N., and W. W. Agresti, "Resolving the Software Science Anomaly," *Journal of Systems and Software*, 1987

[6]Card, D. N., and W. W. Agresti, "Measuring Software Design Complexity," *Journal of Systems and Software*, June 1988

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, February 1986

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D. N., F. E. McGarry, and G. T. Page, "Evaluating Software Engineering Technologies," *IEEE Transactions on Software Engineering*, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," *ACM Software Engineering Notes*, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," *Proceedings of the Seventh International Computer Software and Applications Conference*. New York: IEEE Computer Society Press, 1983

Doubleday, D., *ASAP: An Ada Static Source Code Analyzer Program*, University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S., and C. Brophy, "Experiences in the Implementation of a Large Ada Project," *Proceedings of the 1988 Washington Ada Symposium*, June 1988

[5]Jeffery, D. R., and V. Basili, *Characterizing Resource Data: A Model for Logical Association of Software Data*, University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," *Proceedings of the Tenth International Conference on Software Engineering*, April 1988

[11]Li, N. R., and M. V. Zelkowitz, "An Information Model for Use in Software Management Estimation and Prediction," *Proceedings of the Second International Conference on Information Knowledge Management*, November 1993

[5]Mark, L., and H. D. Rombach, *A Meta Information Base for Software Engineering*, University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L., and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[5]McGarry, F. E., and W. W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," *Proceedings of the Sixth Washington Ada Symposium (WADAS)*, June 1989

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," *Proceedings of the Hawaiian International Conference on System Sciences*, January 1985

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," *Proceedings of the Eighth International Computer Software and Applications Conference*, November 1984

[12]Porter, A. A., L. G. Votta, Jr., and V. R. Basili, *Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment*, University of Maryland, Technical Report TR-3327, July 1994

[5]Ramsey, C. L., and V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management," *IEEE Transactions on Software Engineering*, June 1989

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, March 1987

[8]Rombach, H. D., "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990

[9]Rombach, H. D., "Software Reuse: A Key to the Maintenance Problem," *Butterworth Journal of Information and Software Technology*, January/February 1991

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," *Proceedings From the Conference on Software Maintenance*, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[7]Rombach, H. D., and B. T. Ulery, *Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL*, University of Maryland, Technical Report TR-2252, May 1989

[10]Rombach, H. D., B. T. Ulery, and J. D. Valett, "Toward Full Life Cycle Control: Adding Maintenance Measurement to the SEL," *Journal of Systems and Software*, May 1992

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," *Proceedings of the CASE Technology Conference*, April 1988

[9]Seidewitz, E., "Object-Oriented Programming Through Type Extension in Ada 9X," *Ada Letters*, March/April 1991

[10]Seidewitz, E., "Object-Oriented Programming With Mixins in Ada," *Ada Letters*, March/April 1992

[12]Seidewitz, E., "Genericity versus Inheritance Reconsidered: Self-Reference Using Generics," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1994

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[9]Seidewitz, E., and M. Stark, "An Object-Oriented Approach to Parameterized Software in Ada," *Proceedings of the Eighth Washington Ada Symposium*, June 1991

[8]Stark, M., "On Designing Parametrized Systems Using Ada," *Proceedings of the Seventh Washington Ada Symposium*, June 1990

[11]Stark, M., "Impacts of Object-Oriented Technologies: Seven Years of SEL Studies," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, September 1993*

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," *Proceedings of TRI-Ada 1989*, October 1989

[5]Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," *Proceedings of the Joint Ada Conference*, March 1987

[10]Straub, P. A., and M. V. Zelkowitz, "On the Nature of Bias and Defects in the Software Specification Process," *Proceedings of the Sixteenth International Computer Software and Applications Conference (COMPSAC 92)*, September 1992

[8]Straub, P. A., and M. V. Zelkowitz, "PUC: A Functional Specification Language for Ada," *Proceedings of the Tenth International Conference of the Chilean Computer Science Society*, July 1990

[7]Sunazuka, T., and V. R. Basili, *Integrating Automated Support for a Software Management Cycle Into the TAME System*, University of Maryland, Technical Report TR-2289, July 1989

[10]Tian, J., A. Porter, and M. V. Zelkowitz, "An Improved Classification Tree Analysis of High Cost Modules Based Upon an Axiomatic Definition of Complexity," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

Turner, C., and G. Caron, *A Comparison of RADC and NASA/SEL Software Development Data*, Data and Analysis Center for Software, Special Publication, May 1981

[10]Valett, J. D., "Automated Support for Experience-Based Software Management," *Proceedings of the Second Irvine Software Symposium (ISS _92)*, March 1992

[5]Valett, J. D., and F. E. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," *IEEE Transactions on Software Engineering*, February 1985

[5]Wu, L., V. R. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," *Proceedings of the Joint Ada Conference*, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium-Scale Software Projects," *Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science*. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," *Empirical Foundations for Computer and Information Science* (Proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," *Proceedings of the 26th Annual Technical Symposium of the Washington, D.C., Chapter of the ACM*, June 1987

[6]Zelkowitz, M. V., "Resource Utilization During Software Development," *Journal of Systems and Software*, 1988

[8]Zelkowitz, M. V., "Evolution Towards Specifications Environment: Experiences With Syntax Editors," *Information and Software Technology*, April 1990

NOTES:

[1]This article also appears in SEL-82-004, *Collected Software Engineering Papers: Volume I*, July 1982.

[2]This article also appears in SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983.

[3]This article also appears in SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985.

[4]This article also appears in SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986.

[5]This article also appears in SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987.

[6]This article also appears in SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988.

[7]This article also appears in SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989.

[8]This article also appears in SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990.

[9]This article also appears in SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991.

[10]This article also appears in SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992.

[11]This article also appears in SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993.

[12]This article also appears in SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
December 1994

3. REPORT TYPE AND DATES COVERED
Contractor Report

4. TITLE AND SUBTITLE
Proceedings of the Nineteenth Annual Software Engineering Workshop

5. FUNDING NUMBERS
552

6. AUTHOR(S)
Software Engineering Laboratory

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland

8. PERFORMING ORGANIZATION REPORT NUMBER
SEL-94-006

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
NASA Aeronautics and Space Administration
Washington, D.C. 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
CR-189411

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category: 61
Report is available from the NASA Center for AeroSpace Information,
800 Elkridge Landing Road, Linthicum Heights, MD 21090; (301) 621-0390.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Software Engineering Laboratory (SEL) is an organization sponsored by NASA/GSFC and created to investigatethe effectiveness of software engineering technologies when applied to the development of applications software. 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.

14. SUBJECT TERMS
Software Engineering, Software Measurement, Data Collection

15. NUMBER OF PAGES
357

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited