NASA-TM-105,506

N92-19420#
thru
N92-19435#

SEL-90-006

# PROCEEDINGS OF THE FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

November 28–29, 1990

GODDARD SPACE FLIGHT CENTER

Greenbelt, Maryland

SOFTWARE ENGINEERING LABORATORY SERIES SEL-90-006

IN-61

P- 618

# PROCEEDINGS OF THE FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

NOVEMBER 1990

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771 USA

# 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 for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC, Systems Development Branch

The University of Maryland, Computer Science Department

Computer Sciences Corporation, Systems Development Operation

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect 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.

Single copies of this document can be obtained by writing to

Systems Development Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

# AGENDA

## FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP
## NASA/GODDARD SPACE FLIGHT CENTER
## BUILDING 8 AUDITORIUM
## NOVEMBER 28–29, 1990

**Summary of Presentations**

D. K. Cover and E. J. Smith (Computer Sciences Corporation)

**Session 1**

**Topic: The SEL at Age 15**

*Toward a Mature Measurement Environment*
V. R. Basili (University of Maryland)

*Impacts of a Process Improvement Program in a Production Environment*
G. T. Page (Computer Sciences Corporation)

*Results of 15 Years of Measurement in the SEL*
F. E. McGarry (NASA/Goddard Space Flight Center)

**Session 2**

**Topic: Process Improvement**

*A Framework for Assessing the Adequacy and Effectiveness of Software Development Methodologies*
J. D. Arthur and R. E. Nance (Virginia Polytechnic Institute)

*A Method for Tailoring the Information Content of a Software Process Model*
M. B. Arends (McDonnell Douglas)
S. Perkins (University of Houston)

*Software Technology Insertion: A Study of Success Factors*
R. Lydon (Raytheon)

**Session 3**

**Topic: Measurement**

*Pragmatic Quality Metrics for Evolutionary Software Development Models*
W. Royce (TRW)

*RWP Project Ada Development Metrics and Observations*
R. E. Loesh (NASA/Jet Propulsion Laboratory)

PRECEDING PAGE BLANK NOT FILMED

*Early Experiences Building a Software Quality Prediction Model*
W. W. Agresti, W. M. Evanco, and M. C. Smith (The MITRE Corporation)

**Session 4**

**Topic: Reuse**

*Bias and Design in Software Specifications*
P. A. Straub and M. V. Zelkowitz (University of Maryland)

*SEL Ada Reuse Analysis and Representation*
R. Kester (Computer Sciences Corporation)

*Reuse Metrics and Measurements: A Framework*
D. Reifer (Reiter Consultants, Incorporated)

**Session 5**

**Topic: Process Assessment**

*Cost and Quality Planning for Large NASA Programs*
K. Y. Rone (IBM)

*Effect of Formal Specifications on Program Complexity and Reliability: An Experimental Study*
A. L. Goel and S. N. Sahoo (Syracuse University)

*An Analysis of Defect Densities Found During Software Inspections*
J. C. Kelly and J. Hops (NASA/Jet Propulsion Laboratory)
J. S. Sherif (California State University)

**Panel 1**

**Topic: Experiences in Implementing an Effective Measurement Program**

Michael Daskalantonakis (Motorola)
Bob Grady (Hewlett-Packard)
Ray Wolverton (Hughes Aircraft Company)
Mitsuru Ohba (IBM/Japan)

**Panel 2**

**Topic: Software Engineering in the 1980s: Most Significant Achievements/Greatest Disappointments**

Barry Boehm (Defense Advanced Research Projects Agency, Information Sciences Technology Office)
Larry Druffel (Software Engineering Institute)
Manny Lehman (Imperial College)
Harlan Mills (Software Engineering Technology, Inc)
Vic Basili (University of Maryland)

**Appendix A—Attendees**

**Appendix B—Standard Bibiliography of SEL Literature**

# SUMMARY OF PRESENTATIONS AND PANELS

Donna Cover
Elizabeth Smith

COMPUTER SCIENCES CORPORATION

# SUMMARY OF THE FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

On November 28 and 29, 1990, approximately 500 attendees gathered in Building 8 at the National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center (GSFC) for the Fifteenth Annual Software Engineering Workshop. The meeting is held each year as a forum for information exchange in the measurement, utilization, and evaluation of software methods, models, and tools. It is sponsored by the Software Engineering Laboratory (SEL), a cooperative effort of NASA/GSFC, Computer Sciences Corporation, and the University of Maryland. Among the audience were representatives from approximately 10 universities, 30 government agencies, 9 NASA centers, and 100 private corporations and institutions. Fifteen papers were presented in five sessions:

- The SEL at Age 15
- Process Improvement
- Measurement
- Reuse
- Process Assessment

The sessions were followed by two panel discussions:

- Experiences in Implementing an Effective Measurement Program
- Software Engineering in the 1980s: Most Significant Achievements/Greatest Disappointments

A summary of the presentations and panel discussions is given on the following pages.

## SESSION 1–THE SEL AT AGE 15

Frank McGarry of GSFC introduced the workshop, welcomed attendees, and gave a brief overview of the SEL. He noted that on this 15th anniversary of the SEL, it is appropriate to look back on what has been learned: How has the model of software process improvement evolved? What impact have the SEL experiments had on the production environment? What has the government learned about software from these experiences over 15 years?

The three speakers for this session were Victor Basili of the University of Maryland, Gerald Page of Computer Sciences Corporation, and Frank McGarry of the Goddard Space Flight Center.

### Toward a Mature Measurement Environment
### Victor Basili

Basili discussed the SEL from a research perspective, focusing on software measurement maturity. He indicated that a problem in software engineering is that most of the research being done is "bottom-up," that is, researchers are packaging pieces of technology that can't be put together very easily. The software engineering community needs to create a "top-down," experimental, evolutionary framework that can be focused, logically and physically integrated to produce quality software productivity, and evaluated and tailored for the particular application environment. In short, what is needed, said Basili, are more experimental laboratories conducting SEL-type activities.

From a researcher's point of view, the SEL is a laboratory that helps the researcher to understand the various software processes and products. With this understanding, the researcher builds descriptive models of the processes and products that promote even greater understanding and deeper analysis. Basili outlined three phases in the evolution of the SEL. During Phase 1, the emphasis was on **understanding the environment and measurement**. Phase 2 focused on **improving the process and the product**. Phase 3 involved **packaging the SEL experiences for reuse**, recognizing what was appropriate for the SEL, and implementing improvements to the SEL environment. He stressed that the SEL process is a **hierarchy**: improvement depends on understanding, and assessment and improvement always precede packaging.

Basili explained that the SEL **framework** is based on three evolving concepts: the Quality Improvement Paradigm, the Goal-Question-Metric (GQM) paradigm, and the experience factory. These concepts support the basic SEL belief that experimentation is necessary: no piece of technology, method, tool, or process model works under all circumstances.

One example of SEL experimentation is the current evaluation of the Cleanroom process to determine its applicability for building flight dynamics software. A small, controlled Cleanroom experiment was successfully run at the University of Maryland. From this experiment, it was determined that the concept of "no programmer testing" enforces better code reading, that the Cleanroom process is quite effective for small projects, and that formal methods were hard to apply and required a fair amount of skill. Building on this initial work, the Cleanroom experiment was then extended to a larger case study at NASA. Key lessons learned from this study were that it is possible to scale up from a smaller project to a larger one and that the use of Cleanroom techniques survived very well in an environment with changing requirements. Overall, there were increased productivity and a lower error rate. A negative lesson learned was that better training was needed. An even larger lesson was that successful softwar engineering requires allowing for change.

Basili stated that packaging technology requires two things: (1) the continual accumulation of evaluated experiences in a form that can be effectively understood and modified and (2) their collection into a repository of integrated experience models. The experience factory concept supports technology packaging by analyzing and synthesizing all kinds of experience models, acting as a repository for such experience, and supplying information on that experience to various projects upon demand.

What is needed now, according to Basili, is a set of experience factories in a variety of domains, each focused on packaging local experiences, building and tailoring local models, integrating technologies, studying the issue of scale up, and developing automated aids. Basili encouraged researchers to take advantage of the experimental nature of software engineering to provide gains both to industry and to the research domain in a symbiotic relationship; he further encouraged them to learn from and build on the experiences of the SEL.

## Impacts of a Process Improvement Program in a Production Environment
## Gerald Page

In assessing the impact of a process improvement program in a production environment, Page's presentation addressed the question "Are we [CSC as part of the SEL] any better for trying to create an optimizing process?" Page explained that the SEL process improvement environment has been characterized by a conscious, continuous effort to build higher quality systems by (1) understanding the environment, (2) proposing changes to that environment and then measuring and evaluating those changes against baselines, and (3) capturing and packaging that experience to optimize the process and anticipate uncontrollables. These continuous efforts to improve have led to numerous changes over the last 15 years.

Following are three views of the types of changes made in the SEL.

- **Life-Cycle Process Changes:** Testing activity was identified as a weak area for the SEL. Investigations of various testing techniques, including the independent verification and validation methodology (IV&V), were conducted. IV&V was judged to be inappropriate for the SEL, but code reading, found to be by far the most cost-effective testing technique, was added to the SEL process.
- **Technology/Methodology Changes:** Small improvements were experienced from simply introducing a disciplined methodology into the environment, but it was concluded that gaining *significant* improvement would require dramatic changes in the methodology, technology, or organizational structure of the SEL. The subsequent introduction of Ada technology in the SEL produced very promising results in software reuse, leading to further attempts to optimize the process. Cleanroom techniques have also recently been successfully used.
- **Organizational Changes:** Three organizational changes were involved: staff turnover or growth, staff background, and domain growth.
  - The problems presented by staff turnover/growth were addressed by creating new standards and guidelines and augmenting existing ones. These standards have created a more homogeneous environment, in which the use of consistent terminology and procedures minimizes disruptions due to staff turnover. A Software Management Environment (SME) tool has also proven useful in allowing a manager to compare a current project with past projects to determine the outcome for any new project thought to be deviating from the process models.
  - Staff background has changed over the 15 years from primarily mathematics and physics to primarily computer science. The necessary flight dynamics training has been provided through a required training program designed to help software developers understand the engineering applications in the SEL environment.
  - Domain growth (requiring a staffing increase from 35 to approximately 250 developers over 15 years) has been addressed by augmenting the SEL methodology, making it more flexible and able to handle new applications.

Page identified several ways in which CSC has capitalized on the learning from these experiences and changes:

- A **system development methodology** has been devised based on the SEL experience, with standards and procedures developed to make that methodology more effective.

- The SEL experience with quantitative management has been **packaged** in a manager's handbook and in the "Data Collection, Analysis, and Reporting Handbook." This gives managers guidance on how to monitor a project and how to predict what's going on.

- Required **training programs** have been developed for the different roles in system development: analysis engineers, developers, testers, integrators, and managers.

- Measurement-based **engineering process groups** (researchers, experience factory people, experimenter teams) have been established. These groups recommend changes, develop measures to evaluate the changes, and then package the information to institutionalize the improvements found.

Citing statistics in several categories, Page related that although system complexity, general requirements, and system size have doubled, error rates have gone down, the cost of code has remained relatively constant, and effort and schedule estimating are improving. Therefore, the answer to the question "Are we any better off?" is a resounding "YES."

## Results of 15 Years of Measurement in the SEL
## Frank McGarry

McGarry's presentation focused on what has happened in the SEL over the past 15 years and discussed areas in which SEL measurement efforts have made substantial contributions to NASA/ Goddard's understanding of software engineering.

During this period of time, over 75 to 85 projects supporting NASA missions in a production environment have been studied in SEL experiments. From these experiments, a large amount of software metrics data has been extracted to generate the SEL data base. In addition, 150 to 200 reports have been produced describing experiences and results of the experiments. McGarry stated that the SEL has gained seven major insights from these project experiments and reports, with each of the points substantially confirmed by multiple experiments and documented in the SEL literature.

**Measurement is an essential element of software process improvement.** McGarry stressed that measurement is critical in developing a baseline, in understanding changes to an organization's process, and in assessing impacts of these process changes on its products. Because of measurement in the SEL, impacts of process changes have been observed and determinations made if the changes have in fact lead to improvements. Measurement has also proven valuable as a management aid; for example, measuring error rates has provided an early indication of software quality.

**Many potential diversions exist that can sidetrack a measurement-based process improvement program.** The SEL has experienced three such areas of diversion: excessive planning and replanning, over dependence on statistical analysis, and spending too much time looking at methodologies and technologies that in the end don't improve the process. McGarry cautioned that these and other diversions should be watched for and avoided.

***People are the most important* resource/technology.** There is a tremendous difference in people's potential. However, prudently applied methodologies/technologies can optimize the potential of people, and this is where the focus should be.

**Environmental characteristics should dictate the selected software engineering techniques.** Specific measures/techniques may not apply to all domains. Methodologies and technologies must be tailored to the environment, with associated standards and policies tailored to optimize the particular processes found to be effective in that environment. The SEL *definition of effective* standards requires that they be *written, understood, legacy-based, enforced,* and *measurable.*

**Automation is an instrument of process improvement, not a replacement for process understanding.** Only those processes that are very clearly understood and that can be done manually can be automated. Tools can provide significant benefit to a well-defined experience base, and effective tools must address defined process needs. Immature processes are not automatable.

**The heritage of an environment will strongly influence the process followed.** It was found that making significant changes to the process nonetheless induces very *slow* changes in the environment and the products generated, because corporate memory remains the overriding principle of the way software is developed in that environment. Significant process change also requires a significant expenditure of effort and time.

**Software can be measurably improved through the appropriate use of available technologies.** A combination of the appropriate software methodology and technology can produce a significant, favorable impact on the way software is developed. Examples of technologies that have worked successfully in the SEL are code reading, design criteria, use of Ada, objected-oriented techniques, Cleanroom, and management through measurement. Code reading has repeatedly been shown to improve software reliability at essentially no additional cost. Design criteria standards have been demonstrated to produce more error-free software. Use of Ada and object-oriented design have yielded substantial cost benefits through *reuse*. The Cleanroom technology has shown improvement in reliability and productivity and has helped reduce computer resources consumption. Finally, major changes in planning, adjusting, and control techniques have improved cost and schedule estimation.

McGarry stated that NASA's investment in the SEL has been substantial, but comparing various aspects of software development in the 1976–1980 timeframe with those of the 1986–1990 timeframe demonstrates that there have also been many benefits. McGarry specifically cited such benefits as very controlled, predictable cost per line of code; increased reliability; increased reuse; and decreased rework. Other major improvements have been increased manageability, with less dependence on the capabilities of the personnel; the production of more predictable and consistent software; and the development of a rationale for the methods used.

# SESSION 2 – PROCESS IMPROVEMENT

Marvin Zelkowitz of the University of Maryland chaired this session. In his opening remarks, he stated that the papers in this session addressed the following problem: If most organizations use the typical "waterfall" software life cycle, what variations in the process will improve the quality of the software developed?

The three speakers in this session were Richard Nance of the Virginia Polytechnic Institute (VPI), Mark Arend of McDonnell Douglas, and Tom Lydon of Raytheon.

## A Framework for Assessing the Adequacy and Effectiveness of Software Development Methodologies
## Richard Nance

Nance reported that this work arose when his group was asked to review two software development methodologies (A and B); to compare and evaluate them; and to assess the costs and benefits of continuing with both, using only one, or merging the two in some fashion. The major steps in the study were determining an evaluation approach, developing an evaluation procedure, applying the evaluation procedure, and summarizing the results.

Nance explained that since no comparative development procedures were found documented in the literature, a relevant evaluation procedure was first developed and then applied. The rationale for VPI's evaluation procedure was the following: A project-level software development methodology should have a set of clear **objectives**; a process that clearly defines the **principles** needed to reach those objectives; and, finally, adherence to a process utilizing these principles that produces a product with certain identifiable **attributes**. This philosophy must be tempered by practical concerns, such as keeping the framework open, adjusting to differing priorities, recognizing attribute sampling, and being flexible in evaluation procedure application.

The study team determined a statement of objectives, principles, and attributes for methodologies A and B; identified objectives at the project level that led to principles at the process level; and lastly defined attributes at the product level. The objectives and principles were then linked, showing a high degree of interplay among principles. **Properties**, the measurable things that reflect the presence or absence of the desirable or beneficial attributes in the product, were also defined.

Linkages between objectives and principles and between principles and attributes were defined and substantiated. The evaluation procedure was then applied in a top-down approach to determine the **adequacy** of a methodology. Assessing the **effectiveness** of methodologies A and B was achieved through a bottom-up evaluation process.

Finally, Nance presented the results from comparing methodologies A and B using this procedure. He stated that one of the evaluation procedure's greatest strengths was being able to describe things in terms that management could readily understand. Future efforts will include extending this evaluation approach to the issue of software quality assessment.

## A Method for Tailoring the information Content of a Software Process Model
## Mark Arend

Arend's presentation discussed a procedure for tailoring the documentation products and portions of a methodology dictated by a given software process model. **Tailoring** is the act of taking a fully defined software process model or methodology and selecting those items that are necessary based on the nature of the specific product to be developed. Arend stressed that **software quality**, the degree to which software matches the customer/user needs, is an important consideration that must be ensured whenever tailoring is employed.

Arend explained that tailoring is usually guided by personnel experience, ability, and tradition, and that the McDonnell Douglas team found no formal guidelines for tailoring methodologies. The procedure they subsequently developed and followed in their study was a step-by-step, cohesive approach to tailoring, one that had customer needs and product quality requirements as the driving factors. Customer/user needs were identified and an approach used to reflect these needs in a subset of information products extracted from all possible information products identified in their process model.

To characterize customer/user needs, software quality assurance (SQA) concepts were applied. SQA involves defining **quality factors** and **quality criteria**. User-oriented quality factors were captured through the use of questionnaires and customer/user interviews. Quality criteria, generally more software oriented and more closely related to software testability, were directly derived from the quality factors. Once a good set of factors that the user wanted in the software were identified, a larger set of criteria supporting the existence of those factors was identified. Development and management techniques were then selected that would ensure the presence of these quality criteria.

In the key step of the tailoring procedure, information products that matched or supported the chosen development and management techniques were selected and tailored. Arend explained that information products act as specific vehicles that force us to recognize, formalize, and adhere to techniques to specify, design, and implement software of appropriately selected quality. Therefore, an appropriate subset of all possible information products becomes a significant aid in reaching the goals of the given software project and especially in satisfying customer needs. Arend also stated that if a design methodology is not already imposed, one can be selected based on matching the information products chosen from a methodology with those recommended to achieve the product's quality profile.

Steps remaining to be applied include refining the quality requirements questionnaire, devising a way to weight questionnaire responses to quantify products' quality profiles, and developing a list of information products sorted by quality criteria.

## Software Technology Insertion: A Study of Success Factors
## Tom Lydon

Software technology insertion (STI) in this study consisted of two parts: (1) selecting a new technology, typically a method or tool, and (2) creating an opportunity to insert that new technology in a new or ongoing software project. Lydon clarified that successful STI can be a **perceived** success (the user's sense of labor, computer cost, and time savings) or a **real, measured** success. This study concentrated on **perceived** success by the actual project users.

For the purposes of this study, an STI case applied a single technology to a single project, usually within a single development phase. The study involved 13 different projects with a variety of project characteristics, 21 new software technologies, and numerous study factors, among them technology type, maturity, insertion method, and project size.

Two key people (the lead engineer and the department manager) for each project were surveyed for their perceptions of success or failure. Based on these surveys and on responses to six study questions, each STI test case was ranked and evaluated. A close examination of the top 11 rated STI cases indicated that the main reasons for success were (1) synergy within a project (a good relationship between the using and supporting organizations and a positive attitude by the affected manager), (2) critical need for the capability, (3) synergy between two technologies, and (4) use of a mature and powerful tool. It was significant that three of these factors were organizational in nature, with only one factor related to what was inherent in the technology. In these 11 cases, saving computer costs may or may not have occurred, and meeting **expectations** was not so important as the **perception of time or labor savings or quality improvement.**

In the bottom seven rated STI cases, the main reasons for "failure" were (1) the technology was immature, (2) interface problems arose, (3) the technology was judged to be "not needed" by the lead engineer, and (4) the wrong technical solution was used. For these least successful cases, the technologies being tried may or may not have improved quality but *were* judged to have failed in saving time, saving labor, and in meeting expectations.

Lydon reiterated that this study had focused on **success factors** and on **perceived** STI success. He summarized the results as follows:

- *Saving* ***schedule time*** *and* ***labor costs*** *was the driving force behind the successful STI cases.*
- **Improving quality** seemed to be a necessary but not sufficient condition for successful STI.
- Exceeding users' expectations was not necessary for successful STI, but **not meeting expectations** was sufficient for failure. The lesson is that a support group or organization must control people's expectations.
- There was much greater success for **competence-enhancing** (incremental) improvements than for competence-destroying technologies.
- There was greater success with **mature** versus young or old technologies.
- There was somewhat greater success for **in-house** versus outside supported technologies.

Lydon related that the next step is to link perceived success with real success via software metrics collection. Raytheon is implementing corporate-wide, automatic software metrics collection as a by-product of development.

## SESSION 3 – SOFTWARE MEASUREMENT

The Session 3 chairman was John Valett of the Goddard Space Flight Center. The three papers in this session addressed different aspects of software measurement: Walker Royce of TRW discussed quality metrics and how they might be utilized, Bob Loesh of NASA/JPL presented results from a specific measurement effort, and Bill Agresti of the Mitre Corporation discussed using design measures to predict system quality. These measurement activities share the ultimate goal of showing how measurement can be used to better understand and improve a software product.

### Pragmatic Quality Metrics for Evolutionary Software Development Models
### Walker Royce

Royce's presentation discussed experiences on a 1-million-line Ada project, entitled the Command Center Processing and Display System Replacement (CCPDS-R), undertaken for the United States' Ballistic Missile Early Warning Center. TRW recognized that its transition to developing this and other such large Ada systems would require significant internal research to identify changes necessary for their existing software development and metrics collection/analysis approaches. Several such changes included adopting an evolutionary versus canonical waterfall development approach, using Ada as a compilable design language as well as implementation language, and adjusting cost and schedule estimating techniques accordingly. A 350,000-line subsystem was used as the pilot for selecting meaningful metrics, collecting data, and analyzing results.

The main objective of adopting the evolutionary development approach was to minimize rework. This required fixing things early and **designing for change** to accommodate requirements volatility. The focus of TRW's software quality metrics (SQM) was maintainability—how easy would it be to change the software throughout the lifecycle. Interpersonal communications were also minimized using a small, expert design team; a layered architecture; and Ada as a self-documenting, life-cycle language.

Quality was defined as **the degree of compliance with customer expectations of function, performance, cost, and schedule.** Quality metrics were derived from measuring the amount of rework and plotting these measures as they evolved over time. The evolutionary development approach supported this quality assessment by permitting tangible insight into the end product beginning with the very earliest stages of the program. For example, because a significant amount of coding, testing, and product demonstration had already been conducted by critical design review (CDR), the CCPDS-R approach allowed prediction in some objective terms of the future maintainability and reliability of the software.

Development progress over the life cycle of the program included approximately six builds, reaching 100-percent development around month 35. Royce related that fewer than 0.5 problems per 1000 lines of code occurred during the development and test phases. Addressing difficult design issues early to avoid major "breakage" later, when there would be a larger configuration to maintain, was a major project goal in containing rework. Tracking total rework versus closed rework provided useful data as a progress indicator. Insight into the activities of the test and maintenance organizations and reacting to problems before they escalated were also important to the success of the program.

Royce summarized some of the metrics from the effort. **Rework Proportions:** 6.7 percent of the total manpower devoted to software was spent doing rework on configuration baselines. Approximately 13.5 percent of the total product had to be reworked prior to being delivered. **Modularity:**

The average breakage per change was approximately 53 SLOC per system change order (SCO). **Changeability**: The average SCO took approximately 2 man-days to resolve and fix. Change effort became quite predictable and stabilized over time, demonstrating the success of the approach and the usefulness of the metrics. **Maintainability**: Maintainability was defined as a normalized rework productivity. The software in this effort was determined to be approximately one-half as complex to change as it was to develop from scratch.

Royce concluded that selecting and implementing meaningful software quality metrics require consistency of application, the use of automated tools, and management *and* practitioner acceptance. One advantage of the TRW approach was that it produced quantitative data for decision making and for determining requirements compliance in areas such as maintainability, modularity, and adaptability. It also provided historical data for better future planning. The bottom line is that such quality metrics can be and are being used effectively on large projects.

## RWP Project Ada Development Metrics and Observations
## Robert Loesh

Loesh's presentation discussed metrics experiences on the Real-time Weather Processor (RWP) System being developed by JPL for the Federal Aviation Administration (FAA). The software-intensive system, comprised of approximately 97,000 Ada statements and 280,000 lines of commercial, off-the-shelf software (COTS) implemented in C, has been installed on commercially available hardware. Over the project's 3-year development period, JPL intensively tracked Software Problem Failure Report (SPFR) activity and performed extensive analysis of error counts, time-to-fix, requirements documents changes, and system specification changes.

According to Loesh, approximately 40 percent of the 222 issues addressed by the design team dealt with interface concerns, a strength predicted by Ada advocates and clearly exploited by RWP developers. Taking a slightly different perspective, nearly 66 percent of the total issues were identified by people preparing test descriptions and procedures. This surprising statistic highlighted a major lesson learned on the project: **write the initial version of system test procedures as early as possible.** The scope of the rework associated with such test-procedures-related problem reports is narrower because test procedures are written relatively early in the development cycle, when the immature system is still of manageable size.

Loesh focused on data collected in the areas of system growth as a function of the number of changes applied to the system and of testing and test strategies. Of the approximately 2100 SPFRs generated to date, he stated that the majority have been submitted in the system integration testing phase (40 percent) and in the CSCI integration phase (18 percent). When JPL observed through its metrics program that only 9 percent of the errors were uncovered during individual CSCI testing, this type of activity was shortened because of the low return for the effort invested. This allowed JPL to concentrate its effort on the more revealing CSCI integration and system integration testing phases.

During system integration testing for the Ada code, about 5 errors per 1000 Ada statements were recorded, and there were approximately 2.3 errors per 1000 carriage returns. These numbers are nearly one-half the rate typically found in JPL's FORTRAN projects of similar size. Loesh stressed that this result demonstrated the benefit of Ada: **product quality improvement is notable**, even if transitioning organizations are still struggling with extended schedules and higher costs.

Loesh concluded with a discussion of error correction counts and the associated effort required, observations relating to portability issues, tools used to support JPL's portability analysis, and levels of risk involved to convert portions of the RWP code to increase its portability. At this point, Loesh stated that the RWP team, incorporating X Windows and placing **emphasis on designing for portability**, has achieved approximately 75-percent to 90-percent portability for the future.

Finally, he stressed that the project's metrics analysis is considered preliminary, with adjustments expected shortly when the entire project is completed.

## Early Experiences Building a Software Quality Prediction Model
## William Agresti

Agresti discussed an ongoing research project whose primary objective is to test the hypothesis that Ada software quality factors can be predicted during the design phase. Project goals include developing a set of diagnostic capabilities and determining ways to quantitatively assess the designs of large systems. The technical approach taken was to build multivariate models to estimate reliability and maintainability and to examine the characteristics of the software design itself as captured in the Ada design language. Agresti emphasized that since this project is still in the early stages, the results given in this presentation were preliminary.

The study postulated that by analyzing a software system's static design structure, examining such attributes as design complexity, coupling, and cohesion, researchers could predict the quality (number of errors) expected for a finished system. To test this notion, the experiment looked at the basic architectural decision choices possible in Ada designs.

Two estimation models were developed, one for reliability and one for maintainability. To allow for several definitions of reliability and maintainability and to accommodate a variety of measures, each of the quality factors was modeled as a function of several parameters, including design characteristics, environmental factors, model parameters, and an error term. **Design characteristics** at the architectural level are the features extracted from the design artifact, such as context coupling and visibility, while **environmental factors** include items extraneous to the artifact such as the volatility of changes to the software and the reuse level. The **error term** accounts for any unexplained variation.

A simple notion of a static Ada architecture was used, in which Ada structures were composed of design units from a "parts" bin and design relations from a "connections" bin. This required careful consideration of the many different types of "parts" and the various kinds of "connections" possible.

"Legal" Ada compilation units were composed from these parts and connections and were used as a framework to evaluate project data on 21 Ada subsystems from the SEL data base. In particular, the study looked at the software's reliability (error counts) and maintainability (time-to-isolate-and-fix) data tracked by the SEL. In general, there was good variability in reliability and maintainability, the dependent variables. In some of the subsystems, compilation units were partitioned into library unit aggregations, and the **pattern of information access** throughout these units was also studied.

As an important part of the study, the team examined the number and scope of declarations made visible to the units of a library aggregation (via a "with" clause), looking for the effect that the number of imported and exported declarations may have on a system's future reliability and maintainability. The basic idea was to identify from where and at what levels external resources were accessed for that unit aggregation. As a result of such analysis, simple **static measures** including the number of imports, the number of exports, and the number of cascaded imports were compiled and studied.

Results from the early modeling efforts are still preliminary due to a limited amount of data. However, Agresti provided initial findings for the reliability model case, in which the dependent variable is errors per 1000 lines of code; the relevant variables are context coupling, visibility, and volatility. A good fraction of the variation in error rate is being explained [$R^2 = 0.72$ (adjusted)]; context coupling and change rate are significant in explaining the variation.

Since system and acceptance testing errors might better reflect the architectural issues of interest (i.e., the interunit relations captured in the design), a preliminary reliability model was determined for errors recorded in these phases (unit testing error counts were excluded here). The error rate variation again seemed well explained [$R^2 = 0.78$ (adjusted)]. The context coupling and visibility design characteristics contributed, but the **major environmental factor** was reuse; custom code was a strong indicator in explaining the variations in the data.

Agresti related that the team's early results in formulating estimating models for reliability and maintainability have been encouraging, and that they look forward to exploring additional hypotheses and to developing more robust models that can be subjected to validation.

# SESSION 4 – REUSE

Session 4 addressed software reuse. Sharon Waligora of Computer Sciences Corporation chaired the session and gave some opening remarks on the reuse issue. One of the biggest challenges facing software engineers and managers today is the ever-increasing demand to build larger, more complex systems with more limited resources. If the current trend ccntinues, it is expected that both adequate funding and qualified people will become more scarce in the future. Industry leaders are looking to software reuse to help meet this challenge. Some believe that code reuse is the answer, while others believe that reuse can be maximized only through changes to the early phases of the life cycle. The three speakers in this session addressed three aspects of reuse. Pablo Straub of the University of Maryland discussed the need to improve specifications so that designers are free to create reusable components. The second speaker, Rush Kester of Computer Sciences Corporation, described an effort to characterize successful Ada code reuse in the SEL environment as a basis for developing guidelines for creating reusable components in the future. The final speaker in the session was Don Reifer of Reifer Consultants, Incorporated; his talk addressed reuse metrics anu the resulting lessons learned.

## Bias and Design in Software Specifications
## Pablo Straub

Specification reuse has been recognized as a key to achieving significant increases in software reuse. However, specification reuse can be difficult because specifications are often subliminally tied to particular implementations. Straub addressed this problem by studying implementation bias, or overspecification, which may be translated as the tendency for a specification to implicitly direct the details of an implementation. His study produced a classification of requirements whose goal is to define a framework to explain the nature of implementation bias. The theory yields a precise definition of bias and demonstrates that despite efforts to the contrary, bias is inherent in specifications.

Unlike the canonical software life cycle in which "specification" appears as only the first phase, Straub chose to regard the product of each phase as the specification for the next phase. Straub's successive refinement of the original specification (or "staged specification" approach), and the inherent potential for introducing new or expanding existing errors at each subsequent step, highlighted the need for **high quality**, **abstract** (general), and **complete** specifications. Other desirable qualities are that specifications be **consistent**, **correct**, **reusable**, and **traceable**.

Straub explained that typically the rule to avoid overspecification or bias has been, "Specify **what** the system should do, not **how** to do it." But he also stated that keeping these terms clear in everyone's mind, especially through successive phases, isn't always easy and that the result of such confusion is often the *inverse* of implementation bias, i.e., underspecification. Underspecification in turn leads to assumptions about the final product or may introduce other errors into the software.

To address the over-versus-under specification problem, Straub's study defined a framework for classifying the requirements in a specification. He discussed interrelationships among the many elements of this classification scheme, stressing that among other things, an analyst's aim should be to produce specifications free of **extraneous** attributes (those arising from misconceptions) or **imposed** attributes (those resulting from a restriction in method or language). Otherwise, the consequence of bias is that the solution adopted is not the optimal one. Additionally, he stated, bias cannot be completely eliminated; as long as there are nonexplicit requirements, which is always the case because specifications are rarely complete, there will be a potential for bias.

Other considerations need future effort: the term **requirement** must be formally defined; a **method** to identify bias must be developed; and a **formalism** to write specifications with attributes, such as the origin of the requirement, must be devised.

Another side effect of this research is establishing a **relationship between bias and software defects**. Errors can be related to fictitious requirements; and faults can be related to bias, where bias is a minor fault that doesn't make the system unacceptable but does make it nonoptimal. Within this context, failure can be linked with inefficiency.

Future efforts will focus on testing these ideas by measuring bias in a specific, sizeable project and on exploiting the relationship between errors and bias.

## Ada Reuse Analysis and Representation at the Software Engineering Laboratory (SEL)
## Rush Kester

The focus of this presentation was a description of graphical representations and analysis techniques developed to study the reuse of Ada source code across 1990 Ada components in the SEL. Reuse has been an important part of the culture in Goddard's Flight Dynamics Division (FDD) environment throughout its entire history because of the assumption that there are economic benefits directly related to the amount of code reused without change. With high-level, verbatim reuse, systems can be delivered sooner and at lower cost; can be improved incrementally; and are more reliable. One motivation for this study was to understand the effect of introducing Ada and object-oriented design (OOD) in the SEL, which significantly increased the potential for code reuse. A high degree of reuse was expected due to the nature of Ada and OOD and to the clear focus of the FDD application domain; however, the goal of this study was to *confirm* this hypothesis through objective techniques.

Kester stated that this Ada reuse study was still in the "understanding and characterizing stage" of the SEL's process improvement paradigm. The primary goals of the current phase of the study were to determine the patterns and trends of reuse and to understand the characteristics that distinguish the reused from the nonreused components. Secondary goals were to identify candidate components for a reuse library, identify the applicable domain of a component's reusability within the environment, and address some reuse-related configuration management issues.

Kester related that the FDD environment is mostly FORTRAN-oriented, and that FORTRAN projects *do* emphasis reuse but have not been so successful as the Ada projects. He illustrated this by citing that some of the recent Ada systems are approaching 100 percent total reuse, with 80 to 90 percent reuse without change.

Kester presented a summary of six types of analysis reports, many of them graphical representations, that were used in the study. These reports identified producer and consumer projects; showed the steady increase in reuse from one generation of an application to another; highlighted component lineage, both forward and backward; identified the granularity of reuse for a particular project; depicted the level of functionality that was being reused (i.e., just a single component or an entire branch of the call tree); and finally, reflected coupling between various components through the Ada compilation order.

Some reuse patterns and trends have been observed. Initially, application-independent components, mostly utilities, were reused; now most reused components reflect the flight dynamics domain. Also, Ada "generics" had significant impact on the amount of reuse without changing components, and OOD significantly improved modularity and allowed component reuse from one project to another.

Future efforts will include developing guidance for improving the way software is designed to increase reuse to promote further economic benefit and investigating the characteristics that distinguish reusable components.

## Reuse Metrics and Measurement—A Framework
## Donald Reifer

Reifer reported on the culmination of 3 years' work in reuse metrics conducted by a national team of 40 firms under the auspices of the Joint Integrated Avionics Working Group (JIAWG), chartered by Congress to achieve commonality in avionics systems across all new-generation aircraft. In this effort, a shared technology base is being used to achieve high degrees of reuse on major programs.

Reuse efforts over the past 20 years have been disappointing, in part because many cultural considerations impede government programs. Examples of these considerations include the need for financial incentives for contractors to reuse software, the lack of a functioning reuse identification-insertion-maintenance framework in the software industry, and the lack of a high-level government advocate to champion reuse. Despite these difficult challenges to achieving effective reuse, Reifer stated that the advent of systematic reuse via an objected-oriented paradigm involving representations, languages, and technologies holds promise for the future. The significance of **metrics related to reuse** will likewise grow dramatically. These metrics will be used to govern fees in contracts, to determine the efficiency and effectiveness of libraries, and to judge the quality of reusable software objects against minimum standards of acceptability.

Reifer reiterated that metrics are key to achieving successful reuse, but that for these metrics to be applied effectively, they must be compatible with DOD processes, easy to collect and understand, objective and unbiased in nature, predictive of the future, and incur a minimum cost for measurement. He indicated that the JIAWG is working in several areas to define such comprehensive, quantitative metrics, and he discussed two topics particularly important to the participants: the acquisition ratio and the reuse ratio. The **object acquisition ratio** is a weighted average involving the number of reusable software objects (RSOs) acquired per collection in relation to other attributes of the collection. ["RSOs" were defined as life-cycle products developed to be reused (such as designs, algorithms, code, and test cases) and a "collection" was defined as a homogeneous grouping of clustered objects (such as test cases).] Similarly, the **object reuse ratio** involves the number of reused objects in a collection as a ratio of the total number of objects in that collection, the number of collections, and a weighting factor for each collection.

The study found that the cost of packaging reusable objects varied from 10 percent (for limited reuse packaging) to 36 percent (for extensive reuse packaging), **when a reuse-insertion infrastructure was already operational**. Costs would be higher without such a functioning infrastructure. However, the benefits gained from reuse ranged from 20-percent savings (for planned reuse) to 60-percent savings (for optimized, domain-specific reuse). Ratios were also used to obtain a multiple-instance reuse economic model to look at cost benefits across multiple deliveries to amortize the cost within and across projects. The group continues to refine this model to predict cost savings as a function of the number of reuse instances and to indicate the breakeven point. Quantifying and analyzing software quality factors, such as correctness and testability, and defining similar quality criteria for large reuse libraries are additional areas of importance to the JIAWG.

Because of the wide range of metrics efforts described, Reifer stated that reuse across 143 completed Ada projects has increased from 10-percent nominal in 10 application domains in 1987 to 21-percent nominal this year. To further improve these reuse percentages, the JIAWG is attacking a number of nontechnical barriers that inhibit the effective use of the technology base.

Based on this strong track record, Reifer concluded that the JIAWG is not just studying reuse—it is **doing reuse.**

## SESSION 5 – PROCESS ASSESSMENT

Session 5 was chaired by Rose Pajerski of the Goddard Space Flight Center. The presentations in this session focused on software process assessment and the need to have an understanding of the local environment before an attempt is made to improve any part of the development process. The first presentation, given by Kyle Rone of IBM, covered the entire software life cycle. The other two presentations in the session involved more specific parts of the process. Amrit Goel of Syracuse University discussed process assessment activities in the specifications phase, while John Kelly of NASA/JPL addressed the implementation phase.

### Cost and Quality Planning for Large NASA Programs
### Kyle Rone

Building large, complex programs is very difficult, but earlier and better planning can help minimize or avoid some typical problems. Rone's presentation addressed how to do planning for cost and quality on such large programs as the Space Station and Earth Observing System (EOS), which require very particular and careful planning.

Several essential considerations are key to successful planning: (1) ensuring a compliant product, (2) generating the product within budget and within schedule, and (3) producing a product with the appropriate quality level. These requirements must be integrated and planned concurrently and consistently across releases.

Rone stressed that once models of an organization's process have been defined and tested, managers must be able to calibrate their models to reflect process changes, identify project considerations different from experiences on previous projects that may also induce changes in the models, and periodically revise the set of management and planning techniques to reflect their environment more realistically.

As an illustration of the evolution and refinement of IBM's estimating models, Rone reported that size estimates for Shuttle flight software progressed from + 11 percent using the early model, to -6 percent using the middle model, to only -2 percent on later missions. More astonishingly, ground system estimates came within 1 percent of actuals overall. Discrepancy report data from the Shuttle were also plotted over time and produced a well-defined Rayleigh curve. These curves were used to estimate error estimates across the process.

Given a set of models that truly reflected the processes of the environment, a methodology was followed for doing both software cost engineering and software life-cycle quality management. For cost **engineering**, user requirements were broken int, a set of functions and the system size was estimated. Estimation models were used to predict the labor effort, phase it across time using a Rayleigh curve, and develop the resultant schedules. Other costs, such as subcontractors and overhead, were also added, resulting in a project cost plan. Periodic feedback and replanning allowed requirements changes to be incorporated and operational increments to be factored into the plan. Rone stressed the **critical importance of measurement as the cornerstone of their assessment and replanning cycle**. A similar process was followed for **software life-cycle quality management**.

Rone cited an asymptotic curve relating the product error rate to the percent of the project budget spent on independent testing (IV&V) as a particularly beneficial result of planning based on metrics. It was found that spending much below 10 percent on IV&V yielded an unsatisfactorily high error rate. Above that basic figure, the optimum percent spent depends on the quality requirements (criticality) of the particular project. For less critical projects such as tools and mission control systems, 10 to 20 percent yields good quality (one error per KSLOC) for the least expenditure. For highly critical projects such as a Shuttle model, 80 percent is spent for IV&V to reach an error rate of only 0.1 errors per KSLOC. Spending more than that is counterproductive, since small gains in quality require enormous deltas of expenditure.

The software tool "Squeeze" supported Rone's team in performing software cost and quality engineering for this study and took into account such project characteristics as requirements, complexity, size, and criticality, as well as process and environmental characteristics,.

Rone concluded that managers must plan and continually replan and that well-developed metrics are an essential element of this cycle. The essence of metric management is to **demand that measurement be used** to assess process changes and to provide a more disciplined framework necessary for effectively managing large projects.

## Effect of Formal Specifications on Program Complexity and Reliability: An Experimental Study Amrit Goel

The objectives of Goel's experimental study were (1) to investigate the effect of using formal specifications on project productivity, reliability, and complexity and (2) to compare the results with project versions developed from informal specifications. Formal Z specifications were developed from informal specifications for the NASA Launch Interceptor Program (LIP). These formal specifications were then used to develop three independent versions of LIP in "C". Each study version was tested against a set of 54 test cases from a previous experiment involving LIP and was also executed for 1 million test cases to simulate operational testing.

Goel presented a table comparing program metrics from three different programmers writing "C" code from Z specifications versus three other programmers writing Ada code from informal specifications. Metrics included the number of source lines of code each produced, the number of comment lines, and the system complexity (a combination of internal and external complexity, where the external complexity measured module inter-relationships). The "C" programmers produced source lines of code ranging from 373 to 669, with system complexity ratings ranging from 53 to 81, while the Ada programmers produced source lines of code ranging from 691 to 851, with system complexity ratings ranging from 297 to 334.

Productivity numbers were examined, and consideration was given for the time that some of the study team programmers needed to learn the Z formalisms and to write the Z specifications. Goel's analysis of development effort figures indicated that a considerable amount of design work had actually occurred during the specification phase, somewhat skewing the analysis. The number of errors (not including compilation errors) found by the programmers was also tracked. Programmers A and C, who had written their own Z specifications, found all errors in the development and unit testing phase, while programmer B found most errors in the later, functional testing phase.

Z specifications were judged to be helpful in several areas: certain ambiguities were resolved by looking at the problem more formally; it was possible to express some invariant properties of the system more clearly; and some types of analytical faults were avoided as a direct result of using very formal specifications. The use of Z specifications also exploited the repetitiveness of certain launch conditions; this was helpful in designating functional groupings for design and testing.

Goel drew the following conclusions from this study:

- Use of Z specifications was clearly helpful in **reducing errors**.
- Based on a few metrics, it appears that the **complexity** of code developed from Z specifications was **lower**.
- The total **effort** involved, including learning Z formalisms and developing the formal specifications, was **comparable** to that for developing versions from informal specifications.

However, Goel stressed that this was a very small experiment and did not provide conclusive evidence about the superiority of formal specifications over informal ones. Further work is necessary to explore the feasibility and usefulness of Z for large problems (scalability) and to investigate the reusability of such formal specifications.

## An Analysis of Defect Densities Found During Software Inspections
## John Kelly

Kelly presented results of a 3-year effort at JPL to analyze data assessing the value of software inspections. JPL inspections are detailed technical reviews performed on intermediate engineering products (in-phase reviews); they are highly structured and well defined, carried out by a small group of peers, and controlled and monitored through metrics and checklists. The 203 software inspections involved in the study represented approximately 5500 hours of worktime and included examinations of requirements, architectural design, detailed design, source code, test plans, and test procedures.

Inspection elements tailored to the JPL environment included participants and team composition, training, and support documentation. In addition, a short, optional "third-hour" phase was sometimes used during the inspection meeting to clear up discrepancies and to discuss possible fixes for the defects found. Approximately five team members were used for an inspection, with a total staff time of around 28 hours. Input to the process generally included about 35 pages of documentation or code, and the output from the process was about 35 pages of documentation or code less 4 major defects and about 13 minor defects (where a major defect was defined as one that would cause the system to fail or to miss a requirement; minor defects were all other nontrivial defects).

One major metric used to monitor and control inspections was the number of defects found per page. Defects found per page versus pages per inspection tapered off as more pages were covered in a fixed 2-hour inspection. It was determined that to maintain quality, **limiting an inspection to a maximum of 40 pages optimized the effort.** Another major metric was defect density versus inspection type. Analysis showed that a significantly higher number of defects existed in requirements than in code, and a significantly higher number of defects existed in high-level test plans than in low-level test plans. This indicated that the **major quality problem** at JPL during code development is the **writing of requirements.** Predictive modeling of the defect density versus the inspection type showed an exponentially decreasing density of defects as the code level was approached.

Kelly reported that generally about 1/2 hour was required to fix a defect found early in the life cycle, compared with 5 to 17 hours to fix a defect found during test. Even if the time to find the defect is added to the 1/2 hour time to fix, it is much cheaper to fix the defect early than it is to fix it during testing. One rationale is that defects multiply themselves as the phases move on, creating a "tree" of defects. **Inspection allows the defect to be found at its root node and to be fixed there, which is much less expensive.**

Major conclusions of this study were the following:

- A wider spectrum of errors was found using inspections than was found with previous techniques; the most prevalent types of errors were in the areas of clarity, logic, completeness, consistency, and functionality.
- Increasing the number of pages in a single inspection decreases the number of defects found, with about 40 pages being optimal.
- **The highest defect density was observed during requirements inspections.**
- Larger team sizes (6 to 8 people) seem to be justified for higher level inspections (requirements inspections), because they provide a broader viewpoint and seem to provide an increased defect finding capability.

- Code inspections were superior over a single-person code audit. However, future research may be able to increase the return on investment in code inspections through the use of automated support.

## 1990 SOFTWARE ENGINEERING WORKSHOP – PANELS

Two panels were added this year to the Software Engineering Workshop. The first panel addressed how to establish a successful measurement program. The second panel was designed to stimulate discussion by highlighting the opinions of experts in the field of software engineering.

## PANEL 1 – EXPERIENCES IN IMPLEMENTING AN EFFECTIVE MEASUREMENT PROGRAM

The first panel of the workshop involved discussion by four distinguished panelists who have done extensive work in areas of software engineering such as measurement, experimentation, and data collection: Michael Daskalantonakis of Motorola, Bob Grady of Hewlitt-Packard, Ray Wolverton of Hughes Aircraft Company, and Mitsuru Ohba of IBM/Japan. Frank McGarry of the Goddard Space Flight Center moderated the panel. Each panelist was asked to relate his experiences in defining and working with a successful measurement program and to convey his experiences to other practitioners who may want to implement such a facility in their own organizations. Some of the considerations of interest are what does the measurement program look like, what obstacles had to be overcome, what were the costs and benefits, and what is the long-range outlook.

### Michael Daskalantonakis
### Motorola

### Overview of the Motorola Software Metrics Program

Beginning in 1988, a company-wide program on software metrics was established at Motorola. The primary emphasis of this program has been **establishing an organizational infrastructure** to support the use of software measurement technology across the product groups and to ensure that they use the measurement technology effectively to obtain the maximum benefits. Three important activities have been key in Motorola's successfully launching its metrics program: (1) establishing a Metrics Working Group (MWG) across all Motorola business units to design a standard set of metrics; (2) creating a Metrics Users Group (MUG) to provide a focus for implementing software metrics on the projects, to look into automation (tools), and to serve as a forum for exchanging ideas; and (3) identifying metrics champions within the corporate research and development groups and within business units. Important 2-day training workshops were held to promote the effective use of metrics on the projects, and followup consultation was also provided. A minimum set of software metrics, now **required** by the Motorola Quality Policy for Software Development, was defined and additional metrics were used by projects as necessary.

Daskalantonakis summarized the overall philosophy of the measurement program at Motorola: **The goal is not measurement. The goal is improvement through measurement, analysis, and feedback.**

### Obstacles That Had To Be Overcome

While its program is currently considered a success, Daskalantonakis discussed two obstacles Motorola experienced in establishing its measurement infrastructure.

- A "system" had to be set up to collect the software metrics data and to analyze the data for process improvement. This required obtaining tools to automate metrics collection and analysis activities.

- Software developers and managers had to be convinced and reassured that metrics data would not be inconsistently reponed, misused, or misinterpreted and that the collection effort cost would not unduly burden projects. Training and additional guidelines are being used to address these culturally related issues.

Costs of the Metrics Program

There have been costs to establish the program as well as operational costs. Program costs include man-hours for MWG and MUG meetings and tool development costs. Operational costs for the metrics program from sample Motorola divisions have been 1 percent or less of resources. Post-release metrics costs have been insignificant compared with benefits, but more work needs to be done on automation. In general, Daskalantonakis related, **the overall cost is considered to be acceptable and justified.**

Benefits to Date

The benefits realized by Motorola have clearly justified its investment in a metrics program:

- Software Quality Awareness: People have started thinking about the **software process, the quality of that process, and the quality of the resulting product.** Metrics data have helped the understanding of several problems, have demonstrated the severity of these problems, and have spurred action toward solutions.

- Establishing Baselines and Goals: Metrics have helped establish baselines to identify current progress and to set up aggressive goals, leading to significant quality and productivity improvements.

It is not the metrics themselves that have made the difference, but the actions taken as a result of looking at the data; the benefits stem from analyzing the data and feeding back information to improve the process. In addition, there are many indirect benefits, such as improved acceptance criteria and improved schedule estimation.

Long-Range Benefits Expected

Daskalantonakis concluded with a discussion of several areas expected to yield long-term benefits to Motorola as a consequence of its metrics initiatives:

- Learning From Mistakes: Future problems can be avoided by looking at metrics data from previous projects and learning from their mistakes.

- Improvement in Customer Satisfaction: Improved product quality will promote improved customer satisfaction.

- Cost Reduction: Improved quality and reduced rework cost will lead to significant cost reductions. In addition, resources will be freed up for new software development work.

- Cycle Time Reduction: Productivity improvement is expected to reduce the cycle time, allowing products to reach the market in a timely manner.

Again, Daskalantonakis emphasized that metrics can only highlight the problems and suggest ideas as to what can be done. It is the **action** taken that brings the benefits.

## Bob Grady
## Hewlitt-Packard

### Overview of the Hewlitt-Packard Software Metrics Program

Hewlitt-Packard began its metrics program about 7 years ago as part of an overall productivity and quality improvement initiative. The starting point was a standard, although primitive, set of metrics consisting of code volume, effort, and defect definitions. Over the past 7 years, a supporting infrastructure has been put in place, with activities at the corporate level to support the metrics effort and to provide divisional process assessments. Most of the productivity and quality improvement programs are brought together at the product group level. At the division level, metrics efforts are well supported through the quality managers and productivity managers; the quality managers focus on the quality of the product, whereas the productivity managers focus on the efficiency of the processes. To support all of these measurement efforts, training needs have been addressed through an internal 2-day metrics class that has evolved over time.

At the corporate level, two major software goals, identified as "10X improvements," were established in 1986. The first goal was to **improve by a factor of 10 the postrelease product defect density** over a 5-year period. The second goal was to **improve by a factor of 10 the number of open serious and critical defects**, also over a 5-year period. This second goal complements the first one. Both were directly tied to defects, rather than cost, assuming that there is an indirect relationship to cost. There has been progress toward meeting both goals.

Grady described the Hewlitt-Packard metrics program in terms of a hierarchy. He identified five different stages:

- Acceptance of measurement
- Availability of project trend data
- Use of common terminology necessary for data comparisons
- Experimenting to validate the best practices
- Performing analysis and automated data collection with expert system support

In the Hewlitt-Packard experience, the first three stages were accomplished relatively quickly, within the first 2 years. Now, much time is spent with experiments validating the best practices and in understanding those things that are going right. The fifth stage will take more time to achieve.

### Obstacles That Had To Be Overcome

Grady discussed four obstacles to establishing a metrics program that were more cultural than technical in nature:

- Perceptions of Metrics: The basic perception had to be overcome that metrics were focused only on code analysis. In the industrial area, the feeling is that metrics are driven from business types of practices in terms of project tracking. However, many other items such as effort, quality, and productivity projections versus actuals are also necessary.
- Promotion: Three separate Hewlitt-Packard groups (top management, project managers, and engineers) had to be convinced of the benefits of metrics and had to be shown how to use them correctly.

- Too-Rapid Change: Grady cautioned against the tendency to try to change things too rapidly in what is essentially a slowly changing process. For example, the Hewlitt-Packard metrics effort found the Goal-Question-Metric paradigm useful in overcoming the "leap before you look" syndrome. Other drivers influencing too-rapid change were the "more is better" and "desperation for a breakthrough" syndromes.

- Organizational Changes: Organizational changes cause problems in transitioning new managers into an already-established metrics program. Getting a baseline in place can help with this kind of problem.

What Was Done Right

Grady next shared experiences in six areas that Hewlitt-Packard considers successes in its program:

- Bottom-Up Approach With Metrics Council: A hand-picked metrics council was formed, involving approximately 20 of the more experienced midline managers. Their goal was to identify a set of measures that they felt would be useful and meaningful in managing projects. This approach promoted "ownership" and participation in the program.

- Started Small: The effort started small, using the set of three primitive metrics described earlier. It is felt that starting small and building on that foundation was the correct approach.

- Creating an Environment for Reinforcing Success: This involved (1) putting in place a 2-day training program for all functional managers in the company, with the primary focus of teaching them about the fundamentals of software development as well as metrics; (2) establishing internal Software Engineering Productivity Conferences, with emphasis on reporting practical, measurable improvements in quality or productivity; and (3) identifying productivity managers.

- Establishing a Metrics Class.

- Providing Good Tool Support: A good set of minimal but adequate tools was developed to support the metrics collection effort.

Costs and Benefits

Grady conveyed that talking about specific costs and benefits misses the real issue; there is, in fact, no choice. Those organizations that *do* make the investment in a metrics program to better understand and improve their development process will have a more informed basis for making decisions and, thus, will have a competitive advantage.

He cited a specific example of the benefits of Hewlitt-Packard's applying metrics to the area of failure analysis. The causes of defects were tracked and categorized into defect types such as error checking, logic implementation, user interface, integration, software testing, standards, and data definition. Also identified were the software development phases, such as specifications/requirements, design, code, and environmental support in which the errors were occurring. This provided an overview of the major types of defects for each division, allowing improvement efforts to be focused in those areas with the most errors.

Future benefits are expected from continued use of failure analysis, as well as frcm continued pursuit of 10X improvement goals. Also, the long-term trend of reduced defects will have a significant impact in terms of shifting software effort from maintenance to new development.

## Ray Wolverton
## Hughes Aircraft Company, ITT

Wolverton presented an overview of the ITT programming measurement effort conducted from 1981 to 1986 at its Programming Advanced Technology Center. The goal was to invest $100 million over a 5-year period and earn the company a return of $1.1 billion through productivity improvements. Wolverton's particular job was to *prove* the actual cost savings. Programming measurement involved developing yearly baselines for progress comparison, a project performance reporting system for management, forecasting and diagnostic measurement procedures for use by project personnel, and an enhanced integrated measurement system. Over a 3-year period, information from 106 projects supported data collection, cost model development, and cost and quality trends demonstrations. A lesson from this experience was that **undertaking such an ambitious metrics program requires time.**

There were many programming measurement objectives, but four were the most prominent:

- Provide an early warning of project productivity, quality, schedule, or cost difficulties
- Improve the development and defense of competitive bids
- Compare ITT performance with overall industry performance
- Improve the allocation of resources

The measurement strategy involved dividing responsibilities between the "Programming" (metric study) group and the "Unit" group (individual ITT companies). The Programming group established and enhanced the methodology and tools and instructed the Units in their use. The Units established their responsibilities and collected and analyzed the data.

Activities needed to achieve the program's objectives included identifying baselines and leading performance indicators; formulating resource estimating techniques and quality profiles; promoting programmer/manager development in the metrics area; and conducting programmer competency and task analyses. Wolverton indicated ITT's priority focused on identifying **leading indicators**: they wanted an online tool that would allow ITT world headquarters to determine at will the performance of any of its world-wide units.

The overall strategy to study these performance factors involved the use of an online network measurement questionnaire. The effort collected basic data (defects, resources, and costs) and environmental data (requirements, practices, and products) and organized this information into a data base. Univariate analysis of these data highlighted 13 major factors found to affect programming productivity, quality, and cost. These factors included modern programming practices, programming personnel, the organizational structure, available tools, project complexity, and computer availability. Multivariate analysis was then performed by product category to produce the initial model.

The research identified strongly correlated groups of productivity factors. Higher productivity was influenced by many factors, including requirements specifications factors (personnel experience and the number of rewrites necessary); higher usage of modern programming practices; higher client experience and participation; lower staffing level; and larger target and development computers. It was also found that **all factors that improved productivity also improved quality**. In addition, the presence of one productivity factor was not sufficient to assure higher productivity/quality, but the lack of that factor was enough to guarantee the lack of high productivity/low error rates. Productivity trends and constraints were tracked to validate the results.

Fzctors were categorized as **controllable** (such as staffing level and experience, the development computer, requirements specifications); **uncontrollable** (such as the target computer, timing, memory utilization, and the application complexity); and **other variables** (such as incorrect data, newness of application/design, and documentation requirements). This categorization grouped information in terms that managers could understand. The 13 productivity factors explained 2/3 of the variation in productivity; when combined with developed statements, the 13 productivity factors explained 90 percent of the variation in the effort. **Knowing what can be controlled is important:** this effort determined that about 1/3 of the impact on productivity can be controlled.

One key finding identified by Wolverton's effort was the relationship between productivity and quality, with quality defined as "reduced defects." The number of testing defects/KLOC were plotted versus a productivity measure. A quality baseline was determined to be 20 errors/KLOC, based on the cumulative experience of the study; 90 percent of the projects experienced this defect level or less. Two projects were much higher in the number of defects; in both of these projects, there was a great deal of reuse. Wolverton cautioned that latent errors in untested reused code can cause this kind of result.

The Units reacted positively to the introduction of the metrics program, but indicated that many factors inhibited its effective application (insufficient resources and no time to collect data). However, when one Unit achieved positive results, other Units became interested in creating the same environment. Following initial negative feelings related to the effort and time required for acquiring data and testing proposed measures, the Units gained more confidence in the program and chose to continue an R&D phase of the project.

## Mitsuru Ohba
## IBM/Japan

The Japanese generally believe that "the things other people do are the right things to do." In the software context, "other people" could mean other organizations within a division, other divisions within a company, other companies in the industry, or other industries. Ohba explained that this belief promotes a great deal of learning from others, and that software measurement and analysis activities in particular have benefitted from such a perspective.

### Overview of Japanese Measurement Programs

The following are the standard measurements used in Japanese software measurement programs:

- Size (KLOC): Noncommented source lines of code, including reused source
- Productivity: LOC per programmer month
- Quality: Errors per KLOC

These measures are conceptually the same as those used in the United States and Europe.

### Obstacles Overcome

To establish a measurement system, there had to be agreement on what should be measured, including

- What kind of data should be collected
- How data should be analyzed (models, techniques)
- How results should be fed back into the management and development processes

The answers to these questions vary, because no standard measures exist. This has been a major obstacle for the IBM/Japan program. Much time has also been spent on determining the methods for data analysis. Software models are being heavily used to estimate the number of errors remaining in a program and to estimate the required maintenance effort. However, because so many commercial models are available, choosing the most appropriate models to be used requires much discussion.

### Program Cost

Ohba discussed several activities a centralized organization must perform to establish an effective measurement program: define measures and evaluation systems, define the data to be collected, choose simple methods for analyzing data, develop tools for collecting and analyzing data, maintain the data base, and provide education. In Japan, measurement systems are defined either by (1) a central software technology support group or (2) a quality assurance or equivalent organization. These groups define the measures and the way data are collected based on *de facto* standards or on working papers from various committees. Data collection and analysis are done by project groups, not by the centralized group. With all of these required activities, an effective measurement program is expensive.

Benefits to Date

Ohba stated that it typically takes at least 3 years to see the changes or benefits from a measurement program but by implementing a consistent measurement system, IBM/Japan has achieved the following gains:

- Management by quantitative objectives: setting objectives and reviewing achievements (e.g., software reliability growth estimation)
- Standard and consistent control of the software process by defining the upper and lower control limits
- Incremental and continuous process improvement by setting an annual goal for an organization

As a result of IBM/Japan's applying such software measurement techniques and measurement-based management approaches over the last 10 years, their defect rate has been reduced from 5 errors per KLOC to 0.1 error per KLOC.

Long-Range Benefits Expected

IBM/Japan's long-range goal is the **optimization** of its software process. This would involve the following:

- Design the best process for a project based on past experiences
- Properly monitor and manage that process based on quantitative data analysis/assessment
- Reconfigure the process dynamically, if needed, based on the data analysis results and experiences

Ohba described the "software factory" as an overlapping set of three elements: a process, methods/techniques used to implement the process, and tools/measurements to support the methods/techniques. In a specific application domain, these elements and their region of intersection can be standardized. This ideal "software factory" is the long-term benefit that can be achieved based on this measurement system.

## PANEL 2 – SOFTWARE ENGINEERING IN THE 1980S: MOST SIGNIFICANT ACCOMPLISHMENTS/GREATEST DISAPPOINTMENTS

The second panel of the workshop involved discussion by five experts in the field of software engineering. The first four panelists were Barry Boehm of the Defense Advanced Research Projects Agency Information Science and Technology Office (DARPA ISTO), Larry Druffel of the Software Engineering Institute (SEI), Manny Lehman of the Imperial College, and Harlan Mills of Software Engineering Technology, Inc. (SET). Vic Basili of the University of Maryland provided input as a fifth panelist and also moderated the panel. Each was asked to assess the progress of software engineering in the 1980s by addressing the following questions:

- What have been the most significant achievements for software engineering in the past 10 years?
- What have been the greatest disappointments for software engineering in the past 10 years?
- What are the objective or subjective criteria supporting your assessments?
- What software engineering advances will make the most significant contribution in the next 5 years?

The panelists found amazing similarities in their views of achievements and disappointments in software engineering in the 1980s. Top achievements agreed upon by most of the panelists included the study of process to improve it, the evolution toward object-oriented notions, the creation and standardization of Ada, the establishment and importance of metrics, the evolution of formal methods, the creation of the Software Engineering Institute, the improvement of life-cycle models, and the availability of CASE tools in industry. However, there was a dark side to many of the agreed-upon achievements. The disappointments most often cited were in Ada use and Ada education, CASE tool integration standards and use, and the lack of understanding and implementation of software engineering as a discipline in both industry and education.

Looking forward, the panel suggested the areas of most progress in the next decade. These include the maturation of object-oriented techniques, a focus on software architecture and the true engineering of software, more use and consistency of metrics, and process improvement through measurement.

In the lively question and answer session that followed, the panelists conjectured on how to improve future software. Barry Boehm focused on people: "Double your salary structure, and get rid of your unproductive people," he said, to the delight of the audience. Harlan Mills felt that we must increase the use of mathematical and formal approaches; he said, "We're not doing things the right way," but also agreed with Boehm that "We're not using the right kind of people." Larry Druffel suggested that designers must build changeability into the original design to lower maintenance costs (agreeing with Boehm that older software should be "obliterated" rather than maintained). Manny Lehman further supported this point by stating that, because society is becoming more and more dependent on software, we must also minimize the response time to change; he said that future software must be designed so that "changes can be made in real time," in response to the changing world that software supports. Vic Basili offered that we need to focus on process improvement, so that we can "predict and control future software development".

*Detailed summaries of each panelist's presentation follow.*

**BARRY BOEHM**

Barry Boehm, currently director of the Defense Advanced Research Projects Agency Information Sciences Technology Office (DARPA ISTO), was formerly chief scientist at TRW's Defense Systems Group. He has authored several books and won numerous awards in the areas of software engineering and software measurement.

Boehm summarized his list of the primary disappointments and achievements in software engineering during the 1980's as follows, noting that there are correlations between them:

| MAJOR ACHIEVEMENTS | MAJOR DISAPPOINTMENTS |
|---|---|
| STARS | STARS |
| OBJECT-ORIENTED METHODS | METRICS USAGE |
| THE SOFTWARE ENGINEERING INSTITUTE (SEI) | SOFTWARE ENGINEERING EDUCATION |

| PROJECTIONS FOR THE FUTURE |
|---|
| CONTINUED PROGRESS THROUGH THE SEI |
| MATURATION OF OBJECT-ORIENTED TECHNIQUES |

## STARS

Boehm stated that the Software Technology for Adaptable and Reliable Systems (STARS) program had a promising start in 1982, but, due to a variety of factors, progress in the mid- to late-1980s was disappointing. Now, however, Boehm feels that STARS is achieving progress in both the software engineering process and product areas. In 1986, the structure for defining the products sought by STARS was revised. These products would now be software support mechanisms focused primarily on tools but also on software reuse and the process by which tools should be applied. STARS placed particular emphasis on choosing really well-qualified people to perform the work. Contractors were selected based on demonstrated ability and their commitment to building a commercially viable software support environment. Today, companies such as IBM, Unisys, Boeing, and DEC are well on the way to producing the kinds of software environments that support reuse and provide good process models.

## THE SOFTWARE ENGINEERING INSTITUTE

Boehm stated that great progress was made in software process assessment during the 1980s. At the beginning of the decade, almost nothing was being done in this area. The Air Force Aeronautical Systems Division *did* produce some good checklists for capability assessments, and a team from the SEI created the SEI questionnaire for software process self-assessment. The thoroughness of the SEI's assessment questions and the fact that there is a bit of a carrot-stick approach involved prompted people to take software process concerns seriously. (Carrot = "here are some things that will make you better"; stick = "DOD will use these issues from time to time as source selection criteria"). Companies that have taken these considerations seriously are now doing a much better job of producing software.

However, Boehm feels the achievement in process assessment is not yet fully consummated. SEI process maturity Levels 1, 2, and 3 are defined quite well, but the definition of Level 5 is not very extensive. Thought is needed on clarifying the ultimate goal in terms of a software organization's process maturity expectations.

One of the biggest disappointments for the software engineering community was the assessment results. Eighty-five percent of the software organizations that were involved in self-assessment came out at the Level 1, or "chaotic" stage, the lowest level of the scale. Analysis of these results showed that it is easy to publish policies and standards and to do good briefings, but it is *difficult* and it takes a great deal of *commitment* for a company to follow through on these things and really *use* them to produce better software. The carrot-stick philosophy is helping to improve this deficiency.

## METRICS USAGE

Boehm went on to explain that one important characteristic of the top levels of the SEI software process maturity scale is that one's process be a **measured**, optimized process. This necessitates that a well-defined metrics program be integral to the software organization's manner of doing business. One of the biggest disappointments, however, is the lack of such metrics programs throughout the industry. In Boehm's opinion, people attending NASA/Goddard's Software Engineering Laboratory (SEL) conferences during the 1980s seem to be the only exception to this. During the period of 1979-1981, considerable excitement and interest in software metrics followed the introduction of the COnstructive COst MOdel (COCOMO), but since then relatively few organizations have adopted really comprehensive metrics collection and analysis programs. Reliability is the only area that appears to have received attention. In general, little progress has been made in convincing organizations that it is in their best interests to collect this information and analyze it as a prerequisite to improving their software process.

## OBJECT-ORIENTED METHODS

According to Boehm, there was little in 1980 that represented a different paradigm around which a software product could be designed and organized. However, object-oriented methods, technologies, and program languages gained prominence and began substantially to alter system designers' and developers' thinking. Potentially, a significant achievement in the 1990s is fully developing all of these object-oriented technologies into an integrated support environment.

## SOFTWARE ENGINEERING EDUCATION

Boehm indicated that another huge disappointment has been software engineering education. In 1980, a working group put together the IEEE model curriculum for a master's degree in software engineering that resulted in a careful, well-rounded program. Unfortunately, the IEEE did not follow through on it. A curriculum was subsequently created at the Wang Institute, where it was a strong success until the Institute's dissolution. Fortunately, the seeds of that work are reflected in the SEI's curriculum for a master's degree in software engineering. However, software engineering education at the undergraduate level is still abysmal—people are taught **programming**, not **software engineering**.

A significant achievement is that the SEI has picked up the gauntlet on such topics as education, process assessment, and researching real-time Ada systems development issues. Because of this progress, the software engineering community has a far better understanding of and access to these areas than was possible at the beginning of the 1980s. However, Boehm felt there is still much to be done and that the SEI will be a major contributor in the 1990s.

## LARRY DRUFFEL

Larry Druffel is the Director of the Software Engineering Institute. He has been associated with Ada since 1978 and was the first Director of the Ada Joint Program Office.

Druffel's presentation discussed four major disappointments and six major achievements in software engineering during the 1980s:

| MAJOR ACHIEVEMENTS | MAJOR DISAPPOINTMENTS |
|---|---|
| ADA | ADA |
| FOCUS ON PROCESS | SLOW ACCEPTANCE OF SOFTWARE ENGINEERING EDUCATION |
| SOFTWARE ENGINEERING EDUCATION | LACK OF TOOLS INTEGRATION AND STANDARDIZATION |
| SOFTWARE ARCHITECTURE | CODE REUSE |
| SOFTWARE ENGINEERING ENVIRONMENTS | |
| OBJECT-ORIENTED DESIGN NOTION | |

| PROJECTIONS FOR THE FUTURE |
|---|
| SOFTWARE ARCHITECTURE DESCRIPTION |
| MATURATION OF OBJECT-ORIENTED TECHNIQUES |
| IMPROVED DATA COLLECTION |

Druffel briefly addressed first the achievements and then the disappointments:

- **Ada**: Druffel stressed that the adoption of Ada as an ANSI and ISO standard must be considered one of the major achievements of software engineering in the 1980s. Benefits of the Ada standard are the tremendous strides seen in reuse and transportability and the enormous amount of private investment that has been made in the development of tools, the maturation of compilers, and the support of optimization techniques.

- **Focus on Process**: The focus on the software engineering process is an outstanding achievement. He cited the Space Shuttle work as a particularly good example of the program improvements achievable through such a focus. Druffel stressed that it is important that this focus on process be supported by measurement and education. The most successful efforts in improving process have had at least *ad hoc* measures to support them.

- **Software Engineering Education**: The principal achievement in the area c. software engineering education has been the emergence of the idea that there is now enough codified information that **software engineering can be taught**. The notion has become accepted and there are people willing to teach software engineering.

- **Software Architecture**: Perhaps one of the greatest achievements is that there is now a realization that software architecture needs attention—real analysis on the structure of a system.

- **Software Engineering Environments**: There has been marked progress in the past decade in recognizing the need for improved software engineering environments. In the past, everything was *ad hoc*, using a simple set of tools, with little or no appreciation for integrating these tools into a cohesive environment.

- **Object-Oriented Design Notion**: Druffel stated that we have seen the emergence of a whole new notion of managing objects and designing systems by objects, even though the notion of an object is not yet clearly understood or universally defined. The appearance of these object-oriented techniques and the way they are changing our approach to systems development are very important advances.

Druffel's list of major disappointments included the following:

- **Ada**: The Ada program has also yielded a major disappointment as well as major achievements during the 1980s. In Druffel's opinion, the major disappointment has been the failure of the software engineering community (including DOD, industry, and particularly the academic world) to take advantage of Ada as fully and as rapidly as could have been done.

- **Lack of Acceptance of Software Engineering in the Academic Community**: The disappointment in software engineering education is that the academic community at large doesn't seem to want to accept the idea of *engineering software*. Druffel stated that this limited perspective is holding the industry back.

- **Lack of Standardization in the Integration of Computer-Aided Software Engineering Tools**: Great progress has been made in the development of computed-aided software engineering tools. However, it is a disappointment that there has been a lack of any standardization to enable effective use on a project without a lot of manual intervention.

- **Code Reuse**: As in the decade of the 70s, code reuse remains a major area of disappointment. Some progress is visible in narrow application domains, but wider reuse will not be realized until domain analysis and designing for reuse are given more emphasis.

## SOFTWARE ENGINEERING ADVANCES IN THE NEXT 5 YEARS

Druffel discussed several projected software engineering advances for the coming decade:

- **Software Architecture**: Druffel postulated that software architecture description may be one of the major achievements in the next 5 years. There will be the capability to describe software architecture, agreement on the language and symbolic representation, and the development of analytical techniques to determine for which applications an **architecture is appropriate** and for which it is not.

- **Object-Oriented Design**: There should be a tremendous maturation of object-oriented design and related object-oriented techniques, allowing for their effective use.

- **Data Collection**: The next 5 years should see data collection follow a more structured approach and the development of consistent definitions to allow reasonable analysis across projects. This will be possible only because of the work done through NASA/ Goddard's series of Software Engineering Laboratory workshops and the contributions of those at the University of Maryland and the SEL to this effort.

MANNY LEHMAN

Manny Lehman is a professor of Computer Science at the Imperial College in England. He has done considerable work in software measurement and has authored over 100 technical papers.

Lehman presented his initial list of the most significant achievements and disappointments in software engineering during the 1980s, placing Computer Aided Software Engineering (CASE) tools in both categories:

| MAJOR ACHIEVEMENTS | MAJOR DISAPPOINTMENTS |
|---|---|
| CASE TOOLS | CASE TOOLS |
| RECOGNITION AND ACCEPTANCE OF THE EVOLUTIONARY NATURE OF SOFTWARE | LACK OF CASE PENETRATION INTO THE INDUSTRY |
| FOCUS ON PROCESS AND PROCESS MODELS | INADEQUATE CASE SUPPORT ENVIRONMENTS |
| MORE DISCIPLINED SOFTWARE ENGINEERING PROCESS | INADEQUATE UNDERSTANDING OF SOFTWARE ENGINEERING |
| **PROJECTIONS FOR THE FUTURE** | |
| IMPROVED, INTEGRATED CASE ENVIRONMENTS | |
| IMPROVED COST EFFECTIVENESS THROUGH CASE | |

Lehman's discussion of the most significant achievements included four items, all interrelated in a logical progression:

- The software engineering community is demonstrating an increasing appreciation of the intrinsically **evolutionary** nature of software.
- Clear evidence exists that the community has recognized the **significance** of the **software development process** and of **process models.**
- Wider appreciation and acceptance have been evident in the academic and research communities (and are now emerging in industry) of the importance of **discipline, method, formality, and mechanization** in the software process.
- The community has recognized the need for significant **CASE tools** and the **integrated support environments** to promote their effective use.

According to Lehman, although there may be a very solid understanding of a process, turning the process into reality and executing it with precision require discipline, method, and formality. These requirements clearly indicate the need for automated support, and while many tools have been emerging, industry has not yet provided the integrated support environments necessary to exploit their power.

CASE development has primarily arisen from the search for productivity growth in software development. Consequently, Lehman sees the largest disappointment to be the **failure of CASE to deliver this evident productivity growth.** Precisely because CASE tools have not shown the desired productivity benefits, their use hasn't **penetrated** widely or deeply into the industrial or commercial software development arenas.

Another disappointment has been the very slow development of **satisfactory, comprehensive, transferrable, and usable** support environments. Lehman believes, however, that the reasons for this are now sufficiently well understood. He projected that over the next decade there will be advances in this area, and that much wider penetration of CASE tools will then follow.

A final major disappointment has been the failure to achieve wide industry appreciation of the **true meaning of software engineering** and the **role of software engineers.** The community at large seems to use the terms 'software engineer' and 'programmer' interchangeably. Lehman stressed that these terms are not the same; they are two quite different yet complementary roles. Programming and the programmer are **product** oriented. The software engineer is primarily a **process** engineer, focusing on designing the processes the programmers use, on the methods to be followed, on the tools that can be applied, and on the organizational aspects of software development projects.

Lehman concluded with his prediction that CASE progress in the coming years will be significant, but only if people understand what CASE really is, how it can be transferred to industry, how it can be applied effectively, and how and when benefits can be assessed. Central to achieving this progress is the need for industry to develop the proper types of CASE environments through the *integration* of existing tools, creating comprehensive software development support environments.

In summary, Lehman emphasized that the primary goal of CASE *cannot* be immediate productivity growth, cost reduction, or visible improvements in product quality. Ironically, early CASE use will lead to increased costs, because training, tool and workstation acquisition, and production momentum loss all have an initial cost. What needs to be considered is what will happen in the long term: **improved cost-effectiveness over time associated with higher quality, more reliable, and more adaptable software is the ultimate benefit.** Realizing this benefit depends on using CASE long enough to aggregate small, invisible benefits into larger, visible benefits. The goal of CASE is institutionalizing a process that achieves and maintains user satisfaction with the software product, ultimately leading to productivity growth and revenue growth.

Despite its great potential, however, Lehman cautioned that obstacles to CASE progress include a long lead time before benefits will be realized; the need for major financial investments by the user; difficult cost-benefit analyses because of the many imponderables related to CASE; and a delay in truly quantifying the benefits of CASE until the industry gains significantly more experience in its use.

HARLAN MILLS

Harlan Mills is currently the president and chief technical officer of Software Engineering Technology, Inc. (SET). He has had a distinguished technical career, has authored over 50 refereed papers, and has won numerous software-related awards.

Several of the advances identified by Mills addressed process and methodology topics, while the disappointments cited deficiencies often characteristic of delivered software products:

| MAJOR ACHIEVEMENTS | MAJOR DISAPPOINTMENTS |
|---|---|
| SPIRAL DEVELOPMENT MODEL | LACK OF ENGINEERING DISCIPLINE FOR SOFTWARE DEVELOPMENT |
| METRICS | POOR-QUALITY SOFTWARE |
| SOFTWARE ENGINEERING INSTITUTE (SEI) | LOW PRODUCTIVITY |
| CLEANROOM DEVELOPMENT METHODOLOGY | MISSED SCHEDULES |

| PROJECTIONS FOR THE FUTURE |
|---|
| FORMALIZATION OF SPIRAL MODELS |
| SEI GROWTH |
| IMPROVED METRICS |
| EXPANDED USE OF CLEANROOM METHODOLOGY |

Mills discussed four areas he felt have represented significant achievements in software engineering over the past decade:

- **Spiral Model**: The spiral model of software development, articulated best by Barry Boehm, may replace the traditional waterfall model.
- **Metrics**: Significant developments in metrics for software technical management have emerged from Barry Boehm, Vic Basili, and other people in the field.
- **SEI**: A national resource has been established in the Software Engineering Institute.
- **Cleanroom Methodology**: Encouraging progress has been seen in applications of the Cleanroom approach to the engineering of software under statistical quality control, emphasizing the quality of people.

Mills' list of the greatest disappointments over the past decade stems from the following observation:

> "**Software engineering is used as a *buzzword*:** Software engineering isn't being treated as a real engineering discipline."

Mills feels that the people called "software engineers" today are typically programmers, and that they should be upgraded in some way through better training and the introduction of more discipline and mathematical formalism into the software development process. Mills continued that this perspective has led to the continued, widespread, and unnecessary existence of three conditions:

- **Poor quality, unreliable software**
- **Low productivity in software development:** It has been demonstrated that improvement in productivity by factors of 5 and 10 can be achieved when people "learn to do it right" as prcfessionals.
- **Missed schedule of software deliveries:** Getting things done on time is rarely achieved in the industry.

To further support his position, Mills discussed the properties of Cleanroom engineering and why this method is worthwhile:

- **Statistical Usage Specifications (As Well as Performance and Functional Specifications):** To talk about the reliability of software, Mills feels we need to know the ways in which that software is to be used. This knowledge will better guide us to test particular portions of the software more rigorously than other portions. We're testing in the wrong kinds of ways in the development shops and are waiting to see how the customer actually *uses* the software to determine where testing is really needed.
- **Software Development in a Pipeline of Increments with Separate Certification:** Software engineers are learning to avoid the need to debug programs. Cleanroom involves a pipeline of increments (such as 10,000 lines of code) that can be written without debugging and that are very close to error free.
- **Scaled-up Informal Verification of Software to Meet Specifications:** This involves using mathematical ideas, such as scaling up using axiomatic verification, etc., on 100,000 or 300,000 line programs.
- **Producing Software without Private Debugging Before Public Certification Testing:** The people who certify the software do the testing.

Mills identified four areas of potential, significant achievement in software engineering over the next decade:

- **Formalization of Spiral Models of Software Development for Procurement/ Management:** Software engineering has to have a mathematical foundation, and this is one good area with which to begin.
- **Metrics:** Continued development of metrics for software technical management.
- **Continued Growth of the SEI:** This involves extending its influence over software engineering.
- **Expanded Use of Software Engineering Under Cleanroom Engineering (or Something Similar), with Statistical Quality Control:** This requires using *real* engineering and not cut-and-dried programming.

VIC BASILI

Vic Basili, professor of Computer Science at the University of Maryland, has authored over 100 papers in software engineering, metrics, and methods, and is editor-in-chief of Transactions on Software Engineering.

As the facilitator for this panel and one of the principals in 15 years of SEL research, Vic Basili presented the SEL viewpoint on the most significant achievements and greatest disappointments in software engineering over the past 10 years:

| MAJOR ACHIEVEMENTS | MAJOR DISAPPOINTMENTS |
|---|---|
| IMPORTANCE OF PROCESS AND FORMAL METHODS | MATURING TAKING TOO LONG |
| COMMUNITY RECOGNITION OF NEED FOR MULTIPLE LIFE-CYCLE MODELS | INADEQUATE UNDERSTANDING OF "TECHNOLOGY BUILDING" |
| MEASUREMENT AND METRICS | MEASUREMENT NOT SUFFICIENTLY WIDESPREAD |
| OBJECT-ORIENTED METHODS | LIMITED AUTOMATED SOFTWARE DEVELOPMENT SUPPORT |
| ADA | FEW ADVANCES IN TESTING TECHNIQUES |

| PROJECTIONS FOR THE FUTURE |
|---|
| FOCUS ON ENGINEERING SOFTWARE |
| WIDER USE OF MEASUREMENT |
| REUSING PACKAGED EXPERIENCE |
| MATURING OF PERSONNEL SKILLS |
| INCREASED AUTOMATED SUPPORT |

Some of the earlier panelists had classified a given item as both an achievement and a disappointment. Basili noted that this was often because the item *was* an "achievement," but it hadn't moved fast enough or been adequately assimilated into the community and so was also considered a disappointment. Basili discussed the achievements in two categories:

- **Maturing**
  - **Process and Methods:** Increased recognition of the importance of process and formal methods has been a major achievement over the past 10 years. The SEI process assessment has raised the level of concern about process, and formal methods such as verification and Cleanroom have shown movement in the right direction.

- **Multiple Life-Cycle Models:** Recognition of the need for multiple life-cycle mode's and methods has been another major area of progress. There is no longer only a waterfall model; a spiral model and prototyping models have also been introduced. There are different ways of doing things. There is no longer a one-model mentality in the community.

- **Technologies**
  - **Measurement/Metrics:** Measurement and metrics techniques have matured, and the community now better understands how to apply them. The use of measures/ metrics may not be widespread enough, but from an SEL perspective the technology exists to do the kinds of things needed to reach Level 5 of the SEI software development maturity matrix.
  - **Object-Oriented Techniques:** The use of data abstractions and object-oriented methods has significantly altered the way current systems are being designed and implemented.
  - **Ada:** Ada's benefits are as much related to its promoting wider use of improved software engineering techniques as they are related to the language-specific features of Ada.

Basili discussed several disappointments, many of them parallel to items included in his achievements list:

- Maturing has taken so long.
- Some people are still looking for "magic"—a "silver bullet" that will provide a quick cure for our software development problems. What we must realize is that process improvements require **technology building, maturing, evolution,** and **evaluation.**
- Although some organizations have implemented measurement programs, a general lack **of measurement and formal methods** pervades the industry.
- Too **little effective automated support** for software development exists. CASE is an example of a disappointment in this category. CASE is a bottom-up issue, where pieces of technology have been dealt with but CASE hasn't solved what people are actually trying to do.
- While much attention has been given to the design and implementation areas of software development, **few advances in testing practices** have been made. There are many test techniques in the literature, but most are not available in practice. This is partly because the tests exist only at an academic level rather than as pragmatic ways of doing testing in the real-world.

Basili categorized future achievements as those to be attained over the next 5 years and those to be attained over the next 10 years.

- **Next 5 Years**
  - **Focus on engineering software:** This is the issue of discipline and evolutionary process.

- **Measurement:** Wider evidence of process improvement through measurement is expected.
- **Reuse of packaged experience:** This includes code, as long as it is done in the right context, i.e., as long as reuse is done for a product unit in the context of architecture and in the context of processes that produce the appropriate kinds of useful objects.

- Next 10 Years
  - **Real automated support:** This will require the natural evolution of supporting processes.
  - **Maturing of personnel:** The industry needs staff with consistent backgrounds. Over a long period of time, people who speak the same language and have similar kinds of training will become more prevalent in the software engineering community.

# SESSION 1—THE SEL AT AGE 15

V. R. Basili, University of Maryland

G. T. Page, CSC

F. E. McGarry, NASA/GSFC

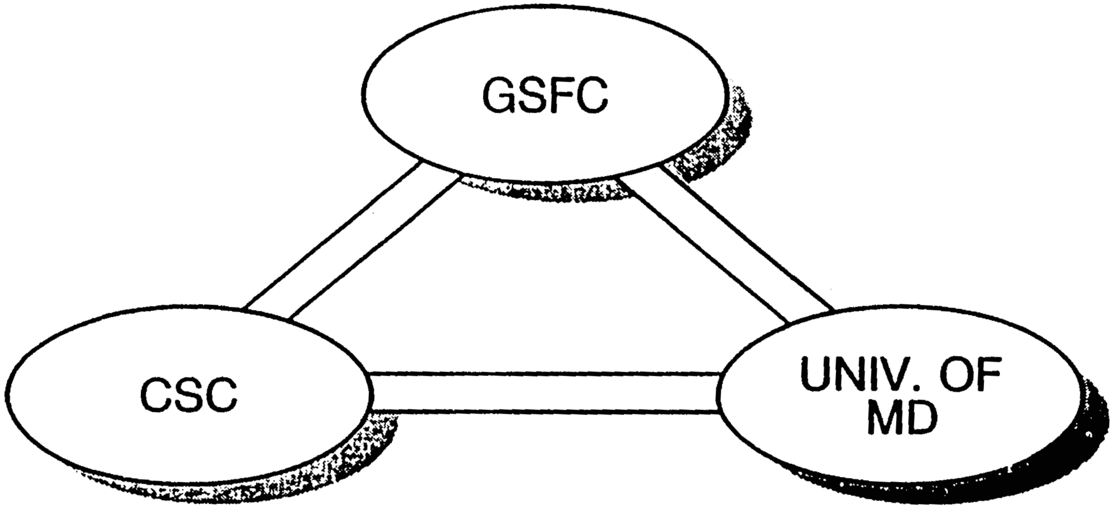
# VIEWGRAPH MATERIALS

# FOR THE

# F. MCGARRY INTRODUCTION

# TO SESSION 1

# SOFTWARE ENGINEERING LABORATORY (SEL)



F. McGarry-1
NASA/GSFC
Page 1 of 5

A498.003

# SOFTWARE ENGINEERING LABORATORY (SEL) BACKGROUND

- EARLY 1970's
  - INCREASING REALIZATION OF SOFTWARE ROLE (COST, IMPORTANCE,...)
  - NUMEROUS SOFTWARE TECHNOLOGIES EVOLVING/MATURING
    - TOOLS (DEBUG AIDS, CODE ANALYZERS,...)
    - STRUCTURED DESIGN/ANALYSIS/IMPLEMENTATION...
    - TESTING/VERIFICATION TECHNIQUES
    - MANAGEMENT METHODS AND AIDS
  - NO GUIDANCE FOR SELECTING/APPLYING EVOLVING SOFTWARE TECHNIQUES
    - LIMITED EMPIRICAL EVIDENCE
    - OVER DEPENDENCE ON MOST VOCAL ADVOCATES
- 1975 - 76'
  - PARTNERSHIP BETWEEN NASA/GSFC AND UNIVERSITY OF MARYLAND FORMED
    - EXPERIMENT WITH AVAILABLE TECHNIQUES IN PRODUCTION ENVIRONMENT (NASA/GSFC)
    - DETERMINE WHICH TECHNIQUES ARE EFFECTIVE (MEASUREMENT)
    - INFUSE IDENTIFIED TECHNOLOGY BACK INTO PROCESS (PROCESS IMPROVEMENT)
  - CSC - AS PRIMARY FLIGHT DYNAMICS SOFTWARE CONTRACTOR BECOMES 3rd PARTNER
- 1976 - 90'
  - CONSISTENT PARTNERSHIP IN SEL (NASA/UM/CSC)
  - EVOLUTION TOWARD UNDERSTANDING SOFTWARE AND OPTIMIZING PROCESS

F. McGarry-1
NASA/GSFC
Page 2 of 5

# SOFTWARE ENGINEERING LABORATORY
# ESTABLISHED 1975 - 76'

- **GOALS**
  - UNDERSTAND THE SOFTWARE PROCESS IN A PRODUCTION ENVIRONMENT
  - DETERMINE IMPACT OF AVAILABLE TECHNOLOGIES
  - INFUSE IDENTIFIED/REFINED METHODS BACK INTO DEVELOPMENT PROCESS

- **APPROACH**
  - IDENTIFY TECHNOLOGIES WITH HIGH POTENTIAL
  - APPLY AND EXTRACT DETAILED DATA IN PRODUCTION ENVIRONMENT (EXPERIMENTS)
  - MEASURE IMPACT (COST, RELIABILITY, QUALITY,...)

- **SIGNIFICANT CONSIDERATIONS**
  - CANDIDATE PROJECTS (IMPACTS/COST/...)
  - DATA/INFORMATION (DEFINING/PROCESSING/VALIDITY/...)
  - EXPERIMENTAL DESIGN (CLASSES OF STUDIES/DOMAIN ANALYSIS...)

# SEL
# PRODUCTION ENVIRONMENT

SOFTWARE CHARACTERISTICS

- SCIENTIFIC (FLIGHT DYNAMICS)
- GROUND BASED (NON-EMBEDDED)
- INTERACTIVE

LANGUAGES

- 75% FORTRAN
- 15% Ada
- 10% OTHER (C, PASCAL, LISP, ...)

PROJECT CHARACTERISTICS

| | TYPICAL |
|---|---|
| ● DURATION (MONTHS) | 24-40 |
| ● EFFORT (STAFF YEARS) | 30-45 |
| ● SIZE (KSLOC) | 100-300 |
| ● STAFF (FTE) | 5-15 |

*SEVERAL PROJECT EXCEED 200 STAFF YEARS
*SEVERAL LESS THAN 3 STAFF YEARS

*HOMOGENEOUS CLASS OF SOFTWARE

*CONSISTENT SUPPORT ENVIRONMENT

*CONTROLLED PROCESS

F. McGarry-I
NASA/GSFC
Page 4 of 5

A498.006

# SEL
# EVOLVING “PROCESS IMPROVEMENT” ENVIRONMENT
# WHAT HAS BEEN LEARNED?

1. HOW HAS THE MODEL OF PROCESS IMPROVEMENT EVOLVED?
   - PHASES TOWARD PROCESS IMPROVEMENT
   - WHAT CLASSES OF EXPERIMENTS/STUDIES ARE NEEDED
   - DO WE UNDERSTAND THE IMPROVEMENT PARADIGM BETTER TODAY

2. WHAT IMPACT HAS THE EXPERIMENTATION HAD ON A PRODUCTION ENVIRONMENT?
   - IS SOFTWARE DEVELOPED DIFFERENTLY NOW (BETTER?)
   - HAS THERE BEEN A CHANGE IN ATTITUDE OF DEVELOPERS
   - WHAT CHANGES HAVE BEEN MADE AS A RESULT OF 'STUDIES'

3. WHAT HAS BEEN LEARNED FROM THE STUDIES?
   - ARE THERE TECHNIQUES THAT HELP (WHICH ONES)
   - WHAT ARE KEY ASPECTS OF PROCESS IMPROVEMENT
   - WHAT ARE MAJOR IMPACTS/LESSONS FOR NASA

N92
19421
UNCLAS

# Towards a Mature Measurement Environment: Creating a Software Engineering Research Environment

Victor R. Basili
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland

## Software Engineering Research

Software engineering researchers are building tools, defining methods and models. However, there are problems with the nature and style of the research. The research is typically bottom-up, done in isolation so the pieces cannot be easily logically or physically integrated. A great deal of the research is essentially the packaging of a particular piece of technology with little indication of how the work would be integrated with other pieces of research. The research is not aimed at solving the real problems of software engineering, i.e., the development and maintenance of quality systems in a productive manner. The research results are not evaluated or analyzed via experimentation or refined and tailored to the application environment. Thus, it cannot be easily transferred into practice. Because of these limitations we have not been able to understand the components of the discipline as a coherent whole and the relationships between various models of the process and product.

What is needed is a top down experimental, evolutionary framework in which research can be focused, logically and physically integrated to produce quality software productively, and evaluated and tailored to the application environment. This implies the need for experimentation, which in turn implies the need for a laboratory that is associated with the artifact we are studying. This laboratory can only exist in an environment where software is being built, i.e., as part of a real software development and maintenance organization. Thus we propose that Software Engineering Laboratory (SEL) type activities exist in all organizations to support software engineering research.

In this paper we will try to describe the SEL from a researcher's point of view. Jerry Page and Frank McGarry will discuss the corporate and government benefits of the SEL. I will try to focus my discussion on the benefits to the research community.

## The SEL as a Research Laboratory

The SEL is a laboratory that allows us to understand the various software processes, products and other experiences, build descriptive models of them, understand the problems associated with building software, develop solutions focused on the problems, experiment with the proposed solutions and analyze and evaluate their effects, refine and tailor these solutions for continual improvement and effectiveness and enhance our understanding of their effects, and build relevant models of software engineering experiences.

The SEL has been in business for over 15 years and, based upon our experiences, its activities have evolved over time. In this section, I will describe the activities as they progressed over three phases.

The first phaseI will call the **understanding** phase because we worked on understanding what we could about the environment and measurement. During this period we measured what we could, used available models to explain the environment and our behavior, and built descriptive baselines and models typifying our environment.

In retrospect we made several mistakes. We collected too much data, i.e., because we did not know what was important we tended to collect all kinds of data hoping they would give us insights into the environment. We often blindly applied models and metrics without understanding the subtle assumptions and whether they were relevant in our environment. In a sense, we tried to evaluate things before we had built a deep understanding of what we were evaluating. We finally began to understand that measurement needed to be based upon models and goals. We established goals and a mechanism for generating measures based upon those goals, the first, primitive version of the Goal/Question/Metric Paradigm. This provided an informal approach to organizing our data. Based upon our goals, we began to build environment specific models by accumulating knowledge on individual projects and building baselines across multiple projects. Eventually we developed descriptive models that characterized the environment. These models included models of resources, defects, and product charactoristics.

Once we had an understanding or characterization of the environment and the projects we were developing, we were able to begin the process of evaluation by comparing new projects against our baselines. This allowed us to proceed to phase two where the focus was on **improving** the process, product, and environment. During this phase, we continued to build up our data base of baselines and models, but we also evaluated and fed back information to the project. Many of these early data models were informal. The data was saved in a data base but the models existed mostly in documents. We began to experiment with various technologies to understand their effect, i.e. how they changed the baselines or the models we had. In order to provide a learning process across projects that would allow us to take advantage of what we had learned and evolve, we developed the Quality Improvement Paradigm, which is based upon an evolutionary, experimental approach to software improvement based upon both project and organizational feedback loops. The Goal/Question/Metric Paradigm continued to evolve to recognize different types of goals and questions and take advantage of the multi-project perspective. We began formalizing process, product, knowledge and quality models.

This need for formalization within the context of the Improvement Paradigm led to the concept of **packaging** models of our experiences so they were reusable on other projects. During this third phase we worked on choosing potentially reusable experiences, recognizing what was appropriate and relevant for the SEL. We began studying notations and mathematical formalisms for defining experiences.

There are several examples of current research projects in packaging experiences. For example, we are working on a project characterization model that allows us to recognize project patterns so that we can predict which projects look like the one we are working on. This allows us to package data for use as cost estimation models based upon our relevant past history [Briand, Basili, Thomas]. Having recognized that most experiences need to be modified for use, we have been defining models of tailorable experiences. For example, we are working on a tailorable test method [Basili, Martschenko, Swain]. The method allows one to choose the appropriate test techniques based upon the defect history of similar projects and the success rate of the techniques in that environment. Another example is the development of a model or reference architecture for different types of software factories [Basili, Caldiera and Cantone]. We are defining process models for reusing experience. We have developed a reuse- oriented evolution model [Basili and Rombach] and are working on integrating experience models [Oivo and Basili]. We have developed the concept of an Experience Factory, whose goal is to package software experiences and provide them to projects upon demand and have integrated the concept with an evolved QIP and GQM.

| | | |
|---|---|---|
| | | **Packaging** |
| | | SEL Ada Process |
| | | SEL Cleanroom Process |
| | | SME |
| | | Managers Handbook |
| | | Experience Factory |
| | **Improving** | |
| | Methodology Evaluation | Ada |
| | Cost Model Analysis | OOD |
| | Test Technique Analysis | Cleanroom |
| | QIP | CASE |
| **Understanding** | | |
| Modeling environment | Design Measures | Test Method |
| Data Collection (GQM) | Cost vs. Size Complexity | Reuse |
| Resource Baselines | | |
| Defect Baselines | | |

**Figure 1. Evolution of Measurement/Studies in the SEL**

Figure 1 represents some of the studies we performed and the hierarchy of the process, one phase based upon the other. That is, there was an understanding process (Phase 1), followed by an improving process (Phase 2), followed by a packaging process (phase 3). You can't improve until you understand, and you can't package until you can assess and improve. We are still understanding and trying to improve; these activities, along with packaging, will go on forever.

## The Research Framework Concepts

We have evolved to a framework [Basili b] that is based on three basic concepts, each of which is itself evolving:

o The Quality Improvement Paradigm (QIP), an evolutionary improvement paradigm, based upon the scientific method, tailored for the software engineering,

o The Goal/Question/Metric (GQM) paradigm, an approach for establishing project, corporate, and research goals and a mechanism for measuring against those goals,

o The Experience Factory, an organization that supports research and development by studying projects, developing and refining models, and supplying them to projects for further analysis and refinement.

The **Quality Improvement Paradigm** consists of the following steps:

o Characterize the current project and its environment with respect to a variety of models.

o Set the quantifiable goals for successful project performance and improvement.

o Choose the appropriate process model and supporting methods and tools for this project.

o Execute the processes, construct the products, collect and validate the prescribed data, and analyze it to provide real-time feedback for corrective action.

o Analyze the data to evaluate the current practices, determine problems, record findings, and make recommendations for future project improvements.

o Package the experience in the form of updated and refined models and other forms of structured knowledge gained from this and prior projects and save it in an experience base so it represents our current state of knowledge and is available for future projects.

The research emphasis is on taking each of these issues associated with the QIP, (e.g., characterizing, goal setting, choosing process, executing, analyzing, and packaging), and formalizing and integrating them. Each of these steps has evolved over the years. We have been building models of characterization. For example, what are good models that allow me to recognize what kind of software project I have and what projects are similar? Based on data, we are using pattern recognition techniques to recognize where to find the most appropriate kinds of experiences related to the current project [Briand, Basili, Thomas].

Goal setting has become a process of integrating models. A goal typically takes the form of analyzing some form of object from some perspective. I need models of both the object of study and the various perspectives of interest on that object.

We want to choose processes. A key issue here is that process is a variable; that I need to select, manipulate and change processes based on the characterization of the project and the environment and the goals established for this particular project.

Execution needs automated support. An automated system, SME, has been developed to support the accessing of data in a packaged form. The analysis and packaging issues are the major focuses of this paper.

The **Goal/Question/Metric Paradigm** is a mechanism for defining and interpreting operational and measurable software goals. Goals may be defined for any object, for a variety of reasons, with respect to various models of quality, from various points of view, relative to a particular environment. A particular GQM model combines models of an object of study, e.g., a process, product, or any other experience model and one or more focuses, e.g., models aimed at viewing the object of study for particular characteristics, such as models of cost, correctness, defect removal, changes, reliability, user friendliness, etc. This implies that there are models of these quality perspectives developed and available for use at anytime.

These models can be analyzed from a point of view, e.g., the perspective of the person needing the information, which orients the type of focus and when the interpretation of the information is made available and for any purpose, e.g., characterization, evaluation, prediction, motivation, improvement, which specifies the type of analysis necessary.

The result is a GQM model relative to a particular environment. Environments are distinguished based upon a variety of factors, e.g., problem factors, people factors, resource factors, process factors, etc.

## Experimental Approaches

Given a form of the scientific method, in the guise of the QIP, a mechanism to generate research hypotheses, in the guise of the GQM, what kinds of experimentation can we perform? The chart in Figure 2 offers four classes of studies that we can and have performed. The approaches can be characterized by the number of teams replicating each project and number of different projects analyzed.

| | | #Projects | |
|---|---|---|---|
| | | One | More than one |
| # of Teams per Project | One | Single Project (Case Study) | Multi-Project Variation |
| | More than one | Replicated Project | Blocked Subject-Project |

**Figure 2. Classes of Studies and Scopes of Evaluation**

The single project case study is where most people begin. There is a project and someone has decided to study it. The results can provide some insight into project development in the environment.

A multi-project variation type study involves the measurement of several projects where factors, such as a method, can be varied across similar type projects. This allows the experimenter to study the effects of variations to the extent that the organization allows them to vary on different projects. In fact, that's literally what we do in the SEL. We have a large number of projects, we have standard baselines of how things should happen, and we start to perturb them by making changes and studying the effects of those changes.

The replicated project study involves several replications of the same project by different subjects. Each of the issues studied is applied to the project by several subjects but each subject applies only one of the technologies. It permits the experimenter to establish control groups.

The blocked subject-project study allows the examination of several factors within the framework of one study. Each of the issues studied is applied to a set of projects by several subjects and each subject applies each of the technologies under study. It permits the experimenter to control for differences in the subject population as well as study the effect of the particular projects.

There are two problems with the controlled types of experiments: (1) they are rather expensive and (2) if done for large pieces of software, for example, one year duration projects, they are hard to control, especially over several replications. Therefore, even though these types of experiments generate stronger confidence in the results than the non-controlled type experiments, they must be performed on small projects so the results do not scale up. If, however, these experiments are run on a small scale achieving reasonable statistical results, then there is motivation to experiment with the technologies on a larger scale in either a case study or a multi-project variation. Combining the results of the controlled experiment and the large- scale case study or multi-project variation, we can gain confidence in the validity of the experimental results.

It is clear in the SEL that we are avid believers in experimentation. We do not believe that any technology, method, tool, process model, etc. works under all circumstances. Everything has limits, areas where it works well or poorly. If we are dealing with technologies, we know they have limits. Experimentation is important in understanding those limits.

<table>
<tr><th>Single Project<br>(Case Study)</th><th>Multi-Project<br>Variation</th></tr>
<tr><td>Independent V&V</td><td>Effect of Methodology</td></tr>
<tr><td>Cleanroom Process</td><td>Resource Model Studies</td></tr>
<tr><td colspan="2">Defect Analysis Studies<br>Ada/Object Oriented Design<br>Code Reuse in Ada/Fortran</td></tr>
<tr><th>Replicated<br>Project</th><th>Blocked<br>Subject-Project</th></tr>
<tr><td>Effect of Methodology</td><td>Reading vs. Testing</td></tr>
<tr><td>Cleanroom Process</td><td></td></tr>
<tr><td>Ada/O-O Design</td><td></td></tr>
</table>

**Figure 3. Example Classes of Studies**

Figure 3 contains several example studies we have performed in the SEL. These studies cut across various experimental classes. When we have found something effective as a case study, we eventually turn it into a multi-project variation because it is effective for the environment.

## An Example Set of Studies

As an example of an effective process with which we have performed multiple types of experiments, consider the Cleanroom approach to software development, as suggested by Harlan Mills. We first ran a replicated project study at the University of Maryland that showed that the approach was very effective. We then decided to run a case study here in the SEL, which again was successful. We have since begun two new projects using the approach and will eventually have enough projects for an analysis based upon multi-project variation.

The key elements of the Cleanroom Process [Dyer], include a mathematically-based design methodology which includes: function specification for programs, state machine specification for modules, reading by stepwise abstraction, correctness demonstrations when needed, and top-down development. The implementation is done without any on-line testing by the developer. There is statistically-based, independent testing, based on anticipated operational use. Testing is done from a quality assurance orientation.

The replicated Cleanroom study had as its goals to evaluate the Cleanroom process with respect to its effects on the process, product and developers relative to differences from a non-Cleanroom process [Selby, Basili, Baker]. The experiment was run at the University of Maryland with 15 three-person teams,10 using Cleanroom. The project was an electronic message system (~ 1500 LOC). The teams were permitted 3 to 5 test submissions and the data collected consisted of background and attitude surveys, on-line activities of the developers, and test results.

The effect of the Cleanroom approach on the process was that the Cleanroom developers (1) felt they more effectively applied off-line review techniques, while others focused on functional testing, (2) spent less time on-line and used fewer computer resources, and (3) tended to make all their scheduled deliveries.

The effect of the Cleanroom approach on the product with regard to static properties was that the products developed using the Cleanroom approach had less dense complexity, a higher percentage of assignment statements, more global data, and more comments. With regard to operational properties, Cleanroom products more completely met requirements and had a higher percentage of test cases succeed.

Based on these results, we decided that it was worth running a case study in the SEL to see if the approach scaled up and how it worked with changing requirements. In applying the approach in the SEL, you will see an application of the QIP with regard to improving process. We begin with the characterization step which asks the question, "what relevant models exist that are available for reuse?" There were three models: the standard SEL model, which defines how software gets developed in the SEL in a

FORTRAN environment; the IBM FSD Cleanroom model that was applied on a prior project, and the experimental model we used for the replicated project.

The SEL goals were to characterize and evaluate the Cleanroom approach in general and specifically with regard to changing requirements. In prior applications, Cleanroom had been used on projects where the requirements were basically fixed at the beginning of the study. One of the questions we were often asked after the replicated project study was "would this technology survive in an environment with changing requirements?" Since we had not experimented with changing requirements, we could not answer the question with much confidence.

What had been learned from the IBM/Cleanroom model application was the basic process model, methods and techniques and that the process very effective in the given environment. From the UoM/Cleanroom model application, we learned that no developer testing enforces better reading, the process is quite effective for small projects, formal methods are hard to apply and require skill, and there may be insufficient failure data to effectively measure reliability.

Based upon the existing models, our goals, and the lessons learned from prior applications of Cleanroom, we defined an initial SEL Cleanroom process model. We stole what was most effective from prior applications; for example, the training was consistent with the University of Maryland course and we emphasized reading by at least two reviewers.

Because this was a real project, and there was concern on the part of some of the developers about the effectiveness of reading, e.g., that you needed to test certain algorithms, we allowed back-out options, e.g., you could request permission to unit test certain types of algorithms. These back-out options were never used, but they did provide a comfort level for the developers. When we didn't know how to hande some aspect of the approach in this environment we applied the standard SEL process model as long as it didn't conflict in principle with what we were trying to do. We monitored and made changes to the process model in real-time. We wrote lessons learned, and we redefined the process for the next time out.

Some of the major positive results of the application of Cleanroom in the SEL include: the approach scales up to a 30,000 SLOC project, it can be used with changing requirements, productivity increased by about 30%, the failure rate during test reduced to close to 50%, there was a reduction in rework effort (95% of the fixes, as opposed to 58%, took < 1 hour to fix), only 26% of faults found by both readers (implying two readers are important), there were effort distribution changes, e.g., more time in design and 50% of code time spent reading, code appears in library later than normal and more like a step function, there was less computer use by a factor of 5.

Negative lessons learned include the fact that better training was needed for the methods and techniques. The kind of training we had at the university wasn't good encugh. For example, we provided training where the examples were stacks, etc. This was not appropriate for the application. (One thing we have done on the second two Cleanroom projects is reuse parts of the first project as examples in the training.) We needed better mechanisms for transferring code to testers and the testers need to add requirements for output analysis of code. As expected, we did not have enough error data (with a 30,000 line project) to seed the reliability model so there was no payoff in reliability modeling in the SEL.

A side effect of this project was that it generated much more interest in improving the requirements. This requirements problem existed independent of Cleanroom, but the approach exposed the problem. So there has been a genuine push in having better defined requirements.

These results were for a 30,000 line project and a particular application. Is that the size limit for the Cleanroom process? Suppose we try a 100,000 line project ... what are the limits of this particular technology? When does it start to fall apart? Even if it doesn't work for a given size project, that's okay ... we now understand the bounds on that technology. It should not be expected that a technology works under all circumstances, every time, and every place. We have to understand as a community that technology has limits and that we have to select, and modify processes appropriate for the situation.

The next two experiments will emphasize the application of the formal models more, we are using the box structure approach, a change in the application domain for one project, and a scale up to a 100 KLOC for the other project.

This has been an example of the Quality Improvement Paradigm in terms of a particular process, and in terms of experimental design moving from controlled experiments to case studies in a real environment, and moving from case study to multi-project environment.

And we continue to evolve.

## Packaging the Experience

We have just discussed a form of packaging, the documentation of the Cleanroom process model. We currently have a working document that represents the model as we understand it today. But it will change as we learn!

Packaging experience requires the continual accumulation of evaluated experiences (learning) in a form that can be effectively understood and modified (experience models) into a repository of integrated experience models (experience base) that can be accessed and modified to meet the needs of the current project (reuse).

Systematic learning requires support for recording experience off-line generalizing and tailoring of experience formalizing experience. Off-line is a key word here. Packaging cannot be done as part of a project development. Someone cannot perform this analysis and build models at the same time they are building software. There needs to be a separate organization, either physically or logically separate.

Packaging useful experience requires a variety of models and an experience base. The models require formal notations that are tailorable, extendible, understandable, flexible and accessible. An effective experience base must contain accessible and integrated set of analyzed, synthesized, and packaged experience models that captures the *local* experiences.

The **Experience . Factory** is a logical and/or physical organization (sepa ate from the project organization) that supports project developments by analyzing and synthesizing all kinds of experience models acting as a repository for such experience supplying that experience to various projects on demand. It packages experience by building informal, formal or schematized, and productized models and measures of various software processes, products, and other forms of knowledge via people, documents, and automated support.

There are a variety of softwa:e engineering experiences that we can package: resource baselines and models, change and defect baselines and models, product baselines and models, process definitions and models, method and technique models and evaluations, products, lessons learned, quality models, etc. In the SEL, they exist in the form of standards, policies, tools. The documents range from sets of lessons learned to a manager's handbook.

There are many forms of packaged experience. We can use mathematical equations defining the relationship between variables, e.g., Effort = $a*Size^b$. We can present raw or analyzed data in the form of histograms or pie charts, e.g., % of each class of fault. We can plot graphs defining ranges of "normal", e.g., graphs of size growth over time with confidence levels. We can write specific lessons learned associated with project types, phases, or activities, e.g., reading by stepwise abstraction is most effective for finding interface faults, or in the form of risks or recommendations, e.g., definition of a unit for unit test in Ada needs to be carefully defined. We can create models or algorithms specifying the processes, methods, or techniques, e.g., an SADT diagram defining Design Inspections with the reading technique a variable dependent upon the focus and reader perspective.

For example, in the SEL we have a whole set of equations that define the relationships between a variety of variables [Basili, Panlilio-Yap]. Management can use these equations to understand, predict, and evaluate. In the SEL, example packaged relationships include:

Effort = 4.37 + 1.43devlines
Effort = 5.5 + 1.5newlines

Docpages = 99.1 + 30.9 devlines
Numruns = -108 + 151devlines
For projects under 50 KLOC we have:
Effort = .877 + 1.5newlines
while for projects over 50 KLOC we have:
Effort = 66.9 + .003 numruns
We have been able to demonstrate that methodology favorable impacts software cost and quality but cumulative complexity unfavorable impacts these factors [Basili a].

We have fault profiles that allow us to compare and analyze environments and projects. For example, what percent of faults of a particular type, based on a particular classification scheme, occur during a standard FORTRAN development. Are the percentages the same for an Ada development? We have been able to show that Ada reduces the percent of interface faults, but not by the amount one might expect based upon the ability of Ada compilers to check for interface faults [Brophy].

## Conclusions

Based upon our experiences, we need a set of experience factories or SELs, each focused on packaging local experiences by building and tailoring local models, integrating technologies, studying scale-up, building experience bases, and developing automated aids.

It is still hard to answer questions like: how big should an SEL be? should the experience factory only be domain specific, should it focus on a homogeneous environment?

If the SELs are focussed on homogeneous environments, we will need to integrate these local experience factories into a high level experience factory that abstracts from local experiences, looks for patterns across environments, and generates the basic models of the science. But how is this accomplished?

What we can do now is take advantage of the experimental nature of software engineering. Processes, products, and environments can be measured and can be used to support practical development and research. The integration of the Improvement Paradigm, the Goal/Question/Metric Paradigm, and the Experience Factory Organization can provide a framework for both development and research.

Based upon our experience, it helps us derive descriptive models of our experiences, understand our experiences and our problems, evaluate and learn from our experiences, and build effective prescriptive models of our experiences and our quality objectives. It can and should be applied today and evolve with technology.

Taking advantage of the experimental nature of software engineering has provided a winning situation for research and development. From a researcher's perspective the

SEL has been a smashing success. Its evolution has been slow, we have made many mistakes, but we have learned a lot.You don't have to make the same mistakes we did, you can learn from our experiences.

## References

[Briand, Basili, Thomas]
Briand, L.C., Basili, V. R., Thomas, W. M., A Data Analysis Procedure for an Effective Application of the Improvement Paradigm, University of Maryland Technical Report, March 1991.

[Basili](a)
V. R. Basili, "Can We Measure Software Technology: Lessons Learned from 8 Years of Trying," Proceedings of the Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, December 1985.

[Basili](b)
V. R. Basili, "Software Development: A Paradigm for the Future," Proc. 13th Annual International Computer Software & Applications Conference, Orlando, FL, September 20-22, 1989

[Basili, Martschenko, Swain]
Basili, V. R., Martschenko, W. N., and Swain, B. J., A Framework for Goal Directed Process Planning, University of Maryland, working paper.

[Basili, Caldiera and Cantone]
V. R. Basili, G. Caldiera, and G. Cantone, A Reference Architecture for the Component Factory, University of Maryland Technical Report CS-TR-2607, March 1991

[Basili, Panlilio-Yap]
V. R. Basili, N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," IEEE COMPSAC, October 1985.

[Basili, Rombach]
V. R. Basili and H. D. Rombach, "Support for Comprehensive Reuse," accepted for publication in Software Engineering Journal, IEE and British Computer Society, July 1991, and also UMIACS-TR-91-23, CS-TR-2606I, February 1991

[Dyer]
M. Dyer, "Cleanroom Software Development Method," IBM Federal Systems Division, Bethesda, Maryland, October 14, 1982.

[Oivo and Basili]
ES-Tame: A Prototype Implementation of the TAME Experience Base, University of Maryland, working paper.

[Selby, Basili, Baker]
R. W. Selby, Jr., V. R. Basili, and T. Baker, "CLEANROOM Software Development: An Empirical Evaluation," IEEE Transactions on Software Engineering, Vol. 13 no. 9, September, 1987, pp. 1027-1037.

[Brophy]
C. Brophy, "Lessons Learned in the Transition to ADA from Fortran at NASA/Goddard," UMIACS-TR-89-84, CS-TR-2305, August 1989

# VIEWGRAPH MATERIALS

# FOR THE

# V. BASILI PRESENTATION

# Towards a Mature Measurement Environment:

# Creating a Software Engineering Research Environment

Victor R. Basili

Institute for Advanced Computer Studies

Department of Computer Science

University of Maryland

# Software Engineering Research

There is a great deal of software engineering research going on, i.e., people are building technologies, methods, models, etc.

**What is the problem?**

The research is mostly bottom-up, done in isolation

It cannot be easily logically or physically integrated

It is not aimed at solving the big problem

It is not evaluated or analyzed via experimentation

It is not refined and tailored to the application environment

It cannot be easily transferred into practice

We cannot understand the relationships between various models of the process and product

# Software Engineering Research

## What is needed?

A top down experimental, evolutionary framework in which research can be focused, logically and physically integrated to produce quality software productively, and evaluated and tailored to the application environment

An experimental laboratory that is associated with the artifact we are studying

*We need SEL type activities to support software engineering research*

# What is the SEL

## from a researchers point of view?

A laboratory that allows us to

understand the various processes, products and other experiences and build descriptive models

understand the problems associated with building software

develop solutions focused on the problems, experiment with them and analyze and evaluate their effects

refine and tailor these solutions for continual improvement and effectiveness and enhance our understanding of their effects

build models of software engineering experiences

# How have the activities evolved?

Evolving concepts for over 15 years

Phase 1

Worked on **understanding** what we could about the environment and measurement

- measured what we could
  - collected too much data
- used available models
  - blindly applied models and metrics
  - tried to evaluate before understanding
- built descriptive baselines and models
  - studied individual projects
  - tried to characterize the environment
- developed the Goal/Question/Metric Paradigm
  - informal approach to organizing data

# How have the activities evolved?

Phase 2

Worked on **improving the process and product**

evaluated and fed back information to project

mostly informal data models

data automated but not the models

experimented with technologies

began to understand effects locally

developed the Quality Improvement Paradigm

informal applied for cross project learning

evolved the Goal/Question/Metric Paradigm

recognized types of goals and questions

began formalizing process, product, knowledge
and quality models

# How have the activities evolved?

Phase 3

Working on **packaging experiences** for reuse

choosing potentially reusable experiences

recognizing what is appropriate for SEL

studying notations for defining experiences

a project characterization model

defining models of tailorable experiences

a tailorable test method

product reuse models/architectures

defining process models for reusing experience

defining a reuse oriented evolution model

working on integrating experience models

developed the Experience Factory concept and
integrated it with an evolved QIP and GQM

# Evolution of Measurement/Studies in the SEL

## Packaging

SEL Ada Process
SEL Cleanroom Process
SME
Managers Handbook
Experience Factory

## Improving

Methodology Evaluation
Cost Model Analysis
Test Technique Analysis
QIP

Ada
OOD
Cleanroom
CASE

## Understanding

Modeling environment
Data Collection (GQM)
Resource Baselines
Defect Baselines

Design Measures
Cost vs. Size Complexity

Test Method
Reuse

# Overview of the Current Framework

## Quality Improvement Paradigm

an evolutionary improvement paradigm, based upon the scientific method, tailored for the software engineering

## Goal/Question/Metric Paradigm

an approach for establishing project, corporate, and research goals and a mechanism for measuring against those goals

## Experience Factory

an organization that supports research and development by studying projects, developing and refining models, and supplying them to projects for further analysis and refinement

# Quality Improvement Paradigm

**Characterize** the current project and its environment with respect to a variety of models.

**Set** the quantifiable **goals** for successful project performance and improvement.

**Choose** the appropriate **process** model and supporting methods and tools for this project.

**Execute** the **processes**, construct the products, collect and validate the prescribed data ,and analyze it to provide real-time feedback for corrective action.

**Analyze** the **data** to evaluate the current practices, determine problems, record findings, and make recommendations for future project improvements.

**Package** the **experience** in the form of updated and refined models and other forms of structured knowledge gained from this and prior projects and save it in an experience base so it represents our current state of knowledge and is available for future projects.

# The Goal Question Metric Paradigm

A mechanism for defining and interpreting operational and measurable software goals

It combines models of

an **object of study**, e.g., a process, product, or any other experience model and

one or more **focuses**, e.g., models aimed at viewing the object of study for particular characteristics

that can be analyzed from a **point of view**, e.g., the perspective of the person needing the information, which orients the type of focus and when the interpretation/information is made available

for any **purpose**, e.g., characterization, evaluation, prediction, motivation, improvement, which specifies the type of analysis necessary

to generate a **GQM model**

relative to a **particular environment**

# Classes of Studies
# Scopes of Evaluation

| | | #Projects | |
|---|---|---|---|
| | | *One* | *More than one* |
| **# of Teams per Project** | *One* | Single Project (Case Study) | Multi-Project Variation |
| | *More than one* | Replicated Project | Blocked Subject-Project |

# Classes of Studies Examples

| Single Project (Case Study) | Multi-Project Variation |
|---|---|
| Independent V&V | Effect of Methodology |
| Cleanroom Process | Resource Model Studies |

Defect Analysis Studies

Ada/Object Oriented Design

Code Reuse in Ada/Fortran

| Replicated Project | Blocked Subject-Project |
|---|---|
| Effect of Methodology | Reading vs. Testing |
| Cleanroom Process | |
| Ada/O-O Design | |

# Cleanroom Process

Key components:

- Mathematically-based design methodology
    - Function specification for programs
    - State machine specification for modules
    - Reading by stepwise abstraction
    - Correctness demonstrations when needed
- Top-down development
- Implementation without any on-line testing by developer
- Independent testing
    - Statistically based on anticipated operational use
    - Quality assurance orientation

# Replicated Cleanroom Study

Study Goal:

Analyze the Cleanroom process to evaluate it with respect to the effects on the process, product and developers relative to differences from a non-Cleanroom process

Environment:

University of Maryland

Electronic message system (~ 1500 LOC)

15 three-person teams (10 used Cleanroom)

Empirical study:

3 to 5 test submissions

Data collected

Background

Attitude survey

On-line activity

Testing results

# Replicated Cleanroom Study

## EFFECT ON PROCESS

Cleanroom developers felt they more effectively applied off-line review techniques, while others focused on functional testing

Cleanroom developers spent less time on-line and used fewer computer resources

Cleanroom developers tended to make all their scheduled deliveries

## EFFECT ON PRODUCT

Static properties:
- Less dense complexity
- Higher percentage of assignment statements
- More global data
- More comments

Operational properties:
- Product more completely met requirements
- Higher percentage of test cases succeeded

# DEFINING AN SEL CLEANROOM PROCESS MODEL

Existing models: standard SEL model,
IBM/FSD Cleanroom Model
experimental UoM Cleanroom model

Goals: characterize and evaluate in general,
and with respect to changing requirements

IBM/Cleanroom model lessons learned:
basic process model, methods and techniques
process very effective in given environment
......

UoM/Cleanroom model lessons learned:
no testing enforces better reading
process quite effective for small project
formal methods hard to apply, require skill
may have insufficient data to measure
reliability

# DEFINING AN SEL/CLEANROOM PROCESS MODEL (Cont.)

Define SEL/Cleanroom process model:

- Use informal state machine and functions
- Training consistent with UoM course on process model, methods, and techniques
- Emphasize reading by two reviewers
- Allow back-out options for unit testing certain modules . . .
- When no new information, use standard SEL activities

Monitor and make changes to the process model in real time

Write lessons learned for incorporation into next version

Redefine process for the next execution of the process model

# SOME LESSONS LEARNED USING CLEANROOM in the SEL

Can scale up to 30KLOC

Can use with changing requirements

Failure rate during test reduced to close to 50%

Reduction in rework effort
    95% as opposed to 58% took < 1 hour to fix

Only 26% of faults found by both readers

Productivity increased by about 30%

Effort distribution changes:
    more time in design
    50% of code time spent reading

Code appears in library
    later than normal
    more like a step function

Less computer use by a factor of 5

# SOME LESSONS LEARNED USING CLEANROOM in the SEL (Cont.)

Better training needed for methods and techniques

Better mechanisms needed for transferring code to testers

Testers need to add requirements for output analysis of code

No payoff in reliability modeling

**Side effects:**

Caused more emphasis on requirements analysis

**Define next experiments:**

Apply formal models more effectively - use box structure approach

Change application domain and keep size the same

Scale up to a 100KLOC project

# Packaging the Experience

Packaging requires the
continual accumulation of evaluated experiences (**learning**)
in a form that can be effectively understood and modified (**experience models**)
into a repository of integrated experience models (**experience base**)
that can be accessed and modified to meet the needs of the current project (**reuse**)

Systematic learning requires support for
recording experience
off-line generalizing and tailoring of experience
formalizing experience

Packaging useful experience requires
a variety of models and formal notations that are tailorable, extendible, understandable, flexible and accessible

An effective experience base must contain
accessible and integrated set of analyzed, synthesized, and packaged experience models that captures the local experiences

# The Experience Factory

Logical and/or physical organization (separate from the project organization) that supports project developments by

analyzing and synthesizing all kinds of experience models

acting as a repository for such experience

supplying that experience to various projects on demand

It packages experience by building

informal, formal or schematized, and productized models and measures

of various software processes, products, and other forms of knowledge

via people, documents, and automated support

# What kinds of experience can we package?

Resource Baselines and Models

Change and Defect Baselines and Models

Product Baselines and Models

Process Definitions and Models

Method and Technique Models and Evaluations

Products

Lessons Learned

Quality Models

*In the SEL, they exist in the form of standards, policies, tools*

# Forms of Packaged Experience

Equations defining the relationship between variables,

e.g. $\text{Effort} = a * \text{Size}^{b}$

Histograms or pie charts of raw or analyzed data

e.g. % of each class of fault

Graphs defining ranges of “normal”

e.g. graphs of size growth over time with confidence levels

Specific lessons learned

associated with project types, phases, activities

e.g. reading by stepwise abstraction is most effective for finding interface faults

in the form of risks or recommendations

e.g. definition of a unit for unit test in Ada needs to be carefully defined

models or algorithms specifying the processes, methods, or techniques

e.g. an SADT diagram defining Design Inspections with the reading technique a variable dependent upon the focus and reader perspective

# PACKAGING EXPERIENCE:

## RESOURCE MODELS

In the SEL,

Example packaged relationships include:

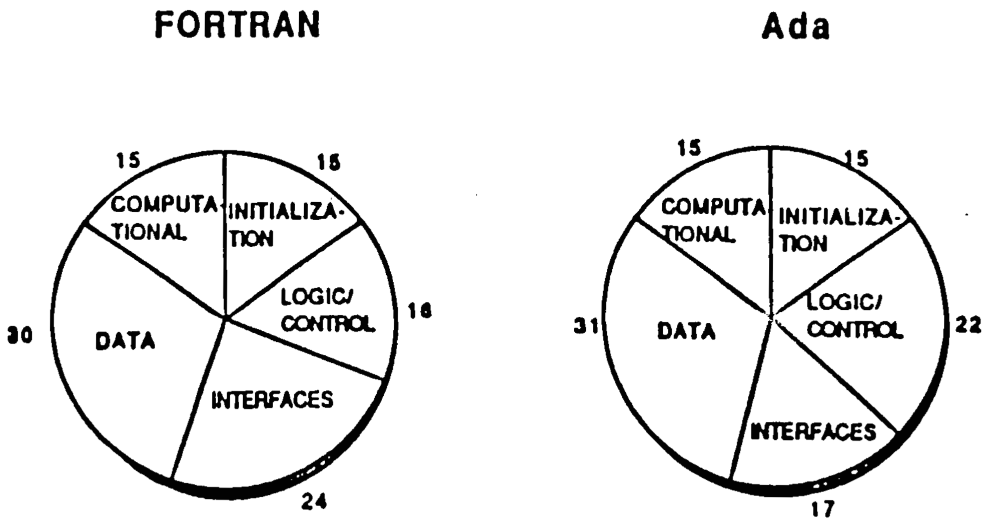
Effort = 4.37 + 1.43devlines
Effort = 5.5 + 1.5newlines
Docpages = 99.1 + 30.9 devlines
Numruns = -108 + 151devlines

Projects under 50kloc:
Effort = .877 + 1.5newlines
Projects over 50kloc
Effort = 66.9 + .003 numruns

Factors that affect cost and quality are:

+methodology (favorable impact)
-cumulative complexity (unfavorable impact)

# CLASSES OF ERROR*



- •ERROR PROFILES QUITE SIMILAR; EVEN FOR DIFFERENT LANGUAGES
- •Ada SOMEWHAT FEWER INTERFACE ERRORS

*BASED ON ERROR FROM 5 Ada AND 8 FORTRAN PROJECTS

# Research Laboratory Needs

We need a set of SELs or Experience Factories
each focused on packaging local experiences by
building and tailoring local models
integrating technologies
studying scale-up
building an experience bases
developing automated aids

*How big should an SEL be?*
*Should it only be domain specific?*

and
the integration of these local experience factories
into a high level Experience Factory that
abstract from local experiences
looks for patterns across environments
generates the basic models of the science

*How is this accomplished?*

# Conclusions

We can take advantage of the experimental nature of software engineering

Process, product, environment can be measured and can be used to support practical development and research

Integration of the
Improvement Paradigm
Goal/Question/Metric Paradigm
Experience Factory Organization
provides a framework for both development and research

Based upon our experience, it helps us
derive **descriptive models** of our experiences
understand our experiences and our problems
evaluate and learn from our experiences
build effective **prescriptive models** of our experiences and our quality objectives

Should be applied today and evolve with technology

You don't have to make the same mistakes we did, you can learn from our experiences

N92
19422
UNCLAS

N92-19422

# IMPACT OF A PROCESS IMPROVEMENT PROGRAM IN A PRODUCTION SOFTWARE ENVIRONMENT: ARE WE ANY BETTER?

Gerard H. Heller
Gerald T. Page

COMPUTER SCIENCES CORPORATION
GreenTec II
10110 Aerospace Road
Lanham-Seabrook, MD 20706
(301) 794-4460

## ABSTRACT

For the past 15 years, Computer Sciences Corporation (CSC) has participated in a process improvement program as a member of the Software Engineering Laboratory (SEL), which is sponsored by the National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center (GSFC). This paper analyzes the benefits CSC has derived from involvement in this program. In the environment studied, it shows that improvements were indeed achieved, as evidenced by a decrease in error rates and costs over a period in which both the size and the complexity of the developed systems increased substantially. The paper also discusses the principles and mechanics of the process improvement program, the lessons CSC has learned, and how CSC has capitalized on these lessons.

## INTRODUCTION

Computer Sciences Corporation (CSC) had some compelling motivations to join with the National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center (GSFC) and the University of Maryland 15 years ago to form the Software Engineering Laboratory (SEL). In the context of 1976 and our partnership with GSFC, we wanted to study our overall flight dynamics software development process closely enough to be able to refine and improve it. Even then, we knew we had to be able to accurately describe and measure that process before real improvements could be made. Slowly and steadily, we embarked on a conscious process improvement program that would help us produce the larger and more complex flight dynamics ground systems required to support the more sophisticated spacecraft being built.

We wanted to build these complex systems with more reliability and greater economy. Our personnel were already committed tc building quality systems; what we needed now was to build quality systems more productively. We also needed to expand the skills of our current personnel and to attract and retain new personnel who would enjoy the twin challenges of doing flight dynamics worl. and simultaneously trying to improve the methods used to do that work.

As competition to provide flight dynamics services increased both here and abroad, CSC became more ambitious in efforts to improve its processes and products and more committed to allocating the resources needed to make these improvements. We wanted to validate our belief that higher quality at lower costs was not a contradiction. We wanted to show that, in fact, those traits go hand in hand and that high-quality software really does cost less.

Sound business practices showed a need to move forward, not only to improve on our current work but to seek new opportunities as well. One way to enter these new business areas was to objectively demonstrate superior products and performance in our work with GSFC. Another way was to pursue and achieve formal recognition by other members of our industry. Our motivations for the SEL partnership were clear and compelling. From our participation in the SEL, we expected to capture specific gains, to learn some vital lessons, and to demonstrate, over time, that we were truly "getting better" at doing flight dynamics work.

Have we achieved these goals after 15 years of participation in the SEL? The rest of this paper answers this question. It describes the principles and mechanics of the SEL process improvement program, including examples of the program in action: examines what we have learned from our role in the program and how we have capitalized on that learning; and analyzes trends over the past 15 years to determine quantitatively whether or not we have met our objectives.

## SEL BACKGROUND

### The SEL

The SEL (Reference 1) is a research project sponsored by NASA/GSFC and supported by the Computer Science Department at the University of Maryland and by CSC. The SEL's mission is to understand and improve the overall software development process. To do this, the SEL conducts experiments with production software projects, measures the effect of the techniques applied, and then adopts the most beneficial methodologies for future projects.

### The SEL Environment

The production software environment studied by the SEL is an environment of similar flight dynamics applications developed by GSFC for such spacecraft problems as attitude and orbit determination and control, mission planning, and maneuver control. These applications are largely scientific and mathematical, with moderate reliability requirements and severe development time constraints imposed by a fixed spacecraft launch date. Table 1 summarizes the current characteristics of this environment.

### The SEL Process Improvement Program

The SEL process improvement program is a conscious, continuous effort to build higher quality systems at lower costs by understanding the environment, measuring and evaluating the results of planned process changes, and capturing and packaging experience to optimize the process and to anticipate uncontrollable changes.

Table 1. Characteristics of the Development Environment Studied by the SEL

| Characteristics | Current State |
|---|---|
| Organization size | >250 people |
| Computing environment | HDS 8063 (IBM 3083)<br>VAX 8820, 11/780 |
| Languages | FORTRAN, Ada |
| Applications | Primarily attitude; some orbit and mission analysis |
| Average system size | 180 KSLOC |
| Average project duration | 2 years |
| Average staff level | 15 to 20 people |
| Staff background | Computer Science, Mathematics, Physics |

For a process improvement program to succeed, it must

- *Be a conscious effort.* Improvements will not happen by themselves: resources must be allocated to make them happen.

- *Be a continuous effort.* Even very mature processes need to be refined in the face of changing environments and advances in technology.

- *Be built on a solid understanding of the environment.* This includes characterizing the products produced and processes used.

- *Achieve understanding by measurement and evaluation.* The parameters of the environment must be quantified to evaluate the effectiveness of changes made to it.

- *Feed back lessons learned.* The results of measurement and evaluation must be fed back into the process to optimize it.

- *Package lessons learned.* Experiences must be packaged so that managers can apply them to their day-to-day challenges and can anticipate changes outside of their control, thus preserving corporate legacy when experienced people leave.

## EVOLVING TO AN OPTIMIZING ENVIRONMENT

Given the principles of the SEL process improvement program, we can now look at that program in action over the SEL's first 15 years. For convenience, SEL activities are grouped into three broad classes: evaluating changes to life-cycle processes, evaluating changes to technology and methodology, and providing support to the development organization.

### Changing Life-Cycle Processes

A first goal of the SEL was to establish a measurement program to capture and quantify the characteristics of the environment, including all its processes and products. The SEL spent much of the first 5 years simply learning how to collect, analyze, and interpret data. This early analysis showed that testing was one of the weakest activities in the flight dynamics development process, and it set the stage for several early experiments in changing a life-cycle process.

The goal in changing a life-cycle process is to identify a particular life-cycle phase or activity as a candidate for improvement, vary just that one element of the process, and then measure the impact on the process and product. If the analysis shows that the change favorably affects quality and/or productivity, it is incorporated into the process. In essence, this type of change can be viewed as "fine-tuning" an existing process.

In 1981, in a step to understand the weaknesses perceived in testing, the SEL evaluated the impact of independent verification and validation (IV&V) in the flight dynamics environment (Reference 2). It applied IV&V techniques on four flight dynamics projects, defined metrics for analyzing the change, and compared these metrics with those of earlier projects that did not use IV&V. The results showed little or no significant improvement in quality and reliability and, at the same time, reflected a substantial increase in development cost. The study concluded that IV&V was not cost effective for use in the SEL flight dynamics environment.

In 1984, continuing its quest to improve testing, the SEL compared three different software verification techniques (Reference 3). It trained a group of professional programmers in structural testing, functional testing, and the peer review technique of code reading, and then gave them programs that had been seeded with errors on which to apply these techniques. After the experimenters calculated such metrics as the number of errors found and the average effort expended to find each error, they concluded that code reading was the most cost-effective technique for uncovering errors in software units. As a result, code reading was incorporated as a formal activity into the flight dynamics software development process.

By participating in these life-cycle process change experiments, CSC has learned several lessons:

- To effectively evaluate and implement life-cycle changes, resources must be allocated; that is, an independent organization like the SEL must be designated, to focus on measuring and evaluating impacts. The job is too big for managers to do in their "spare time." We have carried this lesson beyond the SEL environment by establishing software engineering process groups to perform this type of analysis across the entire Systems, Engineering, and Analysis Support (SEAS) contract (Reference 4) currently being performed for GSFC.

- Peer review techniques are a cost-effective method for isolating errors early in the development life cycle. We have made such techniques a fundamental part of our SEAS System Development Methodology (Reference 5).

## Changing Technology/Methodology

After about the first 5 years of studying the flight dynamics environment and its development process and experimenting with life-cycle process changes, the SEL looked back on its experiences and drew some basic conclusions. One was that following a formal methodology, provided that it is not "labor intensive," can produce a 10- to 15-percent improvement in a software development program compared to not following a formal methodology or following an ad hoc approach (Reference 1). Although adding and subtracting new techniques in the form of life-cycle changes can fine-tune the methodology, it does not produce substantial overall improvements to the program. To achieve substantial changes requires a major overhaul of the formal methodology itself or the insertion of new technology.

The SEL approach to methodology and technology changes is different from the relatively simple experimentation performed for life-cycle changes. Rather than performing a single experiment, evaluating the results, and deciding to implement a new technique across the entire program, the SEL knew that introducing an entire methodology or technology would require a more cautious approach because of risks associated with the immaturity of the methodology or technology and the extensive retraining of staff required. The SEL approach is to experiment with the new methodology or technology via a pilot project or projects, evaluate the metrics collected, hypothesize about the potential benefits, and then repeat the experiment several times to confirm or deny initial hypotheses and to establish trends.

In 1984, the SEL began evaluating a methodology based on the Ada language and object-oriented design. This was a radical change from the top-down structured design techniques and the FORTRAN mindset then in place in the flight dynamics environment. To evaluate the new methodology, the SEL began an experiment in which the same flight dynamics simulator was built in two parallel development efforts: one in FORTRAN and the other in Ada. Known as the GRODY experiment, its results have been documented in a number of papers and reports in the SEL series (Reference 6). Since this first study, five more simulators have been built in Ada, and a separate study was performed to transport one of the simulators from a VAX environment to an IBM mainframe environment (Reference 7). Although the trends on these Ada projects are still being analyzed, a significant increase in reuse, with substantial development cost savings, seems to be the greatest benefit.

Another methodology change with which the SEL has begun to experiment recently is the cleanroom development methodology (Reference 8). This methodology relies on human discipline and peer review techniques to eliminate errors early in the life cycle. It isolates the designers and coders from the testers and prohibits the coders from even compiling their programs. Although the SEL had done some early evaluations of this methodology (the code-reading technique already discussed was adopted from the cleanroom methodology), it did not begin a cleanroom pilot project until 1988. The ACME project used the cleanroom approach to develop one of the subsystems for an attitude ground support system (AGSS). Initial ACME data showed an improvement in error rates (Reference 9). Currently, two other projects are using the cleanroom methodology to confirm and expand upon the initial trends observed on ACME. One effort is trying to reproduce the trends on another project of ACME's scale (approximately 30 KSLOC in size), and the other is trying to scale up and use the methodology on an entire AGSS (more than 150 KSLOC in size) to see if similar trends appear.

By participating in SEL methodology change experiments, CSC has learned other lessons:

- We have been able to minimize the risks of inserting new technology into the flight dynamics environment by measuring and evaluating impacts in a controlled fashion, allowing educated decisions to be made based on quantitative cost/benefit tradeoffs.

- In the case of Ada, we have been able to take advantage of the lessons learned on the pilot projects by communicating them to other organizations within our company through various technology exchange forums.

### Supporting the Organization

The third category of activities in the SEL process improvement program is aimed at supporting the needs of the development organization rather than making controlled changes to the process or environment. This involves the concepts of effectively capturing and packaging experience.

Early in its history, the SEL defined and documented the methodology being used to develop flight dynamics projects. It published a series of documents that established standards and guidelines for both developers and managers in such areas as design, implementation, and testing techniques; life-cycle reviews and documentation;

planning, monitoring, and controlling projects; cost estimation; and product assurance (References 10–15). These documents helped capture experience in the flight dynamics environment and were instrumental in quickly training new staff. As technology changed and the SEL's domain grew, it became evident that these documents had to evolve as well. Thus, the SEL is currently updating this series with the dual objectives of (1) augmenting the methodology to broaden its scope and include new technology and (2) generalizing it where possible to provide greater flexibility for making future changes.

In a related activity, the SEL developed process models for the environment. A process model defines the expected behavior of a particular measure, such as staff resources expended, over the life cycle of a project. Process models capture the experience learned on past projects and package it in a form that can be used on current projects. Models give greater visibility into managing development projects. They allow managers to make at-completion predictions of such measures as resource utilization, error rates, and project schedules. They can also be used to determine when a project is deviating from the typical behavior of past projects and help to determine the causes of such deviations.

The SEL developed a tool that its managers use to take advantage of the SEL process models. This tool, the Software Management Environment (SME) (Reference 16), allows managers to use process models that are based on a pool of projects similar to the ones they are currently managing. It helps them analyze progress on their projects, predict outcomes, and plan alternatives, all with the advantage of using the experience base built up by the SEL in the flight dynamics environment.

The SEL process improvement program has also helped recognize and respond to the changing needs of the staff members in the environment. Over the 15 years since the SEL started, the primary background of the developers in the environment has shifted from mathematics and physics to computer science. In response to this, the SEL initiated a training program to give new developers a basic foundation in flight dynamics applications and quickly familiarize them with the SEL methodology.

By participating in SEL activities to support the organization, CSC has learned even more:

- We need to have a documented methodology used consistently across the environment. Drawing on the SEL's experience in documenting the methodology used in the flight dynamics environment and on our own, more general corporate methodology, we have documented a system development methodology for use across the entire SEAS contract (Reference 5), and we have supplemented this methodology with a set of standards and procedures (Reference 17) to help staff members apply it.

- We know that quantitative management works. Measuring process and product allows us to develop quantitative models that enable projects to be better planned, more accurately estimated, and more effectively controlled. We can also detect deviations from our plans more easily, and hence we can correct problems earlier. Recognizing the importance of quantitative management, we have packaged our experiences in this area in a data collection, analysis, and reporting handbook to be used by our managers on the SEAS contract (Reference 18).

- We need to train our organization in the methodology and in process improvement concepts. We have developed a required training program (Reference 19) for all engineers, developers, testers, integrators, and managers to ensure consistent understanding and application of the SEAS System Development Methodology and of quantitative management techniques across the entire contract.

- We can write better proposals when estimates are backed up with solid data. From a business point of view, being able to point to a quantitative experience base lends credibility to proposals and brings in more work.

## ASSESSMENT OF PROGRESS

We have seen some mechanics of the SEL process improvement program, some specific examples of the types of activities in which the SEL engages, and how CSC has benefited from participating in these activities. Using the SEL's own data, we now address the question "Are we any better?" by examining some growth, reliability, and productivity trends over the past 15 years.

Three areas measure changes in the nature of flight dynamics systems: complexity, general requirements, and system size. CSC performed a study iı 1988 to examine trends in these areas, as well as in software reuse (Reference 20). At that time, it was generally felt that systems were becoming more complex, primarily as a reflection of the increased complexity of the spacecraft they were developed to support. Table 2, adapted from the study, shows a comparison of typical spacecraft configurations in the mid-1970s with those of the late 1980s. This table shows that the required attitude accuracy is 50 times greater than it was, data rates are over 14 times faster, and there are 3 times as many telemetry data types and 10 times as many sensors. In the above-mentioned study, these and other characteristics were combined into a synthetic measure of spacecraft complexity. A plot showing the overall trend in this complexity measure (Figure 1) shows that it has more than doubled over the past 15 years.

Table 2. Comparison of Spacecraft Characteristics

| Characteristics | Mid-1970s | Late 1980s |
|---|---|---|
| Control | Spin stabilized | 3-axis stabilized |
| Sensors | 1 | 8 to 11 |
| Torquers | 1 | 2 to 3 |
| Onboard computer | Analog, simple control | Digital, autonomous |
| Telemetry types | 5 | 12 to 15 |
| Data rates | 2.2 kb/s | 32 kb/s |
| Accuracy | 1 degree | 0.02 degree |

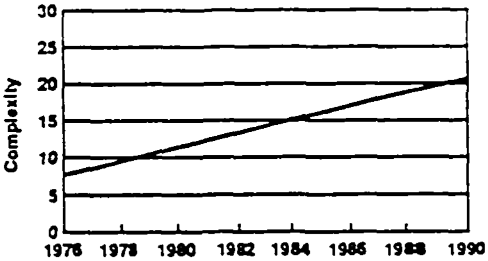


Figure 1. Trends in Spacecraft Complexity

The same study also derived a measure of functional specification complexity to reflect the growth in general requirements. This measure also more than doubled over the past 15 years, yet requirements growth was not directly proportional to spacecraft complexity. For example, going from a spacecraft with one sensor to a spacecraft with five sensors means that software must be developed to process data from all five sensors. Beyond that, however, it may also mean an additional requirement to create a utility that determines the best time to use one sensor instead of another or to create a program that predicts periods when the motor that runs one sensor might interfere with the operation of another sensor. In addition, requirements have been added to build programs that perform such functions as predicting Earth occultation of a given set of stars, predicting Moon interference with sensor operation, or predicting antenna contact times. Thus, both increased spacecraft complexity and general requirements growth can be seen as separate drivers in the growth of system size.

In terms of system size, Figure 2 shows that total number of delivered lines of code (including blank lines and comments) has not quite tripled. At the same time, development error rates have been reduced by 65 percent (Figure 3). Figure 4 shows the trend in the cost per developed line of code. It has remained relatively constant, although the narrowing of the maximum and minimum range lines indicates that it is becoming more predictable.

Looking at all of these trends together now helps us answer our original question. Although a rigorous study of the relationship between spacecraft complexity, requirements growth, and system size has not been performed, one could expect that a doubling in both complexity and general requirements might result in a quadrupling of system size. Since system size did not quite triple, we conclude that developers are now packaging more functionality per line of code than they were 15 years ago. Thus, the SEL process improvement program has enabled us to build systems that provide more functionality per line of code, with significantly fewer errors per line of code, at a lower cost per line of code than systems of 15 years ago. It is clearly possible to improve productivity and lower error rates at the same time.

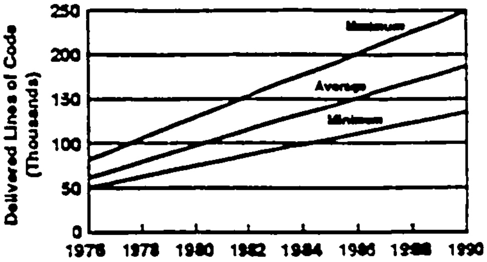


Figure 2. Trends in Size Growth of Flight Dynamics Applications

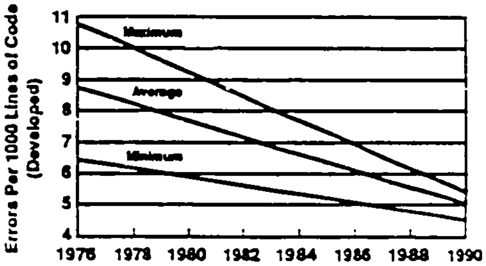


Figure 3. Trends in Development Error Rates

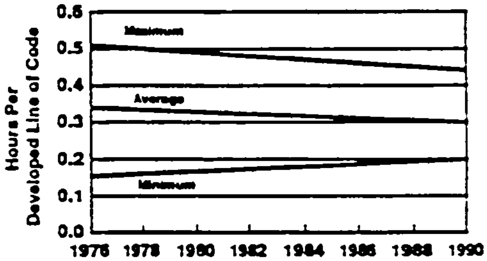


Figure 4. Trends in Software Cost

In addition, process models derived from SEL-collected data have helped us predict error rates and system costs more accurately. Thus, the answer to the question "Are we any better?" has to be an unqualified "Yes."

## ACKNOWLEDGMENT

We would like to thank Michele Bissonette for her help in preparing this paper.

## REFERENCES

1. NASA/GSFC Software Engineering Laboratory, SEL-81-104, *The Software Engineering Laboratory*, D. Card, F. McGarry, et al., February 1982

2. —, SEL-81-101, *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*, G. Page, F. McGarry, and D. Card, June 1985

3. —, SEL-85-001, *A Comparison of Software Verification Techniques*, D. Card, R. Selby, et al., April 1985

4. NASA/GSFC, Request for Proposal (RFP) 5-74300/184, Systems, Engineering, and Analysis Support (SEAS), September 1986

5. Computer Sciences Corporation, *SEAS System Development Methodology (SSDM)*, July 1989

6. Software Engineering Laboratory, SEL-90-004, *Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary*, T. McDermott et al., September 1990

7. —, SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL)*, L. Jun et al., June 1990

8. IBM Federal Systems Division, *Cleanroom Software Development Method*, M. Dyer, October 1982

9. Software Engineering Laboratory, SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis*, S. Green et al., March 1990

10. —, SEL-81-205, *Recommended Approach to Software Development*, F. McGarry, G. Page, et al., April 1983

11. —, SEL-83-001, *An Approach to Software Cost Estimation*, F. McGarry, G. Page, et al., February 1984

12. —, SEL-84-101, *Manager's Handbook for Software Development*, L. Landis, F. McGarry, et al., November 1990

13. —, SEL-85-005, *Software Verification and Testing*, D. Card, C. Antle, and E. Edwards, December 1985

14. —, SEL-86-001, *Programmer's Handbook for Flight Dynamics Software Development*, R. Wood and E. Edwards, March 1986

15. —, SEL-87-001, *Product Assurance Policies and Procedures for Flight Dynamics Software Development*, S. Perry et al., March 1987

16. —, SEL-89-003, *Software Management Environment (SME) Concepts and Architecture*, W. Decker and J. Valett, August 1989

17. Computer Sciences Corporation, *SEAS System Development Methodology (SSDM) Standards and Procedures*, June 1990

18. —, *SEAS Software Measurement System (SSMS) Handbook*, to be published in early 1991

19. —, *Catalog of SEAS Training Courses*, September 1990

20. —, CSC/TM-89/6031, *A Study on Size and Reuse Trends in Attitude Ground Support Systems (AGSSs) Developed for the Flight Dynamics Division (FDD) (1976–1988)*, D. Boland et al., February 1989

## VIEWGRAPH MATERIALS

## FOR THE

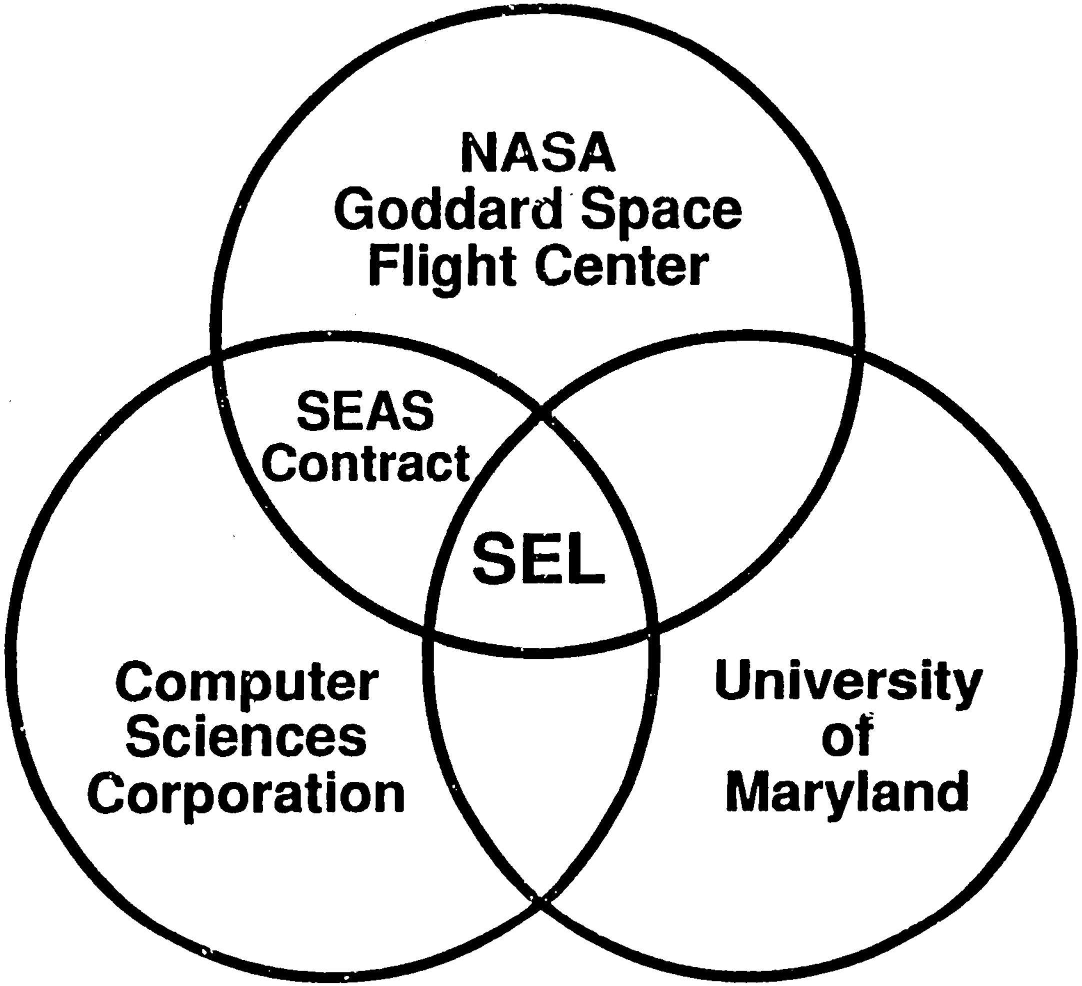
## G. PAGE PRESENTATION

# Impact of a Process Improvement Program in a Production Software Environment: Are we any better?

Jerry Page

November 28, 1990

Computer Sciences Corporation
System Sciences Division



G. Page
CSC
Page 11 of 25

# Software Engineering Laboratory Environment

| Characteristic | Current State |
|---|---|
| Organization Staff Level | > 250 |
| Computing Environment | HSD 8063 (IBM 3083),<br>VAX 8820, VAX-11/780 |
| Languages | FORTRAN, Ada |
| Application | Attitude, orbit, mission analysis |
| Average Svstem Size | 180 KSLOC |
| Average Project Duration | 2 Years |
| Average Staff Level | 15 to 20 |
| Staff Background | Computer Science, Mathematics, and Physical Sciences |

# What Is a Process Improvement Program?

A *conscious, continuous* **effort to build higher quality systems by**

- *Understanding* **the environment**
- *Measuring* **and** *evaluating* **the results from planned changes**
- *Capturing* **and** *packaging experience* **to** *optimize* **the process and to** *anticipate* **uncontrollables**

# Life Cycle Process Changes

- Perform experimental studies on production projects
- Vary one element of process and measure impacts on process and product
- Fine tune process to take advantage of benefits

| Testing Studies | Observations and Actions |
|---|---|
| Code Reading, Functional and Structural Testing | • Code reading most effective technique<br>• Add to process |
| Independent Verification and Validation | • Small effects for relatively large cost<br>• IV&V inappropriate for SEL projects |

# Technology/Methodology Changes

- Test new technology in production environment with pilot project
- Measure impacts on project profiles and products produced
- Package lessons learned, adjust training, and repeat for effect

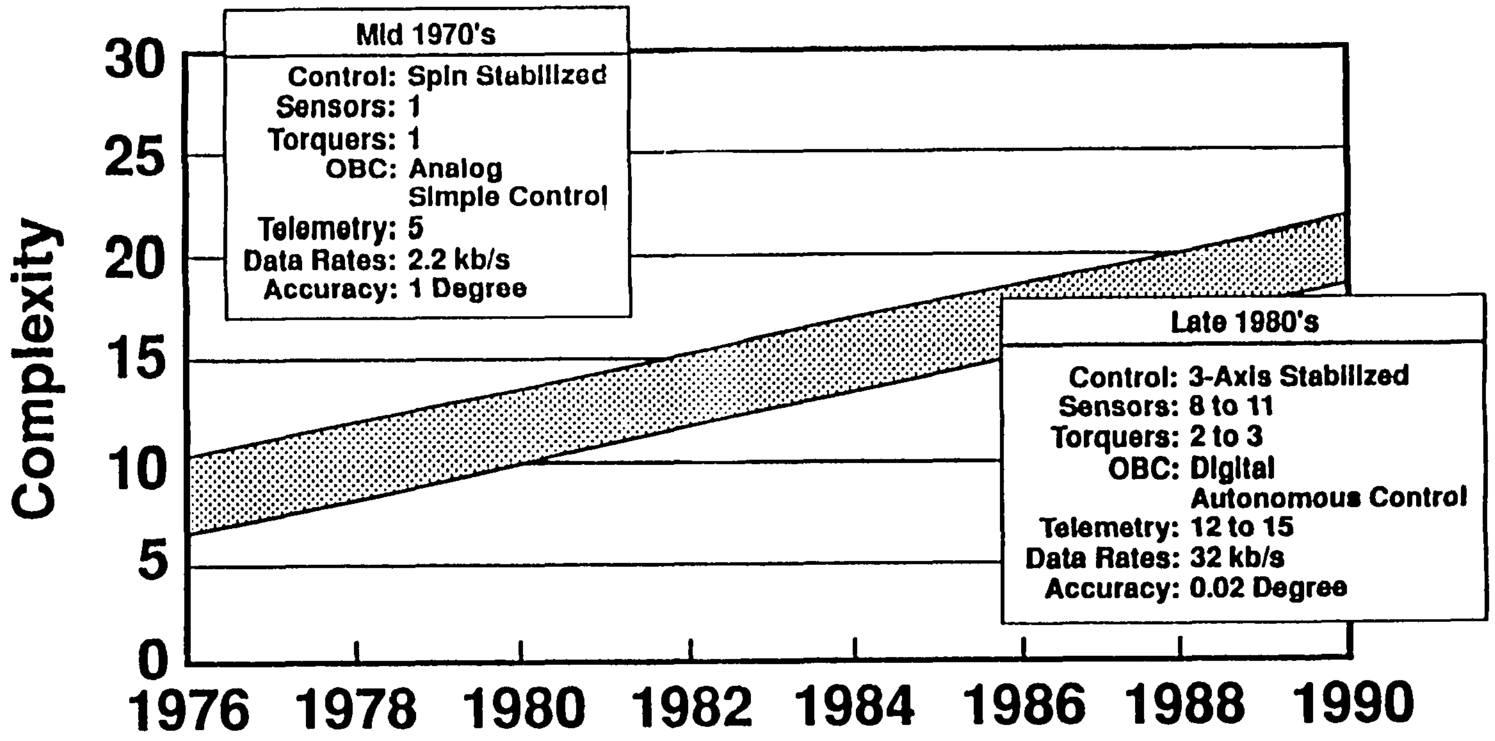
| Technology | Observations and Actions |
|---|---|
| Ada | • Very promising trends on software reuse<br>• Conduct further and more detailed studies |
| Cleanroom | • Initially, error levels very low<br>• Scale up experiment and verify findings |

# Organizational Changes

- Look for deviations from process models
- Determine impacts
- Strengthen definitions of overall approach

| Change | Action Taken |
|---|---|
| Staff Turnover or Staff Growth | • Created and augmented standards and guidelines<br>• Developed Software Management Environment (SME) |
| Staff Background | • Established required training program for developers and managers<br>• Developed Software Development Environment (SDE) |
| Domain Growth | • Augmented methodology to broaden scope<br>• Generalized methodology to make it more flexible |

# System Complexity*



Mid 1970's

Control: Spin Stabilized
Sensors: 1
Torquers: 1
OBC: Analog
Simple Control
Telemetry: 5
Data Rates: 2.2 kb/s
Accuracy: 1 Degree

Late 1980's

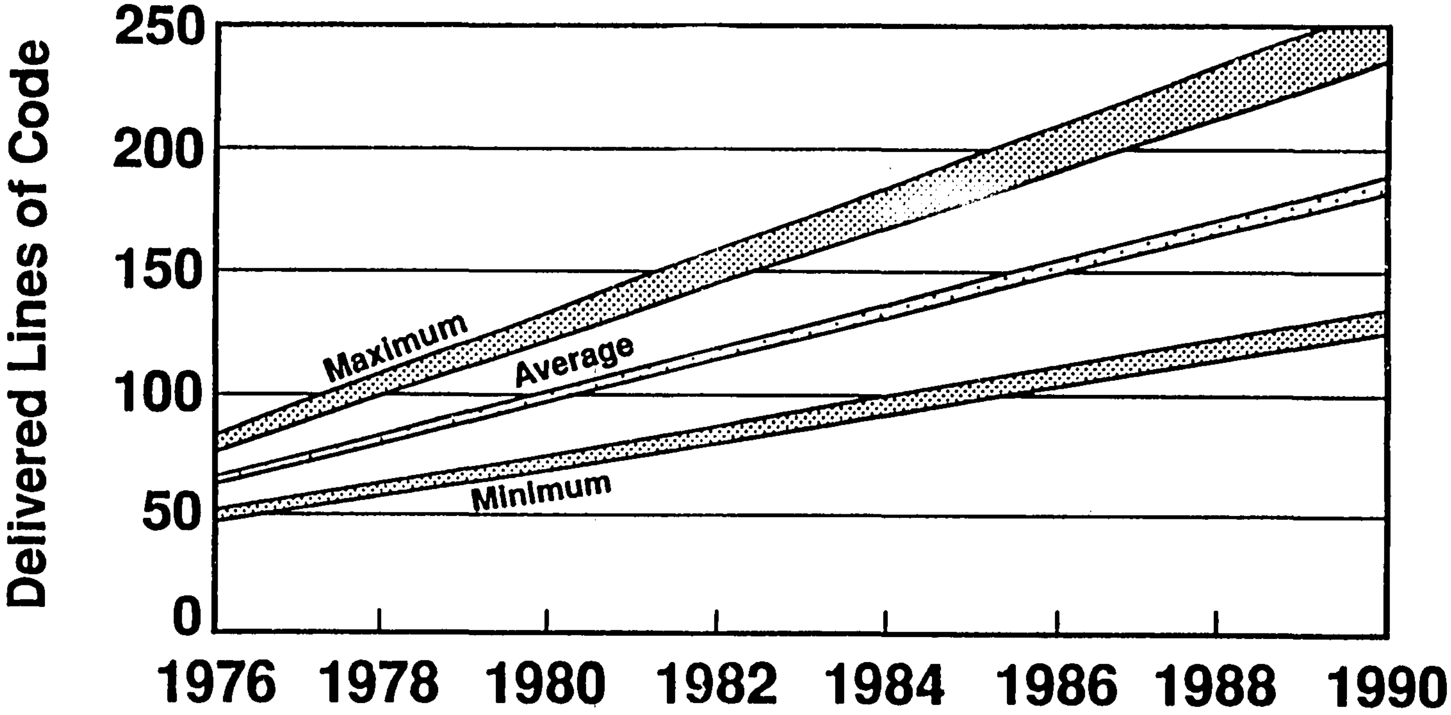
Control: 3-Axis Stabilized
Sensors: 8 to 11
Torquers: 2 to 3
OBC: Digital
Autonomous Control
Telemetry: 12 to 15
Data Rates: 32 kb/s
Accuracy: 0.02 Degree

***Complexity has more than doubled.***

*D. Boland, *A Study on Size and Reuse Trends In Attitude Ground Support Systems (AGSSs) Developed for the Flight Dynamics Division (FDD) (1976–1988)*, CSC/TM-89/6031, CSC, February 1989

Computer Sciences Corporation
System Sciences Division

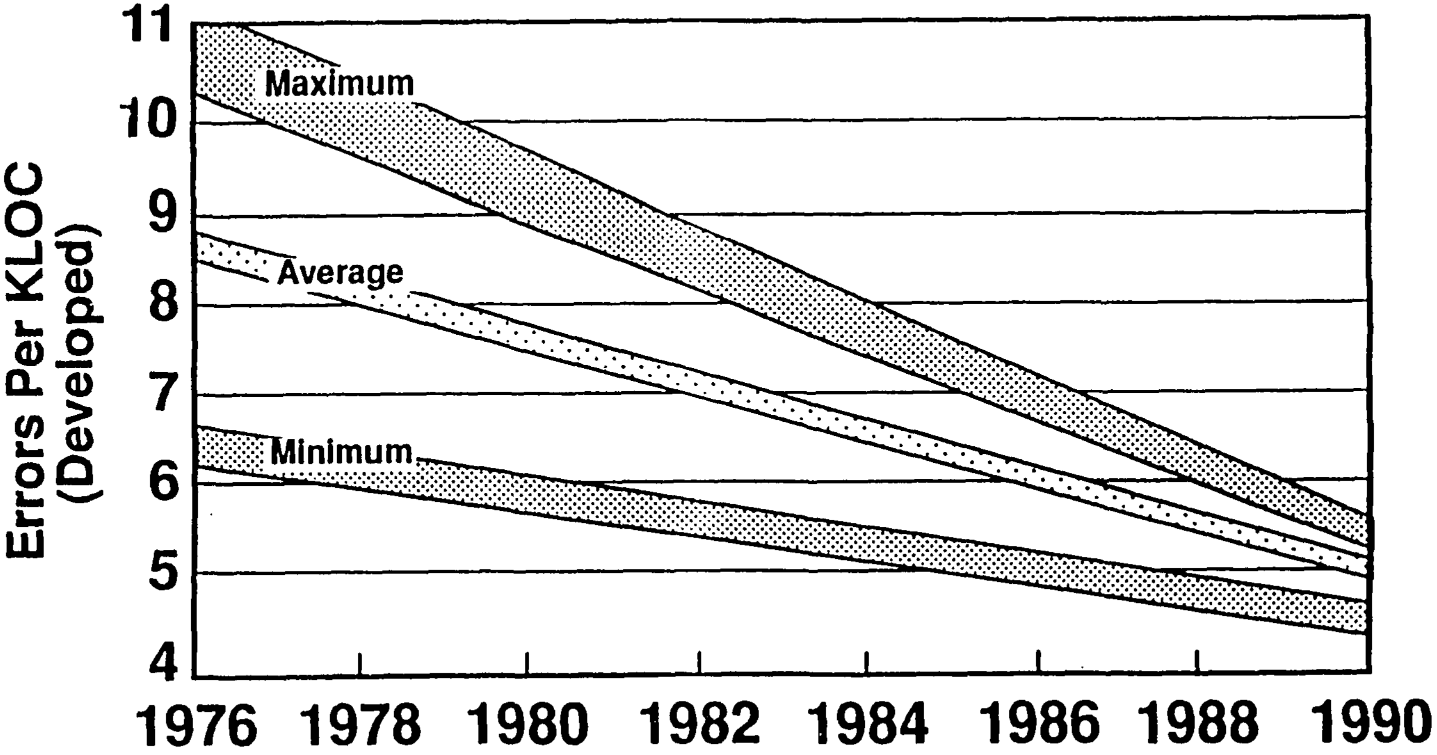
# System Size



***System size has more than doubled.***

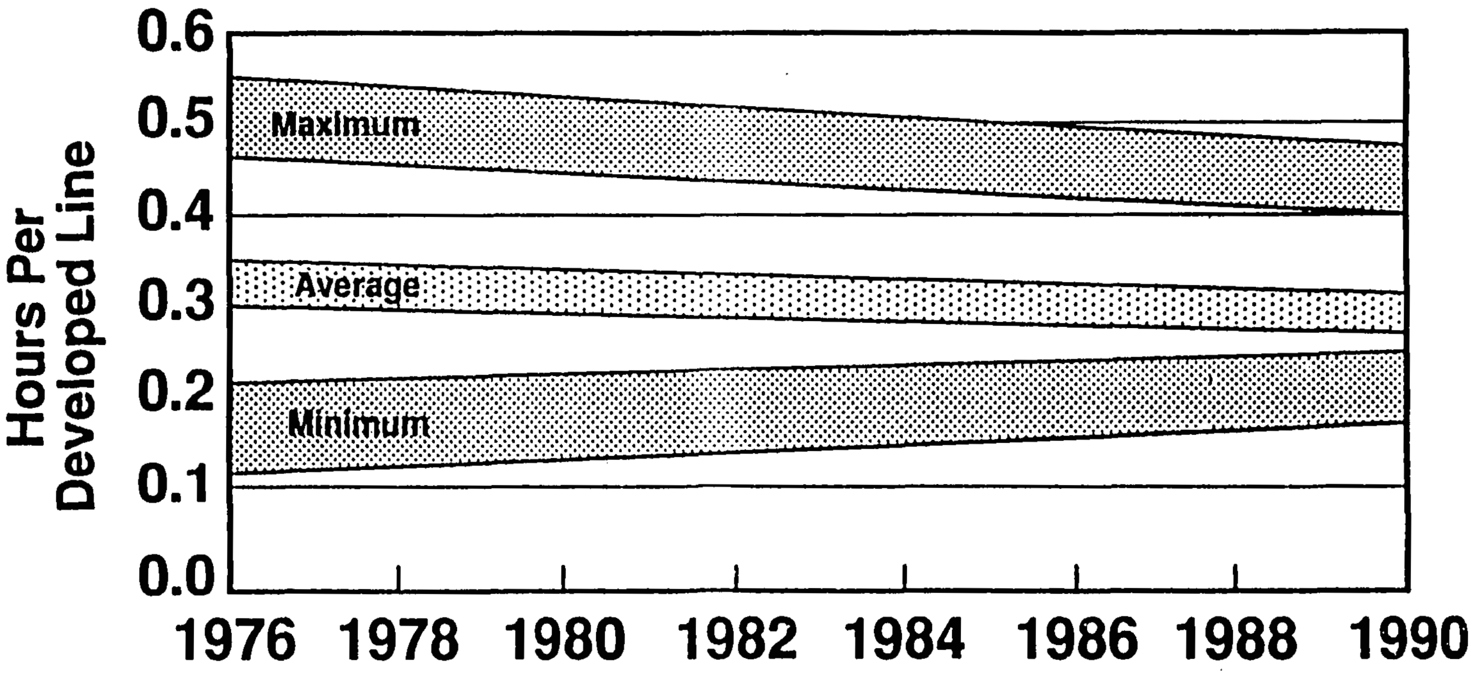
Computer Sciences Corporation
System Sciences Division

# Development Error Rates



***Error raˈes have been reduced by 65 percent.***
***Error models are fairly well established.***

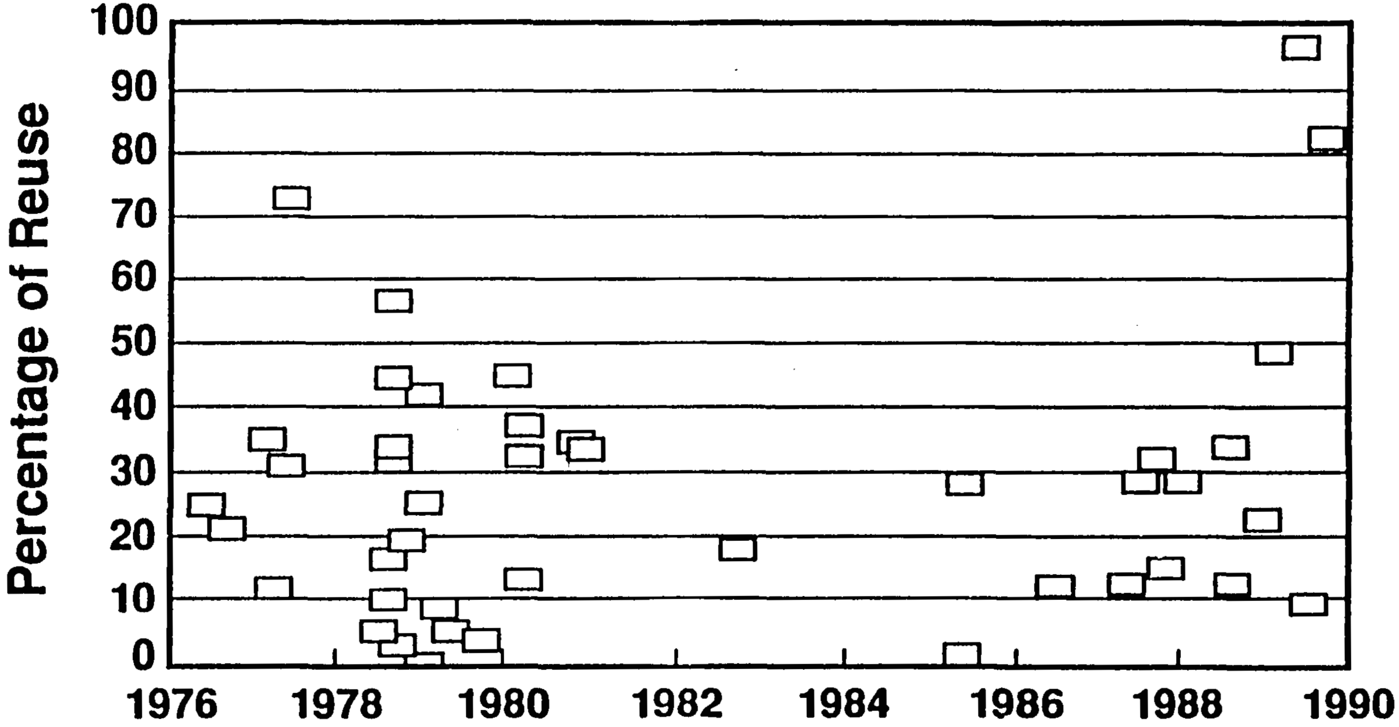
# Cost of Code



***Cost per LOC remained relatively constant.***
***Predictability is improving.***

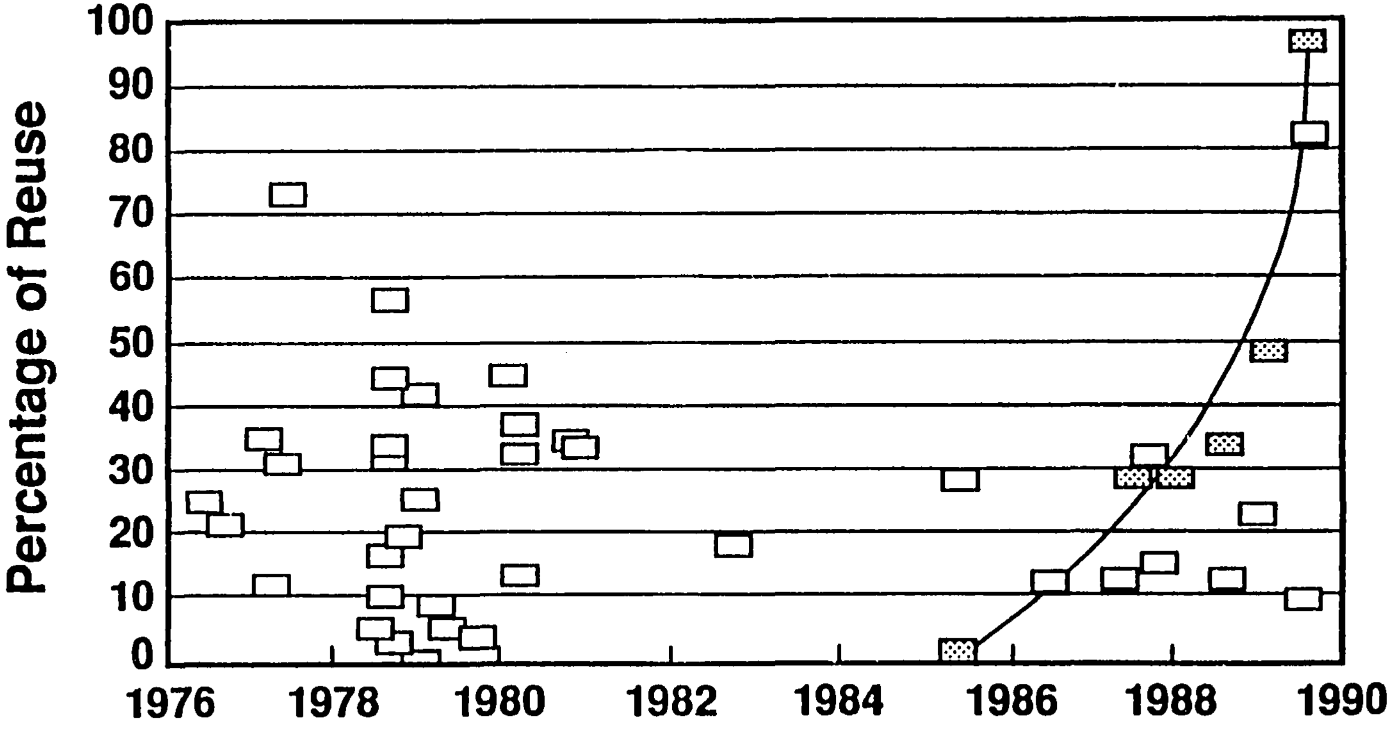
Computer Sciences Corporation
System Sciences Division

# Code Reuse in Systems



***Sometimes interesting things in a picture are lost because of shallow depth of field.***

# Code Reuse in Ada Simulators



***However, searching with a reduced field of view can pay off.***

G. Page
CSC
Page 22 of 25

# What CSC Has Learned

- Quantitative management works
- Peer review works
- You can lower error rates
- You can raise productivity
- You can write more credible proposals when you can back them up with data

# What Has CSC Done to Capitalize on Its Learning?

- Developed a System Development Methodology based on its experience
- Packaged its experience with quantitative management in a manager's Data Collection, Analysis, and Reporting Handbook
- Developed a set of standards and guidelines to complement its methodology
- Developed required training programs for engineers, developers, testers, integrators, and managers to maximize the benefits of its methodology
- Established measurement-based Engineering Process Groups to identify improvement areas, recommend changes, and evaluate the impact of those changes

# Presentation Contributors

Gerry Heller

Tim McDermott

Sharon Waligora

N92
19423
UNCLAS

S1-61

[illegible]

N92-19423

P-33

# TOWARDS UNDERSTANDING SOFTWARE – 15 YEARS IN THE SEL

NC999107

Frank McGarry
Rose Pajerski
GODDARD SPACE FLIGHT CENTER

## ABSTRACT

For 15 years, the Software Engineering Laboratory (SEL) at NASA/Goddard Space Flight Center (GSFC) has been carrying out studies and experiments for the purpose of understanding, assessing, and improving software, and software processes within a production software environment. The SEL comprises three major organizations:

| | |
|---|---|
| • NASA/GSFC | Flight Dynamics Division |
| • University of Maryland | Computer Science Department |
| • Computer Sciences Corporation | Flight Dynamics Technology Group |

These organizations have jointly carried out several hundred software studies, producing hundreds of reports, papers, and documents [Reference 1]—all describing some aspect of the software engineering technology that has undergone analysis in the Flight Dynamics environment. The studies range from small controlled experiments (such as analyzing the effectiveness of code reading versus functional testing) to large, multiple-project studies (such as assessing the impacts of Ada on a production environment). This paper will summarize the key findings that the sponsoring organization (NASA) feels have laid the foundation for ongoing and future software development and research activities.

## I. BACKGROUND

In 1976, NASA/GSFC initiated an effort to carry out experiments within the Flight Dynamics area to attempt to measure the relative merits of the numerous software technologies that were both available and claimed to be significant 'improvements' over currently used practices. Although significant advances were being made in developing new technologies, such as structured development practices, automated tools, quality assurance approaches, and management tools, there was very limited empirical evidence or guidance pertaining to applying these promising, yet immature methods. Primarily to address this situation, the Software Engineering Laboratory (SEL) was formed.

The SEL was formed as a partnership between NASA, the University of Maryland, and Computer Sciences Corporation. This working relationship has been maintained continually since 1976 with relatively little change to the overall goals of the organization during its entire history. In general, the goals have matured; they have not been changed. The goals can be itemized as follows:

1. **Understand**—Improve the insight that exists in characterizing the software process and its products in a production environment,

2. **Assess**—Measure the impact that available techniques have on the software process. Determine which techniques are appropriate for the environment and can improve the software,

3. **Infuse**—After identifying process improvements, package the technology in a form to be applied and useful to the production organization.

The approach taken to attain these three generalized goals has been to apply potentially beneficial techniques to the development of production software and to measure the process and product in reasonable detail to assess quantifiably the applied technology. Measures of concern, such as cost, reliability, and/or maintainability, are defined as the organization determines the major near- and long-term objectives for its software development process improvement program. Once those objectives are determined, the SEL staff designs the experiment, that is, defines the particular data to be captured and the questions that must be addressed in each experimental project.

All of the experiments conducted by the SEL have occurred within the production environment of the Flight Dynamics software development facility at NASA/GSFC. This software can be characterized as scientific, nonembedded, relatively complex software. Projects are typically developed in FORTRAN, although about 25 percent of the projects utilize another language such as Ada, C, or PASCAL. The duration of each effort normally runs from 2 to 3-1/2 years. with an average staff size of approximately 15 software developers. The average size of one of these projects runs approximately 175,000 source lines of code (counting commentary), with about 25 percent reused from previous development efforts. Since this environment is relatively consistent, it is conducive to the experimentation process. In the SEL, there exists a homogeneous class of software, a stable development environment, and a very controlled, consistent management and development process.

The following three major functional organizations support the experimentation and study within the SEL environment:

1. Software developers, who are responsible for producing the flight dynamics application software.

2. Software engineering analysts, who are the researchers responsible for carrying out the experimentation process and producing study results

3. Data base support staff, who are responsible for collecting, checking and archiving all of the data and information collected from the development efforts.

Since its inception in 1976, the SEL has carried out studies and experiments involving nearly 100 flight dynamics projects. Detailed data have been collected and studied, and numerous reports and journal papers have been produced. From all of this analysis and from all of these studies, seven key points have been identified that reflect insight gained by the SEL and which are the guiding principles for future development and research within this organization. These seven key points, described under EXPERIENCES below, address nearly every aspect of the activities within the SEL experimentation process and should provide some guidance to other organizations involved with software development and/or software engineering research.

## II. EXPERIENCES

### *Point 1: Measurement is an Essential Element of Software Process Improvement*

It is imperative that software measurement be an integral component of any software process assessment or process improvement program. Although this point may seem obvious to most, there is evidence that occasionally organizations may initiate 'process improvement' efforts without fully developing considerations and plans for applying measurement. In addition to providing some mechanism for determining the baseline characteristics of the software process before any change is adopted, the measurement aspect is necessary to gauge the impact of any change to the software process. Not only does an organization need to understand if and by how much software 'quality' is improving through enhanced software processes, but even the more elementary assessment as to whether there is any consistency within an organization's process (is it measurable) as well as 'is change observable?' are points addressed only through measurement.

The SEL has focused on software measurement as a tool to aid in determining the effect that changes to the software process may have on attributes of concern (cost, quality, reliability, ...). In addition to this, it has become evident that both the measurement process as well as the measurements themselves are extremely valuable software management tools. The Flight Dynamics environment has adopted the SEL measurement process as an integral component of the development standards and applies key measures in planning and tracking the progress of projects. There are eight key measures that the Flight Dynamics software organization has adopted as essential management aids [Reference 2].

One additional point that has become apparent for the SEL after 15 years of software measurement is that the adoption of an effective measurement program is not cost prohibitive. In fact, the measurement collection process can be essentially a zero cost impact to the development project—provided that a well thought-out set of measures is adopted rather than an ill-conceived large number of measures. The most significant cost attributable to a measurement program is that of processing and effectively analyzing/utilizing the information—and this must be done. It should be obvious that the mere process of collecting measures will be of absolutely no value (even negative value) unless the information is analyzed and factored back into the software process itself. Although the cost impact to the development projects themselves can be near 0-percent overhead, the cost of processing and analyzing information as part of an effective process improvement program will add 10 percent to 15 percent of the development cost.

*Point 2: Many Diversions Exist to a Successful Process Improvement Program*

Most software organizations have either attempted or at least seriously considered adopting a software measurement program. Unfortunately, there are too few examples of projects or companies in general that have sustained an effective measurement program. Many reasons exist to explain why such a critical element of software engineering consistently fails, and the SEL has experienced most of the significant impediments and pit-falls that can discourage the use of measurement programs. Three of the most significant diversions that the SEL has experienced and which seem to plague numerous other software organizations include the following:

1. **Excessive planning/replanning**—If someone is serious about starting a measurement program, it is more important to get started with a very small effort as opposed to developing the full set of measures, tools, analysis approaches, etc. The key is to start small and grow with experience—but at least start.

2. **Over-Dependence on Statistical Analysis**—Although the use of analysis tools is certainly required in applying measurement to the software process, there are occasions when the analysts attempt to uncover more information from available data (measures) than is reasonable. Intuition is an excellent starting point for the analysis process, and it is certainly enhanced or challenged by statistical information; but there is danger in assuming that the mathematical interpretation of some quite inexact figures can lead to a more accurate conclusion than the figures dictate. Too often, the common sense of experienced software developers and managers is ignored in favor of statistics produced with possibly flawed, misinterpreted, or missing data.

3. **Looking Under the Lamp-Post**—As in the case of the person who has lost a coin in the dark part of a street, but chooses to search for it beneath the lighted lamp-post because it is easier to see, software engineers occasionally address those software topics that are easiest to study as opposed to those that are the real problems for software development/management. There has been significant effort put into studying, rebuilding, and modifying such tools as code analyzers, auditors, converters, graphical design aids, etc., when there is doubt as to the real driving need for such small modifications to very old and well-understood technology. Excessive studies continue to be conducted on antiquated complexity metrics and on 15-year-old cost-modeling techniques, when there are extremely difficult areas to be addressed.

such as design measures, software specification tools and analyzers, and integrated environments.

*Point 3: People Are the Most Important Resource/Technology*

In reviewing the results of the numerous studies and experiments that the SEL has conducted over the past 15 years, it is apparent that the most effective technologies, that result in the most significant benefit, are those that leverage the skills of the software developers themselves. Numerous studies outside of the SEL environment have shown that the productivity of individuals can easily vary by as much as a factor of 10 to 1. In addition to this fact, SEL studies have indicated that those methods and tools that emphasize human discipline are far more effective than those that merely attempt to take work away from the developers.

Such software techniques as code reading, inspections, walk-throughs, and all aspects of 'Cleanroom' are examples that have been shown to be extremely effective [Reference 3]. All of these are directed toward maximizing the potential of individuals as opposed to removing the individual from the process.

*Point 4: Environmental Characteristics Should Dictate Selected Software Engineering Techniques*

Experiences in the SEL have verified the expectation that standards, methods, and, in general, all software engineering approaches must be tailored to specific environments. Although the point seems to be obvious, we as practitioners and software engineers often attempt to apply a new technique or method expecting certain improvements without first analyzing whether the methodology is addressing the needs of the environment. For example, if a development organization historically produces highly reliable, well-tested software, then there is probably little benefit to be derived from modifying the testing approach by applying an automated test generator.

Additionally, it must be understood that all software environments evolve with time and undergo some level of change. Because of this, the overall process must be continually observed to identify changing and evolving practices in order to respond with the most appropriate modifications to methods, tools, etc.

*Point 5: Automation is an Instrument of Process Improvement, Not a Replacement for Process Understanding*

As was mentioned previously, the foundation of the process improvement paradigm is that of understanding the software process and associated products— which may then lead to assessment and to process improvement. Automated tools may provide some help in understanding this process, but too often we expect the automation process to resolve problems that we don't clearly understand in a manual sense. If a software developer or manager cannot clearly represent and grasp some process manually, the application of a software tool will only make the process less understood and more ill defined. This overreliance on automation is occasionally exemplified by organizations that move too swiftly in the adoption of CASE (or related technology) before the overall development characteristics are analyzed and the need for automated tools is identified. Another example can be seen in the attempts of managers to use code analyzers, auditors, and automated complexity analyzers to gain insight into 'complexity' without being able to discern this trait in any of the products or processes.

Although it is unwise to try to automate immature processes or to apply tools where no tool is needed, there are excellent examples of tools and overall automation that reflect significant advances in applying this technology to recently maturing disciplines. Such an example is the recent development of the 'Software Management Environment (SME)' [Reference 4] which is used by the Flight Dynamics Division at NASA/GSFC to automate the use and interpretation of historical software data, models, measures, and intuition toward the management of active software projects.

*Point 6: Heritage of the Environment Will Strongly Influence the Software Process*

It seems rather obvious to say that a development environment has its own characteristics of process and process improvement and that the heritage of this environment will certainly influence the development of project after project, but the level to which the past performance of a software organization dominates even the use of significantly different technology is quite surprising. It is the most prevalent influence that the SEL has seen in its environment where evolving, new technology is continually applied to observe impacts to the software process, and major changes to methodology are continually made.

For example, the technology impact from the introduction of Ada into the SEL environment has been under study since 1985 when the first Ada system was developed. One of the early expectations was that there would be a significant change to the effort distribution over the implementation (design, code, test) period for these Ada systems in comparison with previous FORTRAN systems. To date, this has not been observed in the SEL—effort distributions based on these activities have remained essentially the same and continue to reflect past SEL experience. Since changes to an established development process occur slowly, the changes themselves tend to evolve over time as more experience is gained with the new technology. As expected, the use of various Ada constructs (generics, packages, typing, tasking) in the more recent Ada projects is considerably different than in earlier systems.

*Point 7: Software Can Be Measurably Improved Through Appropriate Use of Available Technologies*

Possibly the most important point evinced as a result of the 15 years of study within the SEL is that software (both the process and products) can be quantifiably improved through the selected application of methods, tools, and models that exist today. It has often been argued that since 'the human being' is the dominant factor in any software project, the modification or application of any approach to the development process cannot be observed nor can it have any significant impact on improving measures of importance.

Experience has verified the fact that researchers often attempt to apply and measure, to extremely detailed levels, techniques that may not be 'measurable'; however, it has also shown that overall trends are definitely measurable when the measurement process becomes an integral part of the applied methodology. As was described previously, because a specific software technology may not be applicable to all environments, each environment must clearly define its goals, strengths, and weaknesses before it attempts to observe positive impacts from some modified approach.

There are specific methodologies that the SEL has applied and measured over a long period of time and that have been verified as having positive impact on the cost, reliability, and overall quality of software within the Flight Dynamics environment. Such techniques include 'Reading' (as applied to design, code, and test), Ada, object-oriented development, design criteria (e.g., strength), measurement, and many others. There are software practices that will significantly and measurably improve the software within any specific environment.

## III. OVERALL COST/IMPACT OF THE MEASUREMENT EFFORTS IN THE SEL

For 15 years, NASA has been funding these efforts to carry out experiments and studies within the SEL. There has been significant cost and general overhead to this effort, and a logical question that is asked is 'Has it all been worth it?' The answer is a resounding YES. Not only has the expenditure of resources been a wise investment for the Flight Dynamics area within NASA, but members of the SEL strongly believe that such efforts should be commonplace throughout the Agency as well as throughout the software community. The benefits far outweigh the cost.

Since the SEL's inception in 1976, NASA has spent approximately $14 million dollars in the three major support areas required by this type of study environment: research (such as defining studies and analyzing results), technology transfer (such as producing standards and policies), and data processing (such as collecting forms and maintaining data bases). Additionally, approximately 50 staff-years of NASA personnel effort has been expended on the SEL. During this same time period, the Flight Dynamics area has spent approximately $130 million on building operational software, all of which has been part of the study process to some degree.

During the past 15 years, the SEL has certainly had significant impact on the software being developed in the local environment, and there is strong reason to believe that many of the results and studies of the SEL have had favorable impact on a domain broader than just the NASA Flight Dynamics area. Examples of the changes that have been observed include the following:

1. The 'manageability' of software has improved dramatically. In the late 1970s and early 1980s, this environment experienced wide variation from project to

project in productivity, reliability, and quality. Today, however, the SEL has excellent models of the process; has well-defined methods; and is able to predict, control, and manage the cost and quality of the software being produced.

2. The cost per line of new code has decreased somewhat (about 10 percent), and at first glance this may imply that the SEL has failed at improving productivity. Although the SEL finds that the cost to produce a new source statement is nearly as high as it was 14 years ago, there is appreciable improvement in the functionality of the software, as well as tremendous increases in the complexity of the problems being addressed. Also, there has been an appreciable increase in the reuse of software (code, design, methods, test data, etc.), which has driven the overall cost of the equivalent functionality down significantly. When we merely measure the cost to produce one new source statement, the improvement is small; but when we measure overall cost and productivity, the improvement is significant.

3. Reliability of the software has improved by 35 percent. As measured by the number of errors per thousand lines of code (E/KSLOC), the Flight Dynamics software has improved from an average of 8.4 E/KSLOC in the early 1980s to approximately 5.3 E/KSLOC today. These figures cover the software phases up through and including acceptance testing (beginning of operations). Although the operational and maintenance data are not nearly so extensive as the development data, the small amount of data available indicates significant improvement in that area as well.

4. Other measures include the effort put forth in rework (changing, fixing, etc.) and in overall software reuse. These measures also indicate a significant improvement to the software within this one environment.

In addition to the common measures of software (cost, reliability, etc.), there are many other major benefits derived from such a 'measurement' program as that in the SEL. Not only has our understanding of software significantly improved within the research community, but this understanding is apparent throughout the entire development community within this Flight Dynamics environment. Not only have the researchers benefited, but it is obvious that the developers and managers who have been exposed to this effort are much better prepared to plan, control, assure, and, in general, develop much higher quality systems. One view of this entire program is that it is a major 'training' exercise within a large production environment, and the 800 to 1000 developers and managers who have participated in development efforts studied by the SEL are much better trained and effective software engineers.

## REFERENCES

1. Software Engineering Laboratory, SEL-82-906, *Annotated Bibliography of Software Engineering Laboratory Literature*, P. Groves and J. Valett, November 1990

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

3. Software Engineering Laborary, SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis*, S. Green et al., March 1990

4. Software Engineering Laboratory, SEL-89-003, *Software Management Environment (SME) Concepts and Architecture*, W. Decker and J. Valett, August 1989

# VIEWGRAPH MATERIALS

# FOR THE

# F. MCGARRY PRESENTATION

# TOWARDS UNDERSTANDING SOFTWARE

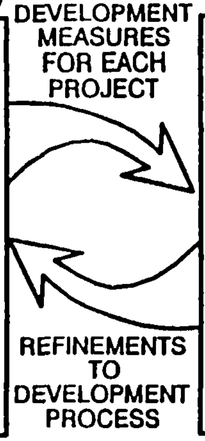
## 15 YEARS in the Software Engineering Laboratory (SEL)

Frank McGarry
Rose Pajerski
and SEL Staff

# SEL ENVIRONMENT

## DEVELOPERS
## (DEVELOP FLIGHT DYNAMICS S/W)

| | |
|---|---|
| STAFF | 150-250 (FTE) |
| TYPICAL PROJECT SIZE | - 150-200 KSLOC |
| ACTIVE PROJECTS | - 6-10<br>(AT ANY GIVEN TIME) |
| PROJECT STAFF SIZE | - 15-25 PEOPLE |
| 1976-1990 | - 75 PROJECTS |


DEVELOPMENT MEASURES FOR EACH PROJECT
REFINEMENTS TO DEVELOPMENT PROCESS

## S/W ANALYSTS
## (STUDY PROCESS)

STAFF 5-10 RESEARCHERS

FUNCTION
- SET GOALS/QUESTIONS/ METRICS
  - DESIGN STUDIES/ EXPERIMENTS
- ANALYSIS/RESEARCH
- REFINE S/W PROCESS
  - PRODUCE REPORTS/ FINDINGS

1976-1990 - 250 REPORTS/DOCUMENTS

## DATA BASE SUPPORT (MAINTAIN/QA SEL DATA)

STAFF 2-5 (FTE)

FUNCTION
- PROCESS FORMS/DATA
- QA
- RECORD/ARCHIVE DATA
- MAINTAIN SEL DATA BASE
- OPERATE SEL LIBRARY

SEL DATA BASE

FORMS LIBRARY

REPORTS LIBRARY

1976-1990 - OVER 150,000 "FORMS"

F. McGarry
NASA/GSFC
Page 11 of 32

A498.009

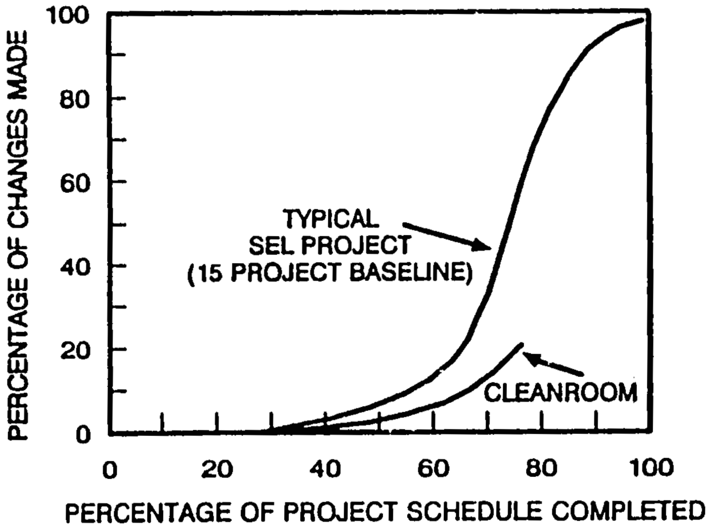
# MEASUREMENT IS AN ESSENTIAL ELEMENT OF S/W PROCESS IMPROVEMENT

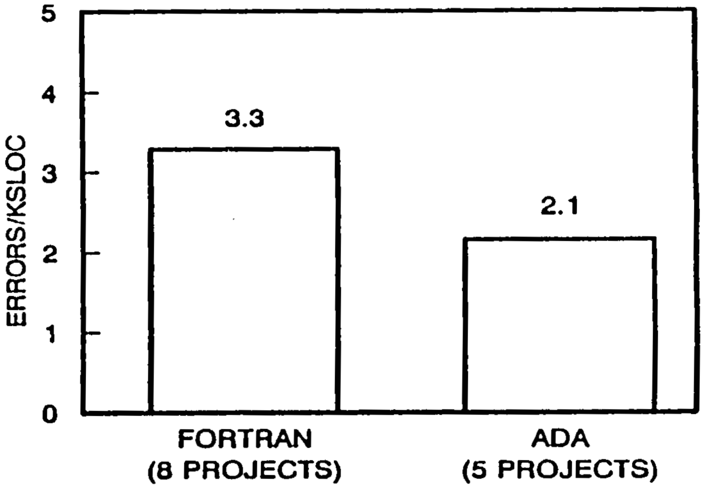
①

- MEASURES DEFINE PROCESS/PRODUCT BASELINE AND GAUGE CHANGE
    - ONLY MEANS OF PROVIDING UNDERSTANDING
    - WITHOUT MEASUREMENT CANNOT DETERMINE CHANGE/IMPROVEMENT
- MEASURES - SIGNIFICANT ASSET TO S/W MANAGEMENT/DEVELOPMENT
    - VITAL FOR PLANNING/ESTIMATING
    - PROVIDES INSIGHT TO HEALTH OF PROJECTS
- MEASUREMENT IS NOT COST PROHIBITIVE
    - EXISTS SMALL/CRITICAL SET OF MEASURES
    - CRITICAL SET LESS THAN 2% IMPACT TO PROJECT
    - BENEFITS FAR OUTWEIGH THE OVERHEAD

# MEASURES - GAUGING CHANGE AND IMPROVEMENT IN THE SEL

1-1





OBSERVING IMPACTS OF PROCESS CHANGE

DETERMINE IMPROVEMENT DUE TO PROCESS CHANGE

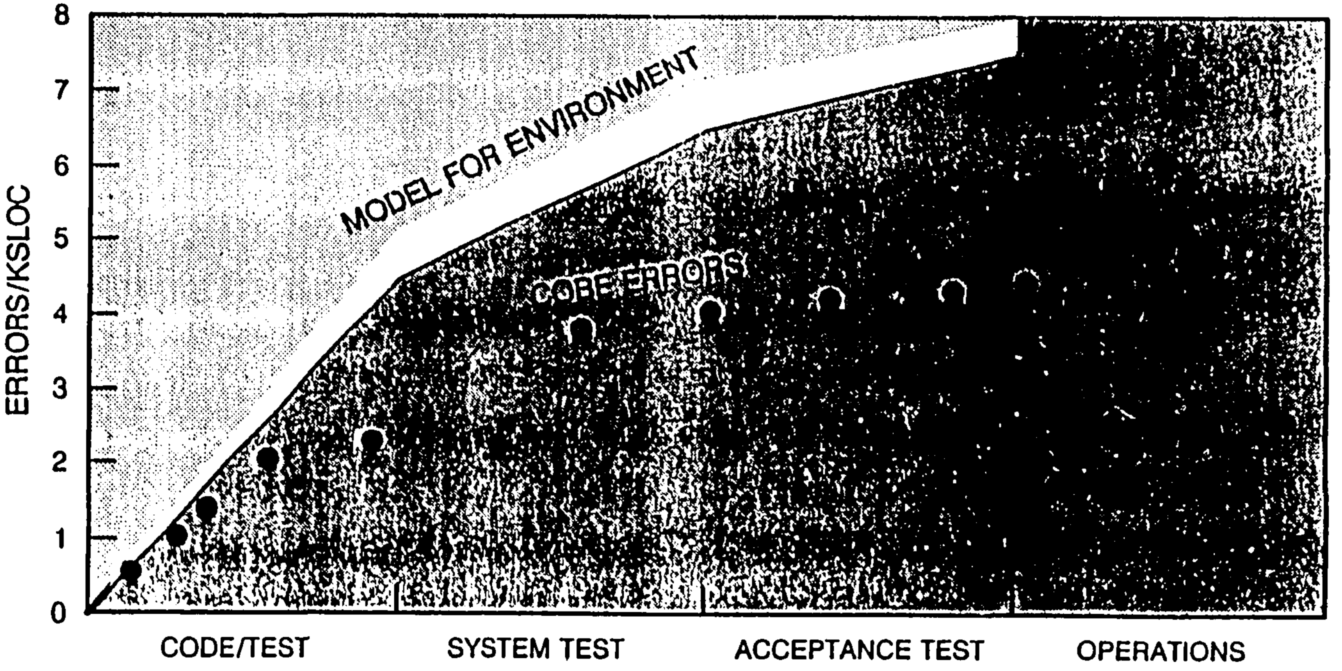
F. McGarry
NASA/GSFC
Page 13 of 32

A496.012

## • MEASUREMENT AS A MANAGEMENT AID

(1-2)

TRACKING "COBE" RELIABILITY



MEASURING ERROR RATES CAN PROVIDE EARLY INDICATION OF SOFTWARE QUALITY

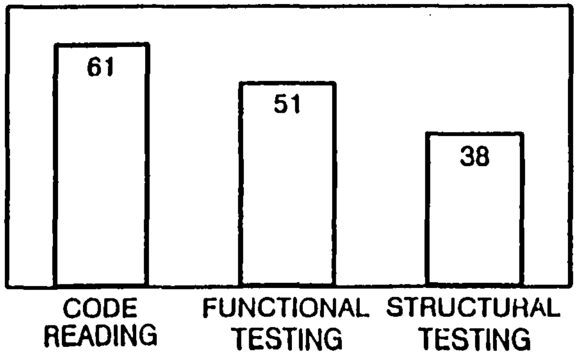
# PEOPLE ARE MOST IMPORTANT RESOURCE/TECHNOLOGY

③

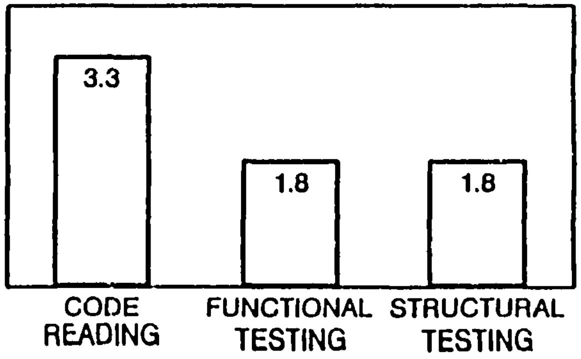
## TEST TECHNIQUES EXPERIMENT DESCRIPTION

- 3 APPROACHES STUDIED
  - CODE READING
  - FUNCTIONAL TESTING
  - STRUCTURAL TESTING
- 32 PEOPLE PARTICIPATED (GSFC, UM, CSC)
- 3 UNIT-SIZED (100 SLOC) PROGRAMS SEEDED WITH ERRORS

% OF FAULTS DETECTED



NUMBER OF FAULTS DETECTED PER HOUR OF EFFORT



EFFECTIVE TECHNOLOGY SHOULD FOCUS ON "PERSONNEL" POTENTIAL

# MANY DIVERSIONS EXIST TO A SUCCESSFUL PROCESS IMPROVEMENT PROGRAM

②

(DIVERSIONS THE SEL HAS BEEN THROUGH)

- EXCESSIVE PLANNING/REPLANNING
  - JUST DO IT/START SMALL
  - LEARN WITH EXPERIENCE
  - RELY ON LOCAL STANDARDS (E.G., TERMINOLOGY)
- OVER DEPENDENCE ON STATISTICAL ANALYSIS
  - INTUITION IS A VERY USEFUL STARTING POINT
  - MAKE USE OF SUBJECTIVE DATA
- LOOKING UNDER THE LAMP POST
  - CODE ANALYZERS/CONVERTERS
  - COMPLEXITY METRICS
  - DESIGN GRAPHIC AIDS

# ENVIRONMENTAL CHARACTERISTICS SHOULD DICTATE SELECTED SOFTWARE ENGINEERING TECHNIQUES

④

- SPECIFIC MEASURES/TECHNIQUES MAY NOT APPLY TO ALL "DOMAINS"

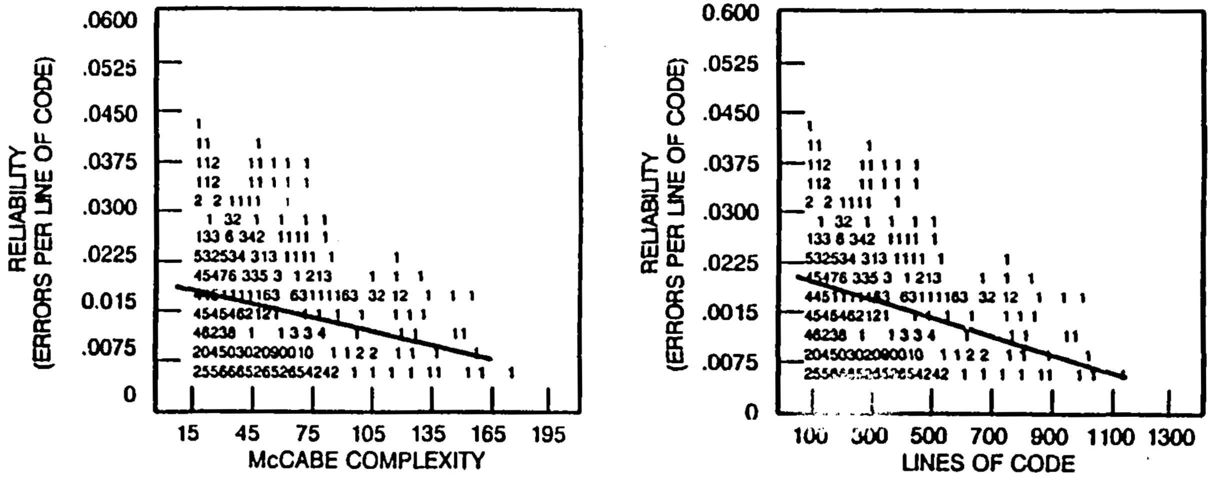
- AS ENVIRONMENT EVOLVES, METHODOLOGIES SHOULD FOLLOW (AND LEAD)

- TAILOR STANDARDS/POLICIES

# SPECIFIC MEASURES MAY NOT APPLY TO ALL "DOMAINS"

4-1

## SOFTWARE MEASURES IN THE SEL



CORRELATIONS

| | TOTAL LINES | EXECUTABLE LINES | McCABE COMPLEXITY | HALSTEAD LENGTH |
|---|---|---|---|---|
| HALSTEAD LENGTH | 0.85 | 0.91 | 0.91 | 1.00 |
| McCABE COMPLEXITY | 0.81 | 0.87 | 1.00 | |
| EXECUTABLE LINES | 0.84 | 1.00 | | |
| TOTAL LINES | 1.00 | | | |

A498.023

SAMPLE OF 688 MODULES

# CHARACTERISTICS OF EFFECTIVE POLICIES

4-2

STANDARDS/POLICIES MUST BE:

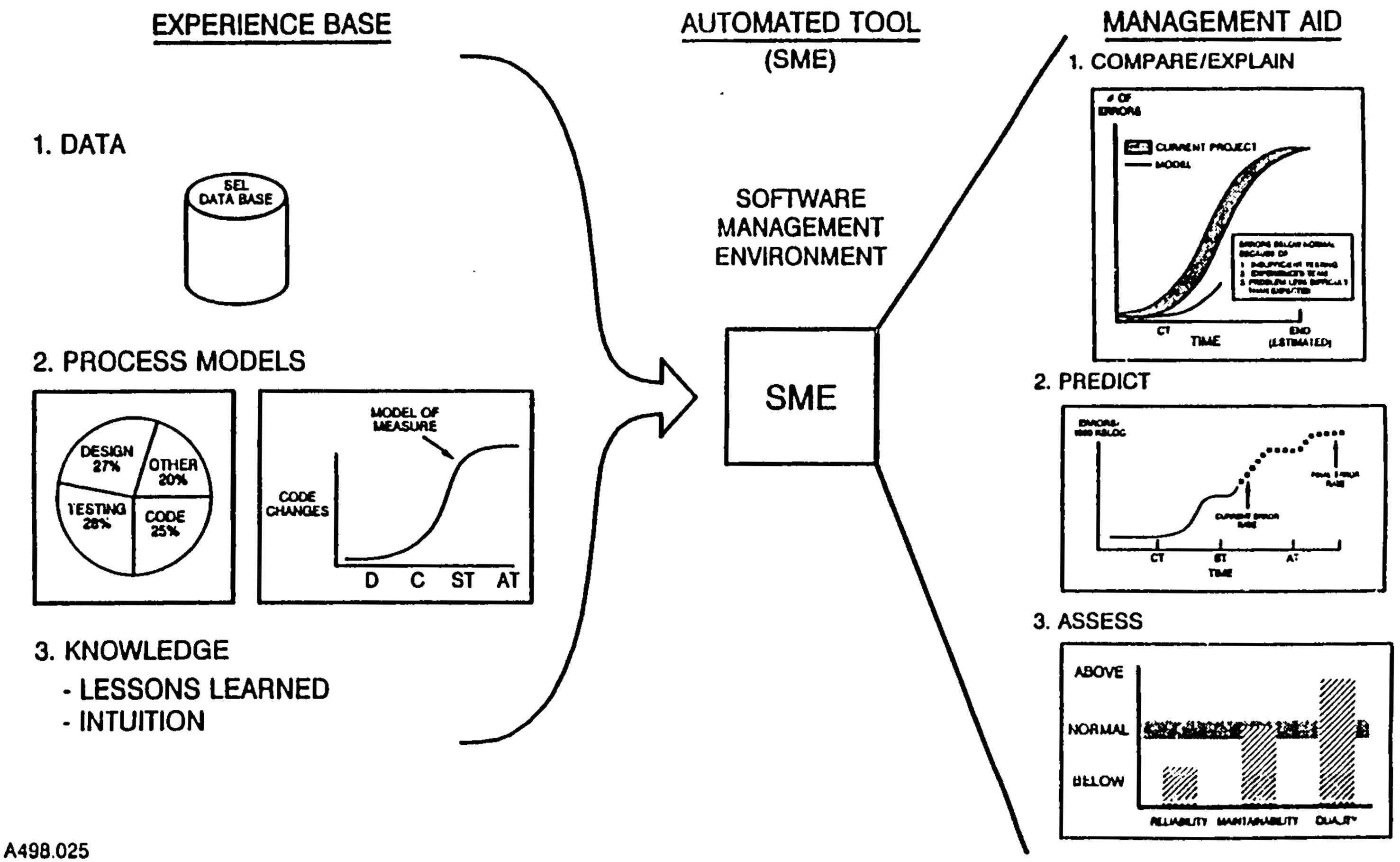
| | |
|---|---|
| 1. WRITTEN | ● MAY BE COMBINED (GENERIC AND TAILORED)<br>● CAREFULLY "PRESENTED" |
| 2. UNDERSTOOD | ● NOT TO INCLUDE EXCESSIVELY ALIEN TECHNOLOGY<br>● TRAINING OFTEN REQUIRED |
| 3. "LEGACY-BASED" | ● DERIVED FROM NEED/LEGACY<br>● CONTINUALLY EVOLVING<br>● ALL ELEMENTS ARE "DEFENDABLE" |
| 4. ENFORCED | ● SUPPORTED BY MANAGEMENT<br>● LIMITED "DETAIL" |
| 5. MEASURABLE | ● OBSERVABLE (CAN TELL IF IT'S BEING FOLLOWED)<br>● REQUIRES SELF-EVALUATION |

# AUTOMATION IS AN INSTRUMENT OF PROCESS IMPROVEMENT (NOT A REPLACEMENT FOR PROCESS UNDERSTANDING)

⑤

- TOOLS CAN PROVIDE SIGNIFICANT BENEFIT TO WELL-DEFINED EXPERIENCE BASE (E.G., SME IN THE SEL)
- "IMMATURE" PROCESSES ARE NOT AUTOMATABLE
  (IF YOU CAN'T DO IT MANUALLY - DON'T TRY TO AUTOMATE IT)
  (E.G., OVER RELIANCE ON CASE/ANALYZERS/AUDITORS/ MEASUREMENT TOOLS)
- EFFECTIVE TOOLS MUST ADDRESS DEFINED PROCESS NEED
  (MATCH SOLUTION TO PROBLEM)
  (E.G., OVERUSE OF CODE TRANSLATOR/CODE ANALYZERS/ TEST GENERATORS, ...)

# (5-1) AUTOMATING A WELL-UNDERSTOOD “EXPERIENCE BASE” IN THE SEL (SOFTWARE MANAGEMENT ENVIRONMENT (SME))

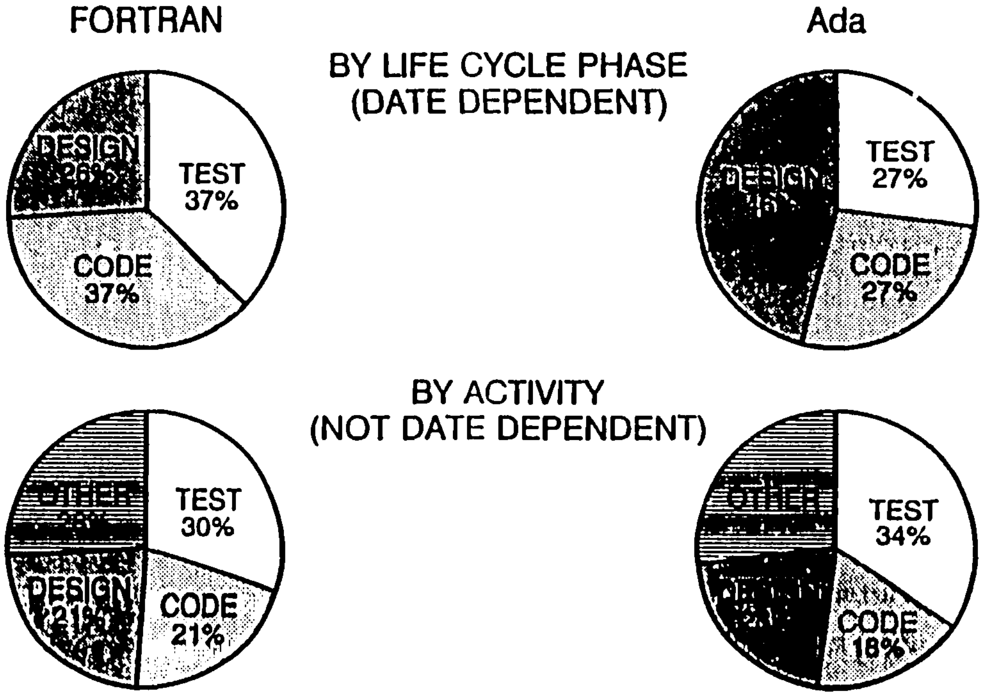


F. McGarry
NASA/GSFC
Page 21 of 32

A498.025

# HERITAGE OF ENVIRONMENT WILL STRONGLY INFLUENCE PROCESS

⑥



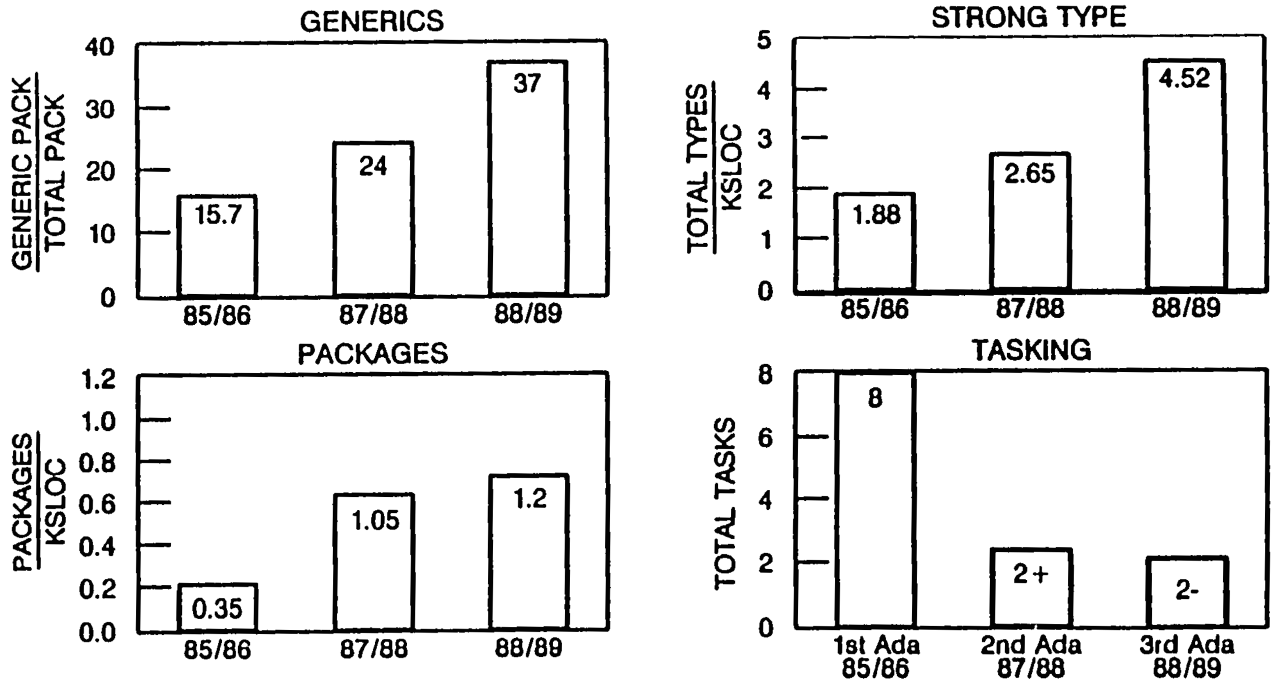
MAJOR DATES CHANGED (CDR, ...) BUT EFFORT DISTRIBUTION STILL QUITE SIMILAR

*BASED ON 5 Ada AND 8 FORTRAN PROJECTS OF SIMILAR TYPE IN THE SEL

# SIGNIFICANT PROCESS CHANGE REQUIRES SIGNIFICANT EFFORT/TIME

(6-1)

USE OF Ada FEATURES



- ● USE OF Ada FEATURES CHANGES APPRECIABLY WITH EXPERIENCE
- ● NOT ALL FEATURES APPROPRIATE FOR APPLICATION

# SOFTWARE CAN BE MEASURABLY IMPROVED THROUGH APPROPRIATE USE OF AVAILABLE TECHNOLOGIES

7

## EXAMPLES IN ONE ENVIRONMENT (SEL)

| TECHNOLOGY | DEMONSTRATED IMPACT |
|---|---|
| - "READING" | REPEATEDLY SHOWN TO IMPROVE SOFTWARE RELIABILITY (NO ADDITIONAL COST) |
| - DESIGN CRITERIA (STRENGTH) | DEMONSTRATED TO PRODUCE MORE ERROR FREE SOFTWARE |
| - Ada | SIGNIFICANT COST BENEFIT THROUGH REUSE |
| - OOD | REUSE |
| - CLEAN ROOM | SIGNIFICANT IMPROVEMENT IN RELIABILITY AND PRODUCTIVITY (ALSO RESOURCE CONSUMPTION DOWN) |
| - MANAGEMENT/ MEASUREMENT | MAJOR IMPROVEMENT IN PLANNING, ADJUSTING AND CONTROL<br>- COST ESTIMATION<br>- SCHEDULE ESTIMATION |

# ASSESSING "STRENGTH" AND "SIZE" AS A STANDARD FOR DESIGN

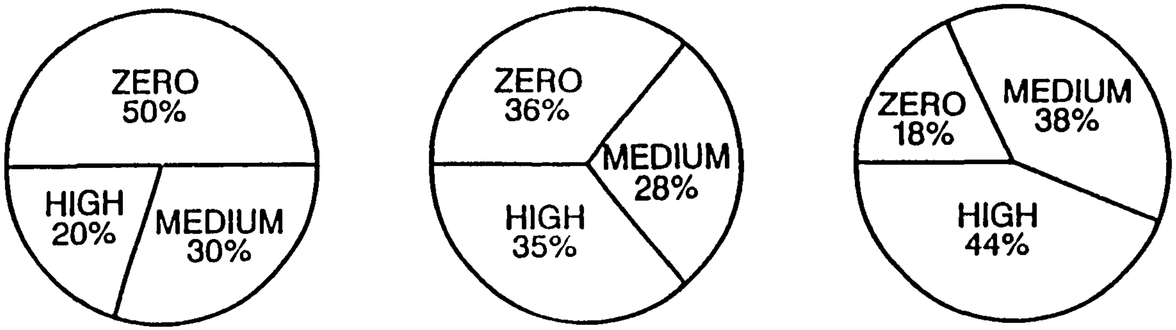
7-1

EXPERIMENT:

- 450 FORTRAN MODULES (ACROSS 4 SYSTEMS - OVER 20 DEVELOPERS)
- DETAILED COST AND ERROR DATA ON ALL MODULES
- DETERMINE RELATIONSHIPS: "STRENGTH" TO RELIABILITY AND "SIZE" TO RELIABILITY

RESULTS:

FAULT RATE FOR CLASSES OF MODULE STRENGTH



HIGH STRENGTH

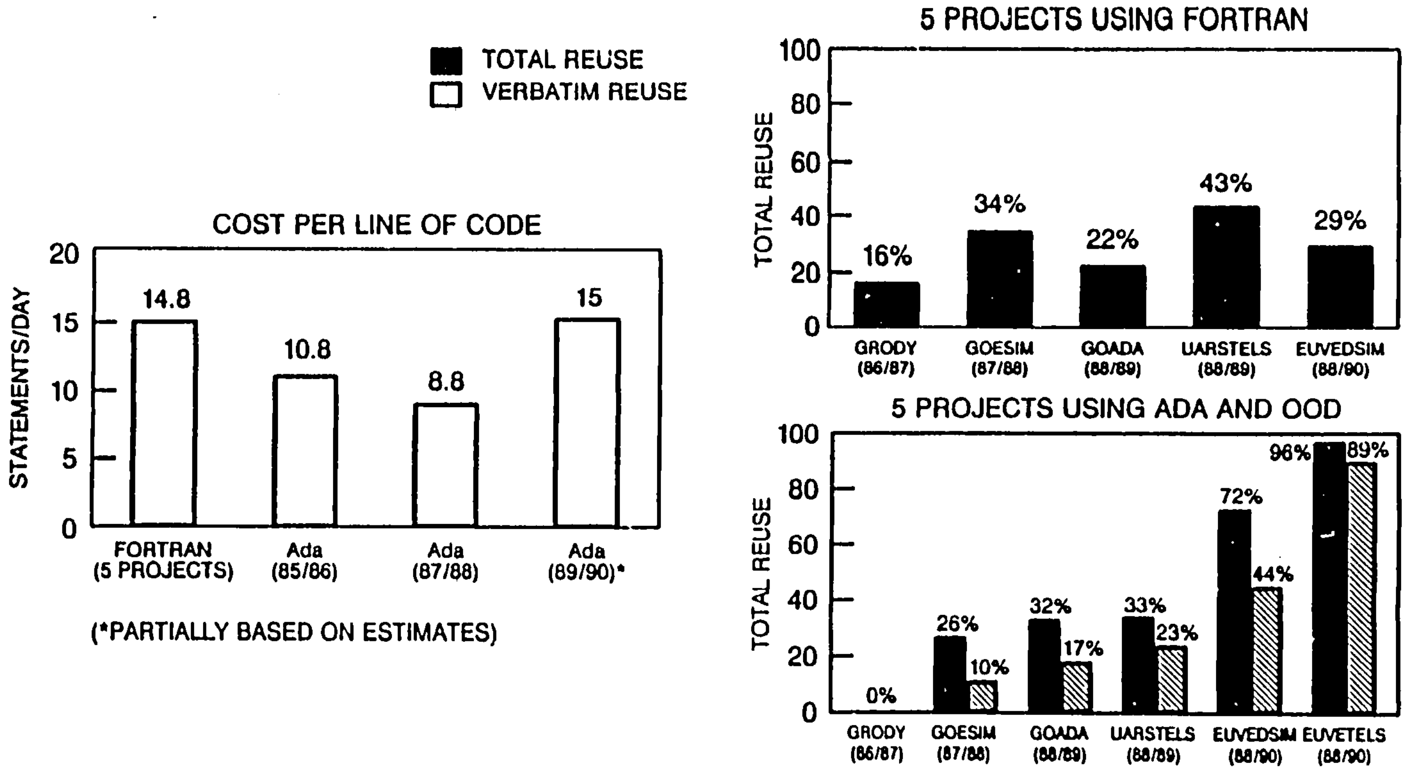
MEDIUM STRENGTH

LOW STRENGTH

# DESIGN MEASURES SUMMARY

- GOOD PROGRAMMERS TEND TO WRITE HIGH-STRENGTH MODULES
- GOOD PROGRAMMERS SHOW NO PREFERENCE FOR ANY SPECIFIC MODULE SIZE
- OVERALL, HIGH-STRENGTH MODULES HAVE *A LOWER FAULT RATE AND COST LESS* THAN LOW-STRENGTH MODULES
- OVERALL, LARGE MODULES COST LESS (PER EXECUTABLE STATEMENT) THAN SMALL MODULES
- FAULT RATE IS NOT DIRECTLY RELATED TO MODULE SIZE

# Ada (AND OOD)* IMPACTS ON "COST" FROM SEL EXPERIENCES



(*PARTIALLY BASED ON ESTIMATES)

1. DEVELOPMENT COST PER STATEMENT HAS BEEN NO "CHEAPER" FOR ADA
2. REUSE POTENTIAL OF Ada IS SIGNIFICANT

*ALL Ada PROJECTS APPLIED OOD TECHNIQUES

F. McGarry
NASA/GSFC
Page 27 of 32

# HAS THE EFFORT BEEN WORTH IT?
# (1975 - 1990)

|  |  |
|---|---|
| • SEL EXPENDITURES (1990 DOLLARS) |  |
| - RESEARCH SUPPORT (UNIVERSITY)<br>(EXPERIMENTATION, ANALYSIS, RESEARCH, REPORTS, ...) | $2.5M |
| - RESEARCH AND TECH TRANSFER (CSC)<br>(ANALYSIS, RESEARCH, REPORTS, OVERHEAD TO DEVELOPMENT PROJECTS) | $5.5M |
| - DATA PROCESSING AND GENERAL SUPPORT (CSC AND OTHERS)<br>(PROCESS/QA DATA, SEL DATA BASE, REPORTS, ...) | $6.0M |
| • PRODUCTION SOFTWARE (FLIGHT DYNAMICS) DEVELOPED<br>(CSC AND NASA) | $130M |

# HAS THE EFFORT BEEN WORTH IT?

## (1975 - 1990)

IMPACT OF SEL RESEARCH*

| | 1976 - 1980 | 1986 - 1990 |
|---|---|---|
| MANAGEABILITY | ● COMPLETE DEPENDENCE ON PERSONNEL CAPABILITY<br>● WIDE VARIANCE IN COST/QUALITY<br>● NO GUIDANCE FOR SELECTING METHODS | ● PROCESS-MODELED AND EFFECTIVE<br>● SOFTWARE MORE PREDICTABLE, CONSISTENT<br>● RATIONALE FOR METHODS USED EXISTS |
| COST PER LINE OF CODE | ≈ 24 SLOC/DAY | ≈ 24 SLOC/DAY |
| RELIABILITY (UNIT TEST THRU ACCEPTANCE) | 8.4 E/KSLOC | 5.3 E/KSLOC |
| CODE REUSE | 15-25% | 25-35% |
| REWORK | 35-40% OF TOTAL EFFORT | 20-30% |

*PROBLEM COMPLEXITY AND SUPPORT ENVIRONMENT HAVE ALSO CHANGED SIGNIFICANTLY

# HAS THE EFFORT BEEN WORTH IT?

## FINAL OBSERVATIONS

- "OUR" UNDERSTANDING OF SOFTWARE HAS IMPROVED SIGNIFICANTLY (WE DO SOFTWARE BETTER)
- CONTRIBUTIONS TO SOFTWARE RESEARCH AND DEVELOPMENT (MEASUREMENT, MANAGEMENT, EXPERIENCE BASE, ...)
- PROFESSIONAL DEVELOPMENT OF DEVELOPERS, MANAGERS, RESEARCHERS
- "NEW" AWARENESS BY MANAGERS, DEVELOPERS (SOFTWARE CAN BE ENGINEERED)

# ONGOING/FUTURE ACTIVITIES FOR THE SEL

GENERAL

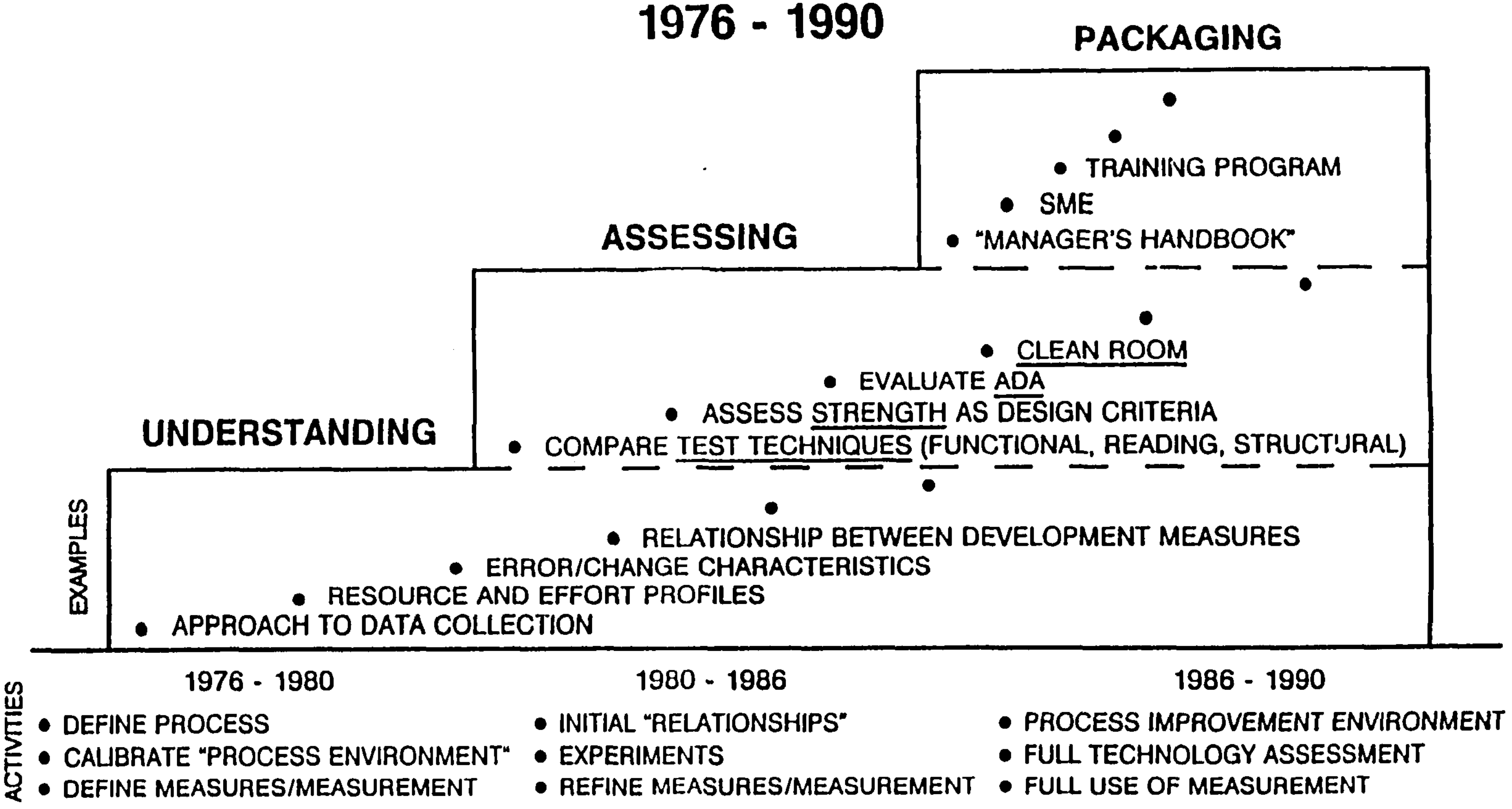
- CONTINUE EVALUATION OF "PROCESS IMPROVEMENT"
  - G/Q/M
  - EXPERIMENTATION
  - REFINEMENT
- DOMAIN ANALYSIS FOR "EXPERIENCE BASE"
  - RELEVANCE TO OTHER "ENVIRONMENTS"
  - CHARACTERIZING DOMAIN OF "EXPERIENCE BASE"
- EXPANSION OF LIFE CYCLE ANALYZED
  - MAINTENANCE
  - SPECS/REQUIREMENTS
  - EXPANDED MEASUREMENT

CURRENT/NEAR FUTURE STUDIES

- CLEAN ROOM (3 ACTIVE PROJECTS)
- Ada (3 PROJECTS)
- OOD (1 ONGOING EXPERIMENT - 2 PLANNED)
- CASE (1 ACTIVE PROJECT)
- REUSE (USING EXISTING SEL PROJECTS)
- MAINTENANCE (3 PROJECTS FOR ANALYSIS)
- TESTING STRATEGIES (EXISTING SEL PROJECTS)
- MEASUREMENT (CHARACTERIZING DESIGNS)

# STUDIES IN THE SEL

## 1976 - 1990



PACKAGING

- ●
- ●
- TRAINING PROGRAM
- SME
- "MANAGER'S HANDBOOK"

ASSESSING

- ●
- ●
- CLEAN ROOM
- EVALUATE ADA
- ASSESS STRENGTH AS DESIGN CRITERIA
- COMPARE TEST TECHNIQUES (FUNCTIONAL, READING, STRUCTURAL)

UNDERSTANDING

- ●
- ●
- RELATIONSHIP BETWEEN DEVELOPMENT MEASURES
- ERROR/CHANGE CHARACTERISTICS
- RESOURCE AND EFFORT PROFILES
- APPROACH TO DATA COLLECTION

EXAMPLES

ACTIVITIES

| 1976 - 1980 | 1980 - 1986 | 1986 - 1990 |
|---|---|---|
| ● DEFINE PROCESS | ● INITIAL "RELATIONSHIPS" | ● PROCESS IMPROVEMENT ENVIRONMENT |
| ● CALIBRATE "PROCESS ENVIRONMENT" | ● EXPERIMENTS | ● FULL TECHNOLOGY ASSESSMENT |
| ● DEFINE MEASURES/MEASUREMENT | ● REFINE MEASURES/MEASUREMENT | ● FULL USE OF MEASUREMENT |

EVOLVING TO AN EFFECTIVE
"PROCESS IMPROVEMENT" ENVIRONMENT

F. McGarry
NASA/GSFC
Page 32 of 32

A498.031

## SESSION 2—PROCESS IMPROVEMENT

R. E. Nance, VPI

M. B. Arend, McDonnell Douglas

R. Lydon, Raytheon

N92
19424
UNCLAS

# A Framework for Assessing the Adequacy and Effectiveness of Software Development Methodologies*

*James D. Arthur and Richard E. Nance*

*Virginia Tech*

Index Terms: Software Development Methodologies, Procedural Evaluation, Evaluating Methodologies, Software Engineering Objectives, Software Engineering Principles, Induced Attributes, Indicators.

## 1. Introduction

Over the past two decades the software development process has changed dramatically. Early software development practices were guided by "seat-of-the-pants" programming styles. Recognizing maintenance difficulties associated with such styles, the software development community began to investigate and identify software engineering principles that could significantly enhance the maintainability and quality of a resulting product. Consequently, development techniques that exploited software engineering principles like abstraction [LISB75], information hiding [PARD72] and stepwise refinement [WIRN71] were formulated and integrated into many software development processes.

The subsequent demand for increasingly complex software systems, however, mandated the *coordinated* use of *complementary* principles, guided by an encompassing software development philosophy that recognized project level goals and objectives, i.e. a *methodological* approach to software development. Today, a myriad of tools, techniques, and development methodologies address the challenging task of producing high quality software. For example, SCR [HENK78] and

* Work supported by the U.S. Navy through the Systems Research Center under Basic Ordering Agreement N60921-83-G-A165 B003-4.

DARTS [GOMH84] are development methodologies that emphasize specific software engineering goals (reducing software development costs and facilitating the design of real-time systems, respectively). SREM [ALFM85] and SADT [ROSD77] are methodology based *environments*. Both focus on particular phases of the software life cycle and are supported by unified sets of complementary tools.

The steady proliferation of design methodologies, however, has not been without a price. In particular, users find increasing difficulty in choosing an appropriate methodological approach and recognizing reasonable expectations of a design or development methodology. Addressing this concern, the research described in this paper outlines a well-defined procedure for

- evaluating the *adequacy* of a software development methodology relative to project goals, and
- assessing the *effectiveness* of a methodology relative to the quality of the product produced.

The evaluation procedure is based on a substantiated set of linkages among accepted software engineering objectives, principles, and attributes. These linkages reflect an assessment perspective structured by the needs, process, and product sequence for system development, and enable a *comparative* scale for determining the adequacy and effectiveness of the supporting development methodology. The identification of code and documentation properties and the definition of metrics for these properties enables an accumulative determination of software engineering attributes, principles and objectives.

To provide a uniform basis for discussion, Section 2 outlines the role of a methodology in the software development process. Section 3 discusses the relationship of software engineering objectives, principles and attributes to the software development effort. Section 4 identifies the commonly accepted objectives, principles and attributes, defines the relationships among them, and then discusses how one evaluates a methodology based on the those relationships. Finally, Section 5 describes an application of the assessment procedure to two Navy software development methodologies.

## 2. What Constitutes a Methodology

Fundamental to the research presented in this paper is a common understanding of what constitutes a "methodology". Simply stated, a methodology is a *collection* of complementary methods, and a set of rules for applying them [FREP77]. More specifically, a methodology

(1) organizes and structures the tasks comprising the effort to achieve a global objective, establishing the relationships among tasks,

(2) defines methods for accomplishing individual tasks (within the framework of the global objective), and

(3) prescribes an order in which certain classes of decisions are made, and ways of making those decisions that lead to the overall desired objective.

In general, software development methodologies should be guided by accepted software engineering principles that, when applied to the defined process, achieve a desired goal. Based on this common understanding of what constitutes a methodology, the following sections present a procedural approach to evaluating the *adequacy* and *effectiveness* of software development methodologies.

## 3. The Role of Objectives, Principles, and Attributes in Software Development

The development of large, complex software systems is considered a *project* activity, involving several analysts and programmers and at least one manager. What then is the role of a methodology in this setting and how does it relate to objectives, principles and attributes? Figure 1 assists in providing an answer to this question.

In general terms, an *objective* is "something aimed at or striven for." More specific to the software development context, an objective pertains to a *project* desirable - a characteristic that can be definitively judged only at the completion of the project.

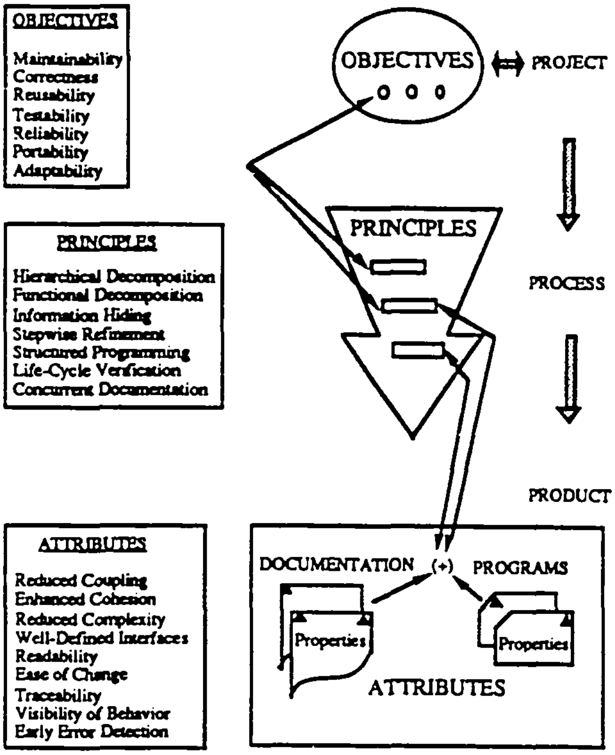


Figure 1

Illustration of the Relationships Among Objectives, Principles, Attributes in the Software Development Process

A software engineering *principle* describes an aspect of *how* the process of software development should be done. The process of software development, if it is to achieve the stipulated objectives, must be governed by these "rules of right conduct."

*Attributes* are the intangible characteristics of the product: the software produced by project personnel following the principles set forth by the methodology. Unlike objectives, which pertain only to the *total* project activity, attributes may be observed in one unit of the product and absent in another. The claim of presence or absence of an attribute is based on the recognition of *properties*, which contribute evidence supporting the claim. Properties are observable, and can be subjective as well as objective in nature.

Influenced by Fritz Bauer's original definition of software engineering [BAUF72] and reflecting the above description of software engineering objectives, principles and attributes, the rationale for the evaluation procedure described in this paper is founded on the philosophical argument that:

> The *raison d'etre* of any software development methodology is the achievement of one or more *objectives* through a *process* governed by defined *principles*. In turn, adherence to a process governed by those principles should result in a *product* (programs and documentation) that possesses *attributes* considered desirable and beneficial.

This philosophy, exemplified by Figure 1, is tempered by practical concerns:

(1) While a set of software engineering objectives can be identified, this set might not be complete, and additions or modification should be permitted.

(2) Objectives can be given different emphasis within a methodology or in applications of a methodology.

(3) Attributes of a large software product might be evident in one component yet missing in another.

## 4. A Framework for Evaluating Software Development Methodologies

A broad review of software engineering literature [BERG81, CHML90, GAFJ78, JACM75, PARD79, PARD72, SCOL78, WARJ76] leads to the identification of seven objectives commonly recognized in the numerous methodologies:

(1) Maintainability – the ease with which corrections can be made to respond to recognized inadequacies.

(2) Correctness – strict adherence to specified requirements,

(3) Reusability – the use of developed software in other applications,

(4) Testability – the ability to evaluate conformance with requirements,

(5) Reliability – the error-free performance of software over time,

(6) Portability - the ease in transferring software from one host system to another. and

(7) Adaptability – the ease with which software can accommodate to change.

The authors note that these definitions, as well as others presented in this section, are abridged: they are primarily intended to reflect a *working* understanding based on general literature usage.

Achievement of these objectives comes through the application of principles supported (encouraged, enforced) by a methodology. The principles enumerated below are extracted from the references cited above (and others) as mandatory in the creative process producing high quality programs and documentation.

(1) Abstraction - defining each program segment at a given level of refinement.

  (*a*) Hierarchical Decomposition – components defined in a top-down manner.

  (*b*) Functional Decomposition – components partitioned along functional boundaries.

(2) Information Hiding – insulating the internal details of component behavior.

(3) Stepwise Refinement – utilizing a convergent design.

(4) Structured Programming – using a restricted set of control constructs.

(5) Concurrent Documentation – management of supporting documents (system specifications, user manual, etc.) throughout the life cycle.

(6) Life Cycle Verification – verification of requirements throughout the design, development. and maintenance phases of the life cycle.

Employment of well-recognized principles should result in software products possessing attributes considered to be desirable and beneficial. A short definition of each attribute is given below.

(1) Cohesion - the binding of statements within a software component.

(2) Coupling - the interdependence among software components.

(3) Complexity - an abstract measure of work associated with a software component relative to human understanding and/or machine execution.

(4) Well-defined Interfaces - the definitional clarity and completeness of a shared boundary between a pair of components (hardware or software).

(5) Readability - the difficulty in understanding a software component (related to complexity).

(6) Ease of Change - the ease with which software accommodates enhancements or extensions.

(7) Traceability - the ease in retracing the complete history of a software component from its current status to its design inception.

(8) Visibility of Behavior - the provision of a review process for error checking.

(9) Early Error Detection - indication of faults in requirements specification and design prior to implementation.

The software development process, illustrated in Figure 1, depicts a *natural* relationship that links objectives to principles and principles to attributes. That is, one achieves the *objectives* of a software development methodology by applying fundamental *principles* which, in turn, induce particular code and documentation *attributes.* From a more detailed perspective, Figure 2 defines the *specific* set of linkages relating objectives to principles and principles to attributes. As described below, these linkages provide a framework for assessing both the *adequacy* of a methodology as well as its *effectiveness.*

### 4.1 Assessing the Adequacy of a Methodology

The enunciation of objectives should be the first step in the definition of a software development methodology. Closely following is the statement of principles that, employed properly, lead to the attainment of those objectives. In tur~, the application of those principles within a structuied software development process will yield a product that exhibits desirable attributes. The important

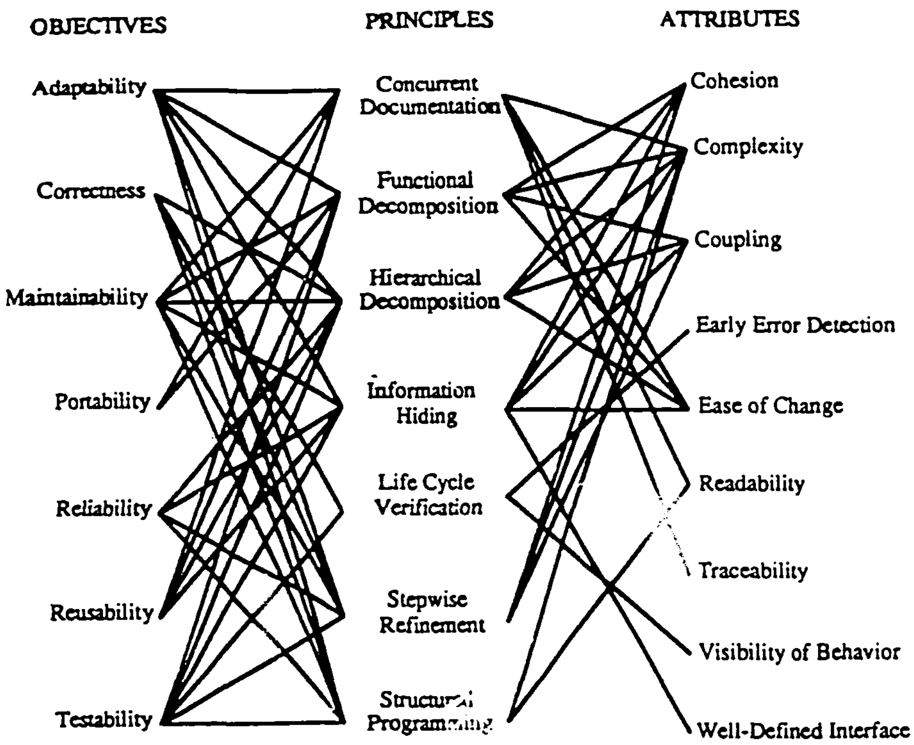


Figure 2

Linkages Among the Objectives, Principles and Attributes

correspondence between the objectives and principles and between the principles and attributes is shown in Figure 2; a literature confirmation of these relationships is discussed in [ARTJ87].

The adequacy of a software development methodology can be defined as its ability to achieve the software engineering objectives corresponding to those dictated by system needs and requirements. Intuitively, the adequacy of a methodology is assessed through a top-down evaluation scheme starting with an examination of stated methodological objectives relative to system needs and requirements. This step is then followed by a comparison of stated methodological principles and attributes with those deemed most appropriate. An examination of linkages defined by the evaluation procedure reveals the "most appropriate" set. Relative to the framework depicted by Figure 1 and the sets of linkages defined in Figure 2, an application of the evaluation procedure to the assessment of methodological *adequacy* is outlined below.

*Objectives cf the Methodology:* The identification of *objectives* and the relationships tying objectives to needs and requirements is usually accomplished by reading the descriptions of a software development methodology. Evaluation at this level is quite subjective; however, the absence of a clear statement of objectives for a methodology should trigger an alarm: Is the "methodology" only a tool or an incomplete set of tools without coherent structure? A methodology should not be faulted, however, for emphasizing certain objectives at the expense of others; such prioritization can be highly dependent on the application domain.

*Principles Defining the Process:* Based on the objectives emphasized by the methodology and the predefined set of linkages among objectives and principles, the next step in assessing the adequacy of a methodology is an investigation of the software development *process*. That is, given a stated set of methodological objectives, one asks: Are the *principles* supported by the methodology consistent with those deemed necessary (through linkage examination) to achieve the stated set of objectives? The presence of principles without corresponding objective(s) or vice versa should evoke concerns. Although this level of evaluation is inherently subjective, some analytical quality is introduced through the established objective/principle correspondence.

*Attributes of the Product:* The third step in the assessment process, formulating the set of *expected* product attributes, is based on the fact that principles govern the process by which a software product is produced. That is, a given set of principles should induce a consequent set of product attributes. Obviously, the expected set of product attributes should correspond to those desired by the software engineer, and to some extent, be implied or stated in the description of the software development methodology. More objectivity is introduced at this level because, although intangible. evidence of the attributes should be discernible in the product.

### 4.2 Assessing the Effectiveness of a Methodology

While a top-down evaluation process reveals deficiencies of a software development methodology, the *effectiveness* of a methodology is assessed through a bottom-up evaluation process. As the term implies, the effectiveness of a methodology is defined as the degree to which a methodology produces a desired result. In particular, the effectiveness of a methodology is reflected by a product's conformance to the software development process defined by that methodology. We note, however, that elements *independent* of the methodology can influence its effectiveness, e.g. an inadequate understanding and/or use of the methodology.

*The Existence of Product Attributes:* Assessing methodological effectiveness starts with an examination of the software product (code and documentation) for the presence or absence of attributes. Because attributes are intangible, subjective qualities, the current evaluation is based on defined properties that provide evidence as to the presence or absence of attributes. More specifically, the computation of metric values reflect the extent to which particular properties are observed. In turn, this information is used to synthesize the extant set of product attributes. Clearly, the set of attributes determined from a product evaluation should agree with those induced by the corresponding development methodology. Set mismatch can imply an inappropriate software development methodology, an inadequate understanding of the methodology, or perhaps, the failure of users to adhere to the principles advocated by the methodology.

*Implied Principles and Objectives:* Knowing which attributes are present in the product, and the extent to which they are assessed present, provides a basis for implying the use of software engineering principles in the software development process. The rationale for such a statement is based on the observation that a principle-to-attribute linkage conversely indicates an attribute-to-principle relationship. Implying principle usage must be tempered, however, because of the many-to-one relationships existing between attributes and principles. Similarly, using the established linkages among objectives and principles, one can speculate on the achievement of stated software engineering objectives.

In summary, the three levels of examination defined by top-down evaluation process establishes a procedure for determining how well a methodology can support perceived needs, requirements, and the software development process. Conversely, the bottom-up evaluation process reveals how

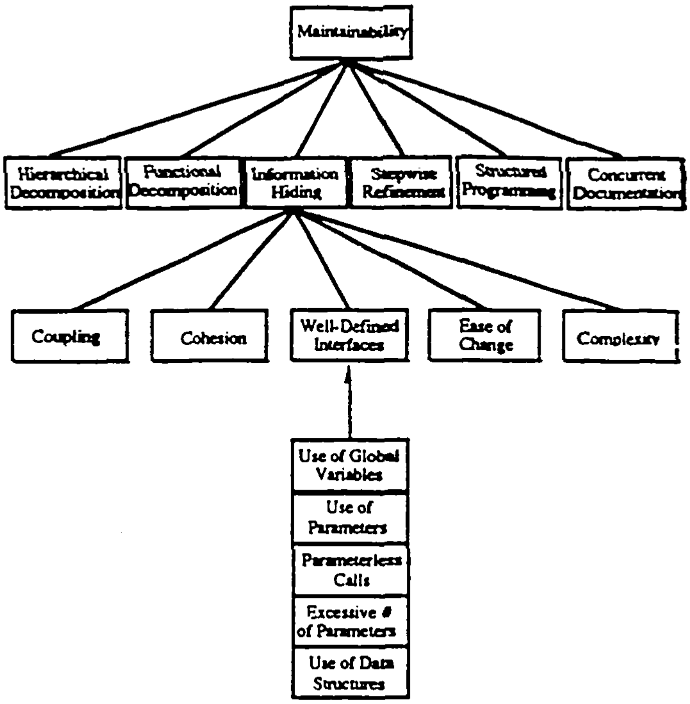


**Figure 3**

Illustration of the Evaluation Process

well the methodology is applied in the software development process through the use of *quantitative* measures to support an objective, *qualitative* assessment.

## 4.3. An Illustration of the Evaluation Process

To illustrate how the evaluation scheme can be applied, we refer the reader to Figure 3 while considering the single objective of *maintainability.* Formally, maintainability can be defined as the ease with which maintenance can be performed to a functional unit in accordance with prescribed requirements. Accepting maintainability as an objective mandates the inclusion of six principles (hierarchical decomposition, functional decomposition, information hiding, stepwise refinement. structured programming and concurrent documentation) contributing to the realization of that objective. That is, if a methodology emphasizes maintainability as an objective, then it should also stress the use of the six principles that are related to maintainability.

Expanding on one of those principles, information hiding, we note the five attributes (reduced coupling, enhanced cohesion, well-defined interfaces, ease of change, and low complexity) that should be evident in software developed using a process governed by the principle of information hiding. This set of expected attributes is then compared to the desired set for correspondence.

In general, a methodology should emphasize the same set of software engineering objectives derived from project level requirements. The methodology should correspondingly stress the set of principles linked to those objectives. Additionally, the expected set of product attributes (defined by the linkages among principles and attributes) should agree with the set deemed most desirable by the project manager. If the above conditions are met, then the candidate methodology is assumed to be adequate relative to project level, software engineering objectives.

On assessing the effectiveness of an methodology, let us first observe the relationship between a particular attribute and specific product characteristics. Referring again to Figure 3. and narrowing our attention to one of the attributes, well-defined interfaces, we identify one set of characteristics related to the well-defined interfaces attribute. These characteristics form the set of *observable* properties which contribute to the claim that a piece of software exhibits a well-defined interface. Although the properties shown are only a subset of those previously identified [ARTJ86], they represent *both* confirming and contrasting elements. For example, the use of global variables has a negative impact on well-defined interfaces. The use of structured data in parameter calls, however, has a positive impact.

Hence, to determine the effectiveness of a methodology one assesses the extent to which product attributes are present (or absent), and then propagate the results of that assessment through the sets of linkages defined by the evaluation procedure. As discussed by Kearney [KEAJ86], however. that assessment process must be predicated on validated metrics.

## 5. Application of the Evaluation Procedure

Based on the defined set of linkages among objectives, principles, and attributes, the operational specification of the evaluation procedure is guided by two fundamental axioms:

(1) the methodology description and project requirements provide standards, conventions, and guidelines that *describe how to produce* a product, and

(2) the project documentation, code, and code documentation *reflect how well* the development process prescribed by the methodology is followed.

As described below, an application of the evaluation procedure, guided by the above two axioms, illustrates the utility and intrinsic prower of the evaluation procedure in assessing the adequacy and effectiveness of a methodology. Provided in this illustration is a characterization of the components used in the evaluation process, an individual assessment of two methodology descriptions, an analysis of associated products, and a summary of the results. The authors note that a substantial part of the characterization and assessment process is embodied in the operational aspects of applying the evaluation procedure. Length restrictions, however, prevent their discussion. For information on the operational aspects the authors refer the interested reader to [ARTJ86, ARTJ87].

### 5.1 Data Sources

A joint investigation of two comparable Navy software development methodologies and respective products is detailed in [NANR85]. The investigation effort utilizes:

- Four software development methodology documents for

  (1) identifying the pronounced software engineering objectives, principles, and attributes, and

(2) assessing the adequacy of each methodology through the objective/principle/attribute linkages defined by the evaluation procedure, and

- Eight software system documents and 118 routines, comprising 8300 source lines of code, for

(1) determining the evident set of product attributes, and

(2) via the attribute/principle/objective linkages, empirically assessing the principles and objectives emphasized during product development.

The following section provides a summary of the results and illustrates the utility and versatility of the procedural approach to evaluating software development methodologies. For simplicity, we refer to the software systems as system A and system B (and methodology A, methodology B, respectively).

### 5.2 Analyzing the Methodological Description and Associated Product

The initial step in the evaluation process is to perform a "top-down" analysis of methodologies A and B, to reveal the set of *claimed* software engineering objectives, principles, and attributes. Because both methodologies have experienced evolutionary development, a *clear* statement of their respective methodological objectives is lacking. Nonetheless, as detailed in Figure 4, the documentation for methodology A does appear to stress the objectives of *reliability* and *correctness* supported by the principles of *structured programming, hierarchical decomposition,* and *functional decomposition.* Following the objective/principle relationships defined by the evaluation procedure, for each objective stressed in methodology A only three of the necessary six principles are emphasized. The implication is, that unless the principles of life-cycle verification, information hiding and stepwise refinement are implicitly assumed *and* utilized, correctness and reliability are compromised.

| | Methodology A | Methodology B |
|---|---|---|
| **Objectives** | | |
| Maintainability | | Yes |
| Correctness | Yes | |
| Reusability | | |
| Testability | | |
| Reliability | Yes | Yes |
| Portability | | |
| Adaptability | | Yes |
| **Principles** | | |
| Hierarchical Decomposition | Yes | |
| Functional Decomposition | Yes | |
| Information Hiding | | |
| Stepwise Refinement | | |
| Structured Programming | Yes | Yes |
| Concurrent Documentation | | Yes |
| Life Cycle Verification | | |
| **Attributes** | None | None |

**Figure 4**

Pronounced Objectives, Principles, and Attributes

Using metric values and properties, a corresponding "bottom-up" examination of product A provides some interesting results. The Kiviat graph displayed in Figure 5a illustrates the extent to which each attribute is *assessed* as present in the product. (Attribute ratings are restricted to an arbitrarily chosen 1-10 scale.) Note that (reduced) complexity attains the highest rating – 8.0 out of 10.0, closely followed by readability (7.4) and cohesion (6.8). Based on the three principles stressed in methodology A, the evaluation procedure predicts that (reduced) complexity, readability, and cohesion *should*, in fact, be among the product attributes.

In concert with the stated objectives and principles for methodology A, Figure 5b reveals that structured programming (7.7) is the prominent principle used in developing system A, followed by stepwise refinement (6.7), hierarchical decomposition (6.4), and functional decomposition (6.4). Figure 5c depicts the results of emphasizing these principles in terms of methodology objectives. In

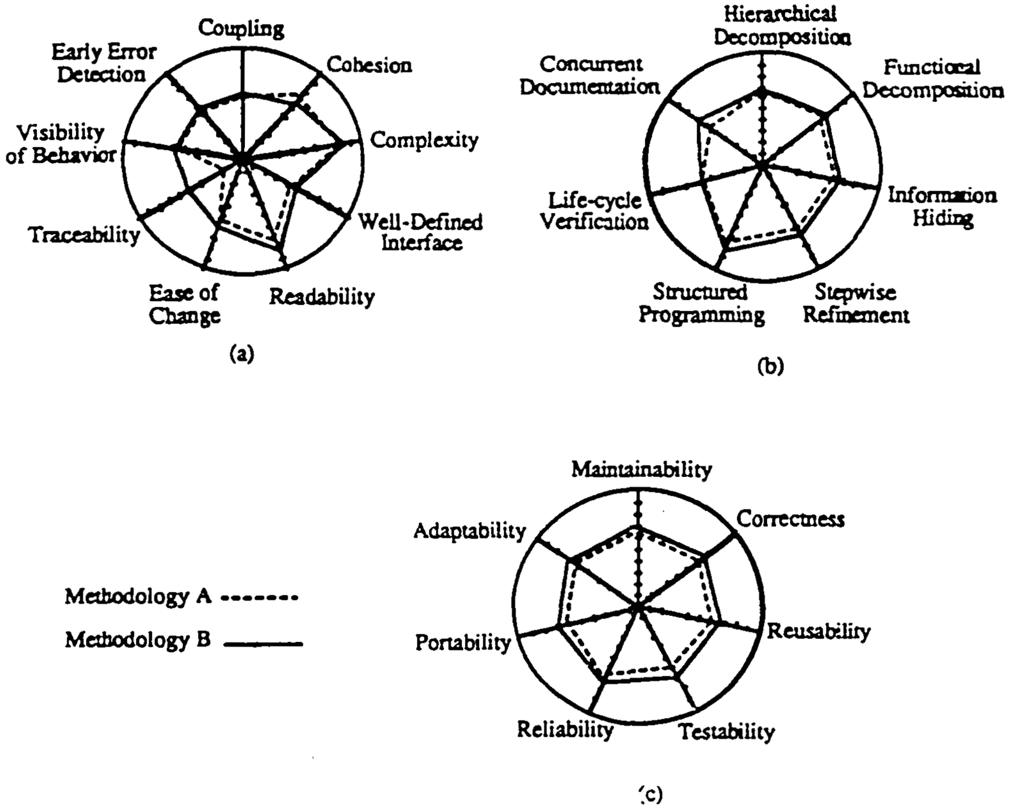


**Figure 5**

Detected Presence of Objectives, Principles, and Attributes

particular, reliability is rated as the major software development objective (6.7). Although correctness is also stressed by methodology A, ascertaining correctness necessitates life-cycle verification. This principle is neither emphasized by methodology A, nor evident in the software product. As illustrated by Figures 5a, 5b and 5c, other objectives and principles are given some emphasis during the software development process for system A. It is the authors' opinions, however, that because they are not explicitly stressed in methodology A, the associated product suffers.

For methodology B, the objectives enunciated in the documentation are *maintainability, adaptability*, and *reliability*. *Structured programming* and *concurrent documentation* are the emphasized principles. Like methodology A, however, a complete set of supporting principles are not stated.

*Hierarchical decomposition*, *functional decomposition*, and to some extent *information hiding* are implicitly assumed as underlying principles of methodology B. According to the linkages among objectives and principles, all of the above principles (both stated and assumed) are required to achieve the objectives explicitly stated in methodology B.

Subsequent analysis of product B and a "bottom-up" propagation of the results through the linkages defined by the evaluation procedure reveals structured programming as the most prominent principle (8.3), closely followed by concurrent documentation (7.0). Moreover, the evaluation also indicates that the implicitly assumed principles of methodology B are utilized – stepwise refinement. hierarchical decomposition, functional decomposition, and information hiding rate 6.9, 6.7, 6.7, and 6.3, respectively. Finally, the results imply that during the development of product B the objectives of maintainability, adaptability, and reliability are most emphasized. The above assessments are illustrated in Figures 5a, 5b, and 5c.

To summarize, the evaluation procedure reveals that both methodologies lack a clear statement of goals and objectives, as well as sufficient principles for achieving the objectives that are emphasized. Moreover, glaring deficiencies are apparent in both software development methodologies. That is, both fail to actively support the principle of information hiding and also have difficulties in incorporating the desirable attributes of traceability and well-defined interfaces in respective system products. In general, the evaluation procedure does accurately assesses the software engineering objectives, principles, and attributes *espoused* by methodologies A and B. Of particular significance, however, is that the objectives and principles *determined* to be "emphasized" during the product development process, yet not stated in the methodology documentation, are precisely those that are *implicitly assumed* important by the software engineers developing products A and B. A more detailed account of the evaluation can be found in [NANR85].

## 6. Conclusion

Tools, techniques, environments, and methodologies dominate the software engineering literature, but relatively little research in the evaluation of methodologies is evident. This work reports an initial attempt to develop a procedural approach to evaluating software development methodologies. Prominent in this approach are:

(1) an explication of role of a methodology in the software development process,

(2) the development of a procedure based on linkages among objectives, principles, and attributes, and

(3) the establishment of a basis for reduction of the subjective nature of the evaluation through the introduction of properties.

An application of the evaluation procedure to two Navy methodologies has provided consistent results that demonstrate the utility and versatility of the evaluation procedure [NANR85]. Current research efforts focus on the continued refinement of the evaluation procedure through

(a) the the identification and integration of product quality *indicators* reflective of attribute presence, and

(b) the *validation* of metrics supporting the measure of those indicators.

The consequent refinement of the evaluation procedure offers promise of a flexible approach that admits to change as the field of knowledge matures. In conclusion, the procedural approach presented in this paper represents a promising path toward the end goal of objectively evaluating software engineering methodologies.

# References

[ALFM85] Alford, M., "SREM at the age of Eight; The Distributed Computing Design System," *IEEE Computer*, Vol. 18, No. 4, April 1985, pp. 36-54.

[ARTJ86] Arthur, J.D., Nance, R.E. and Henry, S.M., "A Procedural Approach to Evaluating Software Development Methodologies: The Foundation," Technical Report TR-86-24, Virginia Tech, 1986.

[ARTJ87] Arthur, J.D. and Nance, R.E., "Developing an Automated Procedure for Evaluating Software Development Methodologies and Associated Products," Technical Report SRC-87-007, Systems Research Center, Virginia Tech, 1987.

[BAUF72] Bauer, F.L. "Software Engineering," *Information Processing 71*, North Holland Publishing Company, 1972.

[BERG81] Bergland, G.D. "A Guided Tour of Program Design Methodologies," *Computer*, Vol. 14, No. 10, October 1981, pp. 13-36.

[CHML90] Chmura, L.J., Norcio, A.f., and Wicinski, T.J., "Evaluating Software Design Processes by Analyzing Change Data Over Time," *IEEE Transactions on Software Engineering*, Vol. 16, No. 7, July 1990, pp. 729-740.

[FREP77] Freeman, P., "The Nature of Design," *A Tutorial on Software Design Techniques*. Second edition, IEEE Computer Society Press, 1977, pp. 29-36.

[GAFJ81] Gaffeney, J. E., "Metrics in Software Quality Assurance," *Proceedings of the National ACM Conference*, November 1981, pp. 126-130.

[GOMH84] Gomaa, H. "A Software Design Method for Real-Time Systems," *Communications of the ACM*, Vol. 27, No. 9, September 1984, pp. 938-949.

[HENK78] Heninger, K. L., J. W. Kallander, J. E. Shore, and D. L. Parnas, "Software Requirements for the A-7E Aircraft," NRL Memorandum Report 3876. Naval Research Laboratory, Washington, D. C., November, 1978.

[JACM75] Jackson, M., *Principles of Program Design*, London: Academic Press. 1975.

[KEAJ86] Kearney, J., *et. al.*, "Software Complexity Measurement," *Communications of the A.C.M.*, Vol.29, No. 11, November 1986, pp. 1044-1050. Monterey, CA, March 1987, pp. 238-252.

[LISB75] Liskov, B., Zilles, S., "Specification Techniques For Data Abstraction," *IEEE Transactions on Software Engineering* , Vol. SE-1, No. 1, March 1975, pp. 7-19.

[NANR85] Nance, R.E., Arthur, J.D. and Dandekar, A.V. "Evaluation of Software Development Methodologies," A Final Report of the Immediate Software Development Project, The Department of Computer Sciences, Virginia Tech, December 1985.

[PARD76] Parnas, D., "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, March 1976, pp. 1-9.

[PARD72] Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM* , Vol. 15, No. 5, May 1972, pp. 330-336.

[ROSD77] Ross, D., "Structured Analysis: A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January, 1977, pp. 16-34.

[SCOL78] Scott, L., "An Engineering Methodology for Presenting Software Functional Architecture," *Proceedings of the Third International Conference on Software Engineering* , NY, 1978, pp.222-229.

[WARJ76] Warnier, J. *Logical Construction of Programs*, 3rd edition, trans. B. Flanagan, NY: Van Nostrand Reinhold, 1976.

[WIRN71] Wirth, N., "Program Development by Stepwise Refinement," *Communications of the ACM*, Vol. 14, No.4, April, 1971, pp. 221-227.

# VIEWGRAPH MATERIALS

# FOR THE

# R. NANCE PRESENTATION

# A FRAMEWORK FOR ASSESSING THE ADEQUACY AND EFFECTIVENESS OF SOFTWARE DEVELOPMENT METHODOLOGIES

A Presentation to the
Fifteenth Annual Software Engineering Workshop

*Richard E. Nance*
*James D. Arthur*

Systems Research Center
and
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia

28 November 1990

# THE ORIGIN

## Immediate Software Development Issues
## for
## Embedded Systems Applications
## in Surface Combatants
## (25 March - 15 September 1985)

Issue: Multiple Software Development Methodologies

(a) *Review* two software development "methodologies" (A and B)

(b) *Compare* and *evaluate* A and B

(c) *Assess* the costs and benefits of:

- Continuing witn multiple software development methodologies
- Using only one software development methodology
- Transitioning to an alternate software development methodology

| | | |
|---|---|---|
| A | ==> | B |
| B | ==> | A |
| A/B | ==> | ??? |

# OUTLINE

Evaluation Approach

- Objectives, Principles, Attributes Framework

Development of an Evaluation Procedure

- Software Engineering and Software Development
- A Structured Evaluation Procedure
- Data Sources

Application of the Evaluation Procedure

- Summary of Sample Data
- Illustration of Procedure Application

Summary of Results

Future Work

# EVALUATION APPROACH

1. Develop an Evaluation Procedure

   - What is a methodology?

   - How can they be compared?

2. Apply the Evaluation Procedure

   - In consonance with our Navy sponsor, and with

   - Contributions from software development sites and oversight agencies.

# ON METHODOLOGIES

What is a methodology?

A methodology is a collection of complementary methods, and a set of rules for applying them. More specifically, a methodology

(1) organizes and structures tasks comprising the effort to achieve a global objective, establishing the relationships among tasks,

(2) defines methods for accomplishing individual tasks (within the context of the global objective), and

(3) prescribes an order in which certain classes of decisions are made, and ways of making those decisions that lead to the desired objective.

# RATIONALE FOR THE EVALUATION PROCEDURE

A set of objectives can be identified that include those postulated by any software engineering methodology. A methodology defines those principles that characterize a proper and appropriate development process. Adherence to a process governed by these principles should result in a product (programs and documentation) that possesses attributes considered desirable and beneficial.

Philosophy tempered by practical concerns:

(1) Sets of objectives, principles, attributes are open.

(2) Prioritization of objectives recognized.

(3) Components of large software system vary – attribute sampiing.

(4) Flexible application of evaluation procedure – consonant with project objectives.

# FRAMEWORK FOR SOFTWARE DEVELOPMENT

OBJECTIVES

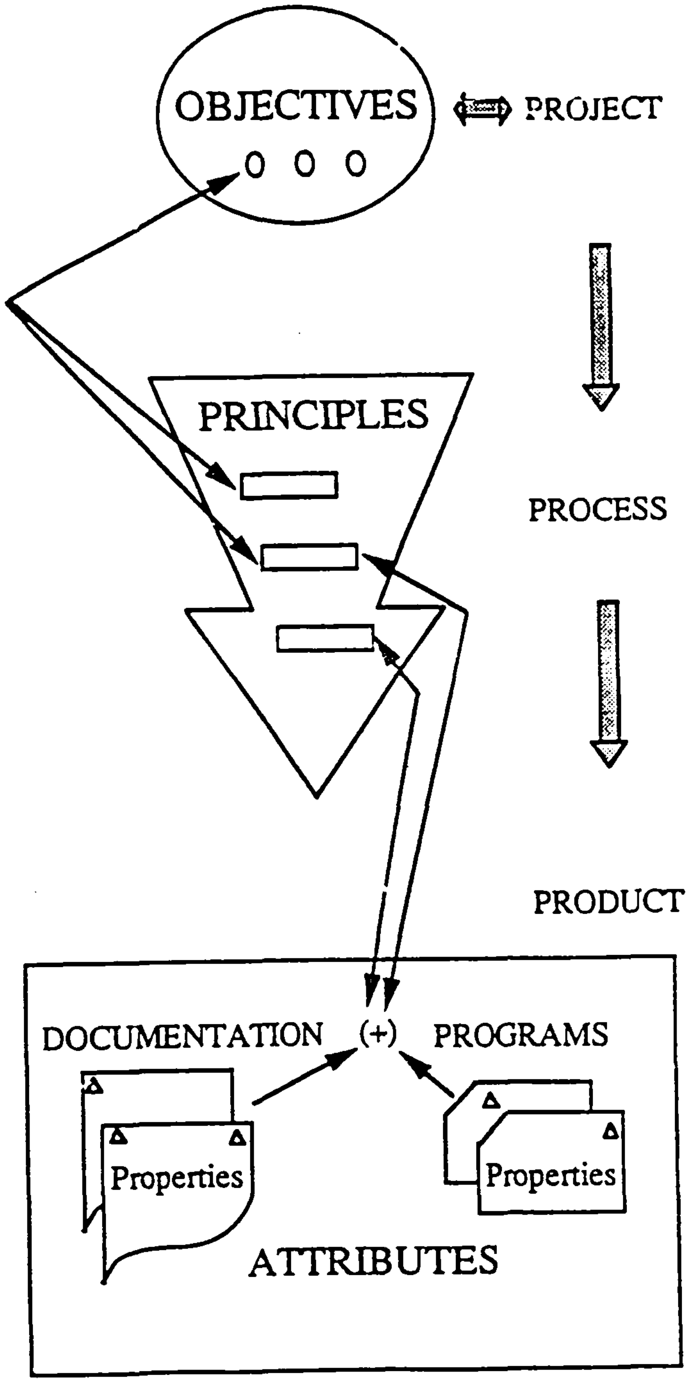
Maintainability
Correctness
Reusability
Testability
Reliability
Portability
Adaptability

PRINCIPLES

Hierarchical Decomposition
Functional Decomposition
Information Hiding
Stepwise Refinement
Structured Programming
Life-Cycle Verification
Concurrent Documentation

ATTRIBUTES

Reduced Coupling
Enhanced Cohesion
Reduced Complexity
Well-Defined Interfaces
Readability
Ease of Change
Traceability
Visibility of Behavior
Early Error Detection



# PROCEDURE DEVELOPMENT

1. Identify Objectives

   - What qualities are desirable?

2. Define Principles

   - How are desirable qualities obtained?

3. Link Principles to Objectives

   - Which principles contribute to each objective?

4. Define Resulting Attributes

   - Use of a principle induces what desirable attributes?

5. Define Properties Associated with Attributes

   - What properties give evidence of attribute presence or absence?

   - How to measure properties?

# PRIMARY SOFTWARE ENGINEERING OBJECTIVES

(1) **Adaptability** - the ease with which software can accommodate to changing requirements

(2) **Correctness** - strict adherence to specifications

(3) **Maintainability** - the ease with which corrections can be made to respond to recognized inadequacies

(4) **Portability** - the ease in transferring software to another host environment

(5) **Reliability** - the error-free behavior of software over time

(6) **Reusability** - the use of developed software in other applications

(7) **Testability** - the ability to evaluate conformance with specifications

# PRIMARY SOFTWARE ENGINEERING PRINCIPLES

(1) **Abstraction** - defining each program segment at a given level of refinement

(a) *Hierarchical Decomposition* - components defined in a top-down manner

(b) *Functional Decomposition* - components partitioned along functional boundaries

(2) **Concurrent Documentation** - management of supporting documents (system specifications, user manuals, etc) throughout the life cycle

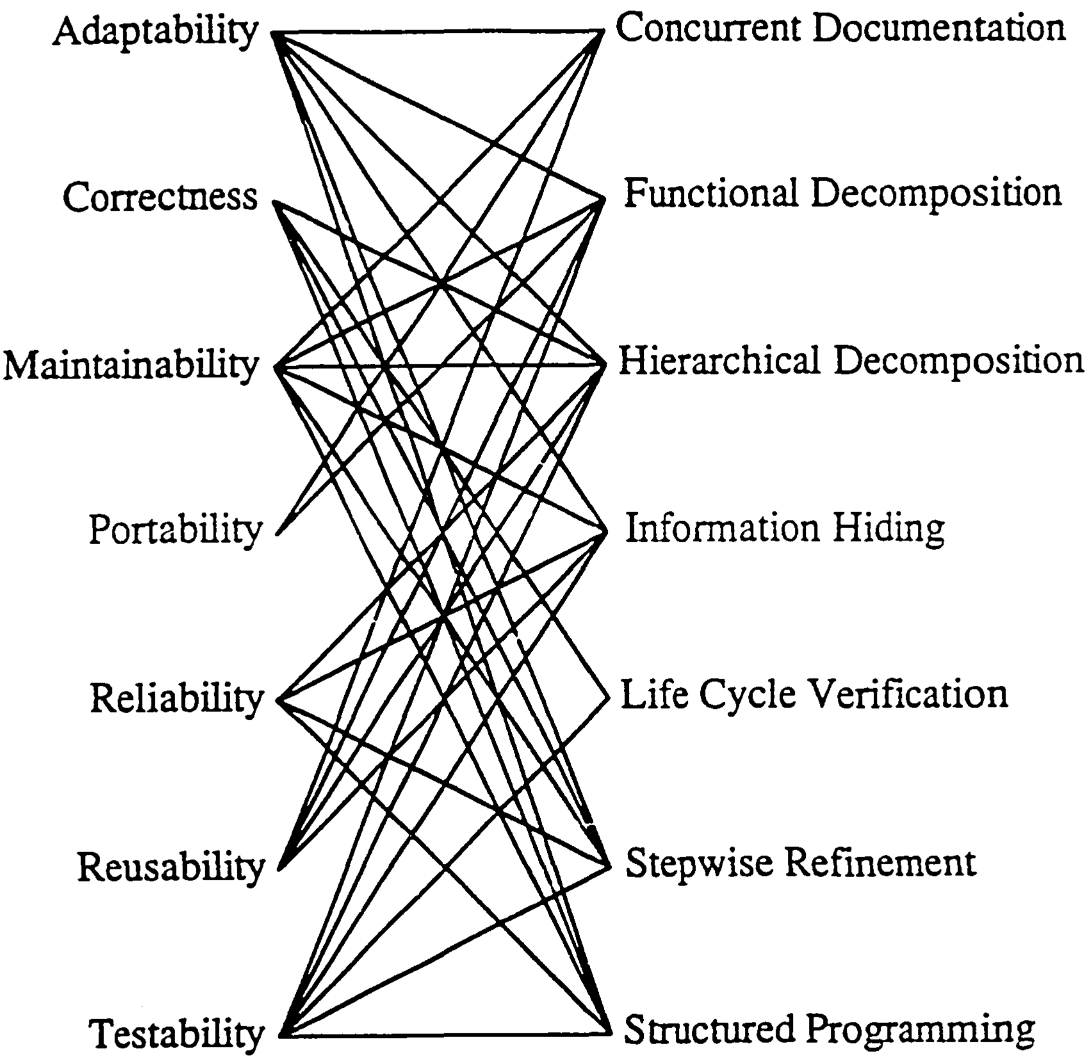
(3) **Information Hiding** - insulating the internal details of component behavior

(4) **Life Cycle Verification** - verification of requirements throughout the design, development, and maintenance phases of the life cycle

(5) **Stepwise Refinement** - utilizing convergent design

(6) **Structured Programming** - using a restricted set of program control constructs

# OBJECTIVES / PRINCIPLES LINKAGES



Adaptability

Correctness

Maintainability

Portability

Reliability

Reusability

Testability

Concurrent Documentation

Functional Decomposition

Hierarchical Decomposition

Information Hiding

Life Cycle Verification

Stepwise Refinement

Structured Programming

# PRIMARY SOFTWARE ENGINEERING ATTRIBUTES

(1) **Cohesion** - The binding of statements within a software component

(2) **Complexity** - an abstract measure of work associated with a software component

(3) **Coupling** - the interdependence among software components

(4) **Early Error Detection** - indication of faults in requirements, specification and design prior to implementation

(5) **Ease of Change** - software that accommodates enhancements or extensions

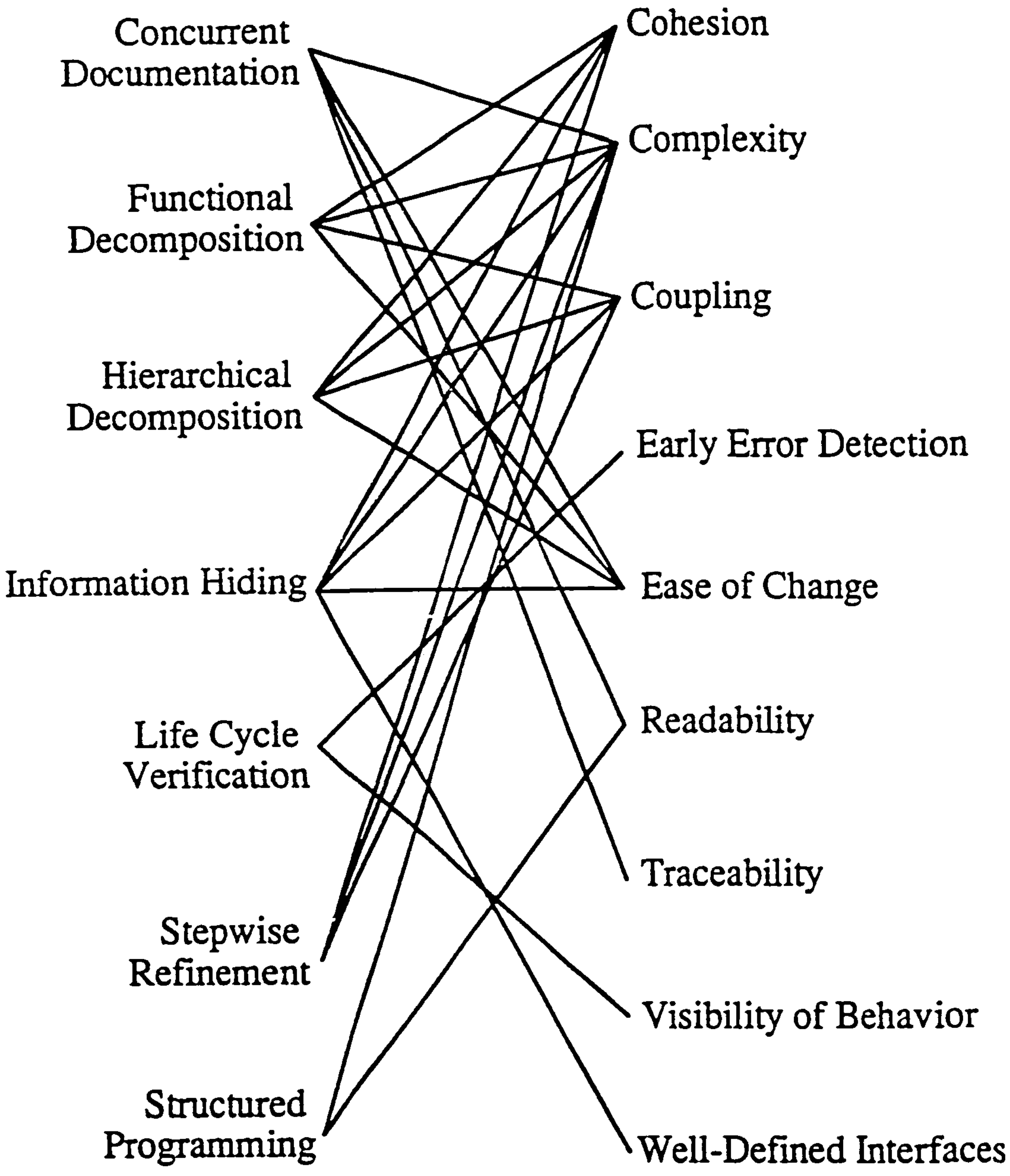
(6) **Readability** - the difficulty in understanding a software component

(7) **Traceability** - the ease in retracing the complete history of a software component from its current status to its design

(8) **Visibility of Behavior** - the provision of a review process for error checking

(9) **Well-Defined Interfaces** - the definitional clarity and completeness of a shared boundary between software and/or hardware (software/software, software/hardware)

# PRINCIPLES / ATTRIBUTES LINKAGES



Concurrent Documentation

Functional Decomposition

Hierarchical Decomposition

Information Hiding

Life Cycle Verification

Stepwise Refinement

Structured Programming

Cohesion

Complexity

Coupling

Early Error Detection

Ease of Change

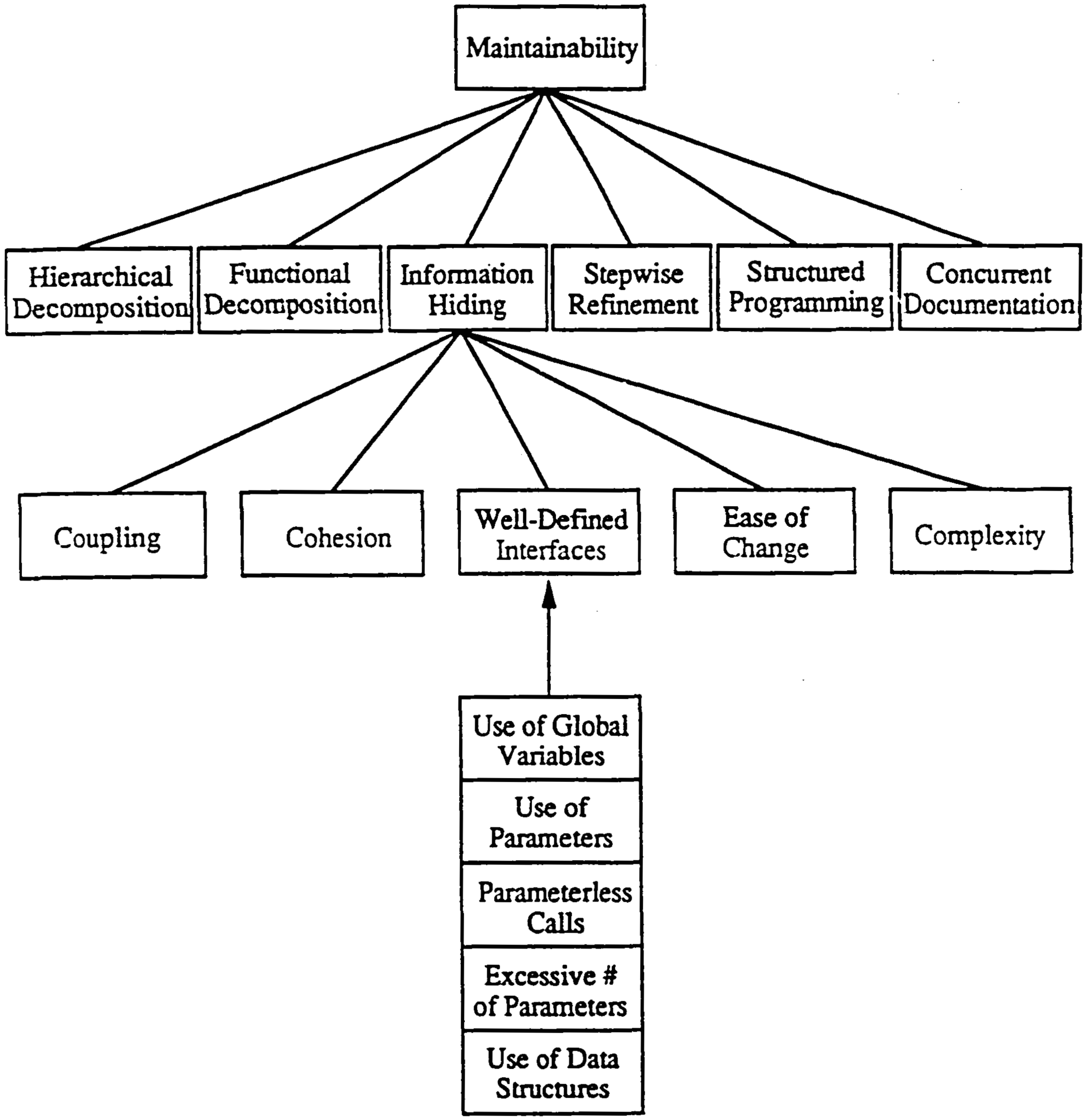
Readability

Traceability

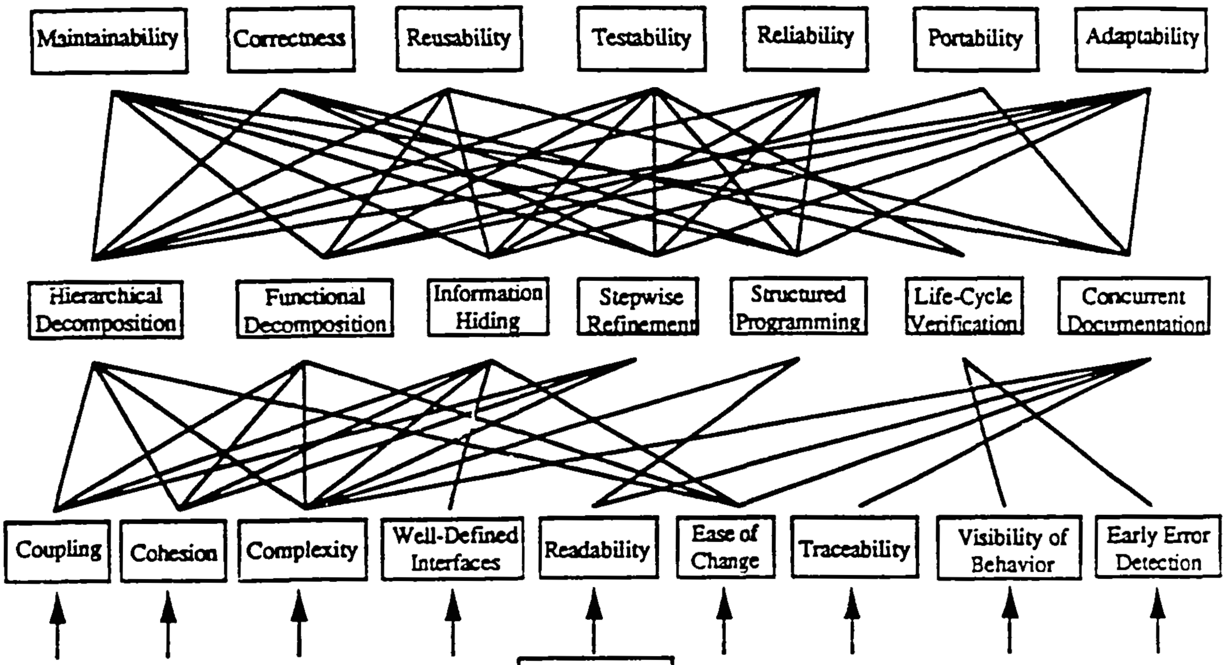
Visibility of Behavior

Well-Defined Interfaces

# ILLUSTRATION OF THE EVALUATION PROCEDURE



# SETS OF DEFINED LINKAGES



| | | | |
|---|---|---|---|
| | | Control Structures | (+) |
| | | GOTO's | (-) |
| (+) | Special Characters | Code Indentation | (+) |
| (+) | Symbolic Constants | # Block Comments | (+) |
| (+) | Documentation Readability | Meaningful Names | (+) |
| (+) | Completion & Accuracy of Documentation | # Single Line Comments 'In Line' | (-) |
| (-) | Embedded Alternate Language | Correct Grammar & Spelling | (+) |
| (+) | Short Modules | Consistent Comments | (+) |

# THE OPA FRAMEWORK FOR EVALUATION: SUMMARY

Fundamental to the evaluation procedure are several sets of linkages:

| Linkages | Defined | Substantiated |
|---|---|---|
| • Objectives / Principles | (33) | (33) |
| • Principles / Attributes | (24) | (24) |
| • Attributes / Properties | (125) | (114) |
| 66 Automatable | | |

Assessing the *adequacy* of a methodology is achieved through a "top-down" evaluation process.

Assessing the *effectiveness* of a methodology is achieved through a "bottom-up" evaluation process.

# APPLICATION OF THE PROCEDURE: SUMMARY OF SAMPLE DATA

Documents (Primary)

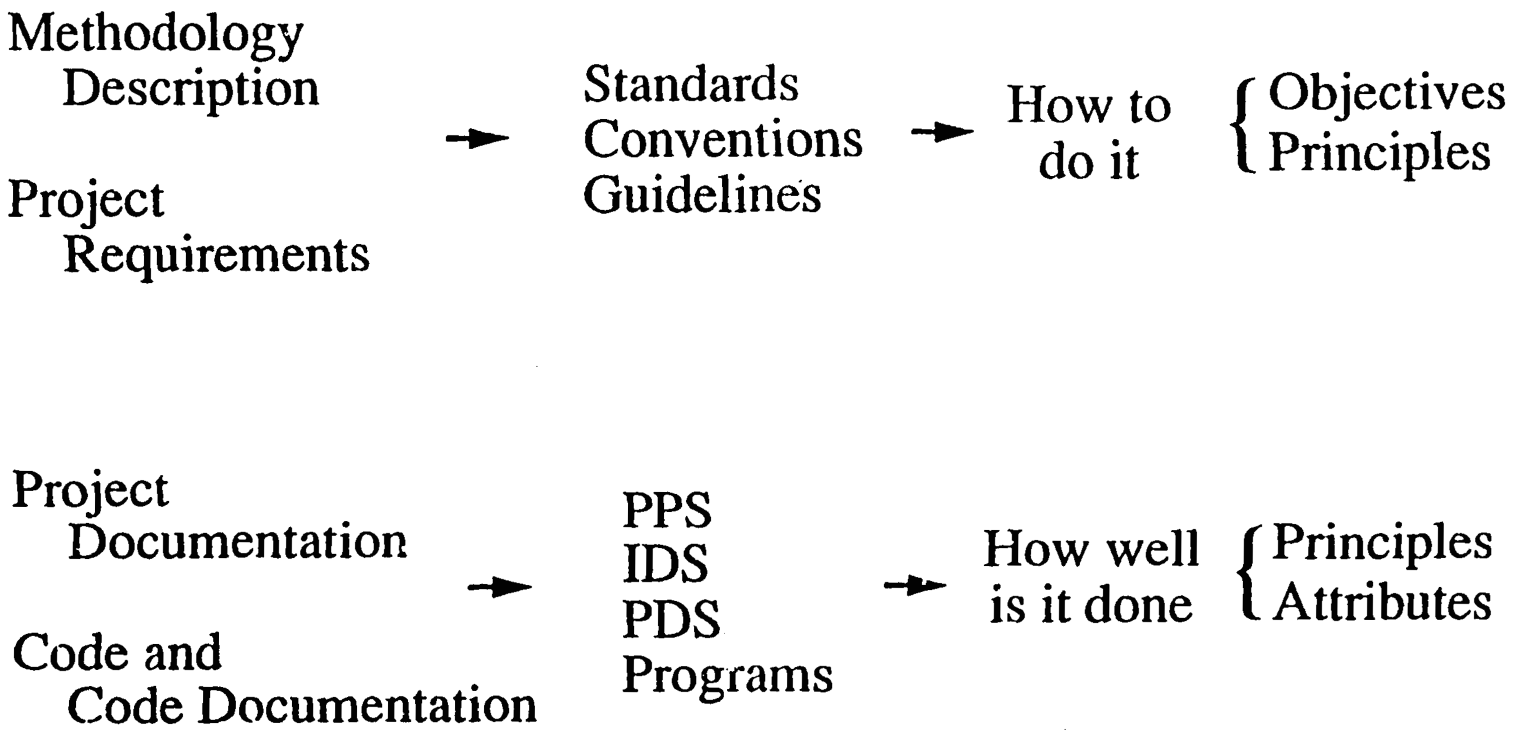
A: The Combat System Development Plan
The Computer Programming Manual
The Program Development Manual
Six Numbered Documents (PDS, IDS)

B: Functional Description Document
Two Numbered Documents (PDS, IDS)

Source Code:

| | | |
|---|---|---|
| A: | Routines = 17 | SLOCS = 1170 |
| | SysProcs = 2 | SLOCS = 1370 |
| B: | Routines = 99 | SLOCS = 5729 |

# DATA SOURCES AND IMPLICATIONS



Methodology Description

Project Requirements

→ Standards Conventions Guidelines → How to do it { Objectives Principles

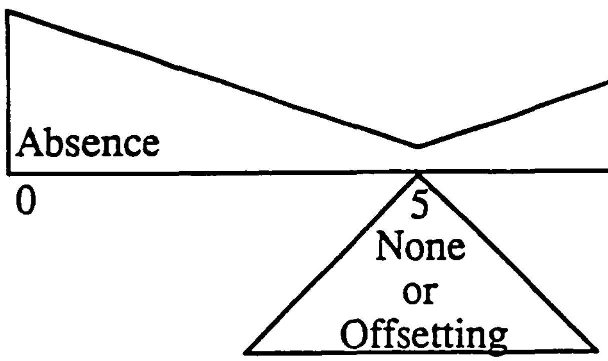
Project Documentation

Code and Code Documentation

→ PPS IDS PDS Programs → How well is it done { Principles Attributes

# AN ACCUMULATION OF EVIDENCE

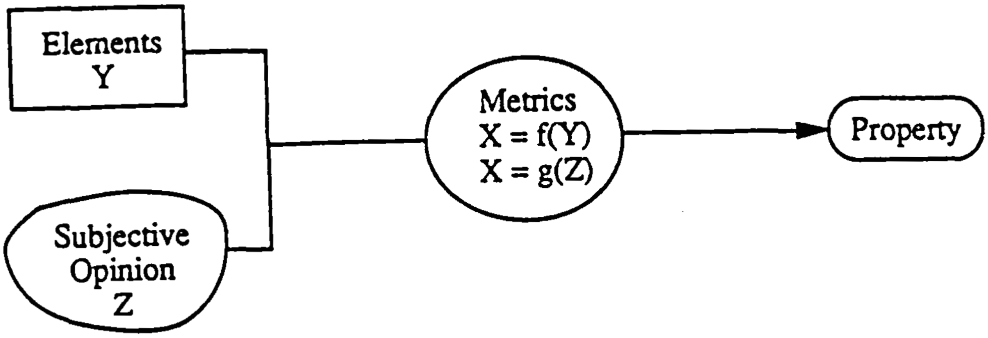
"Demonstrating that software possesses a desired attribute (or does not) is not a proof exercise; rather, it resembles an exercise in civil litigation in that evidence is gathered to support both contentions (the presence or absence) and weighed on the scales of comparative judgement."



Measurement Scale

# ELEMENTS, METRICS AND PROPERTIES

- Relationship



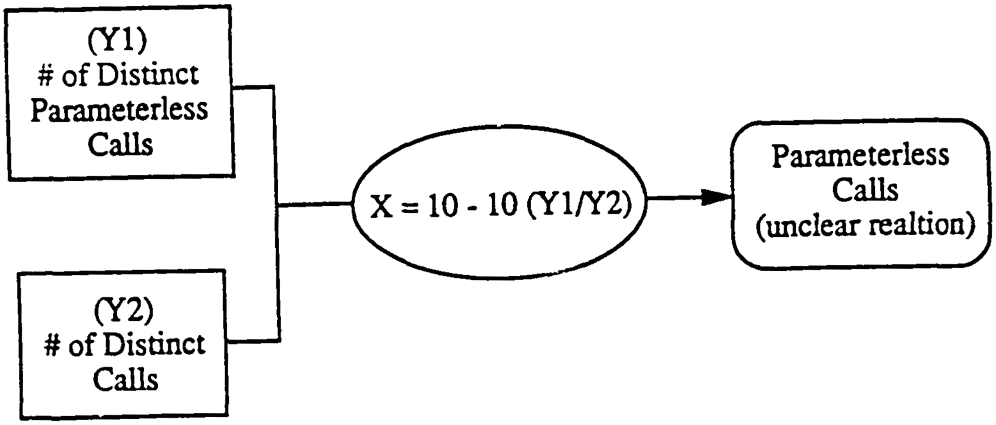
Elements
Y

Subjective
Opinion
Z

Metrics
X = f(Y)
X = g(Z)

Property

- Code Example



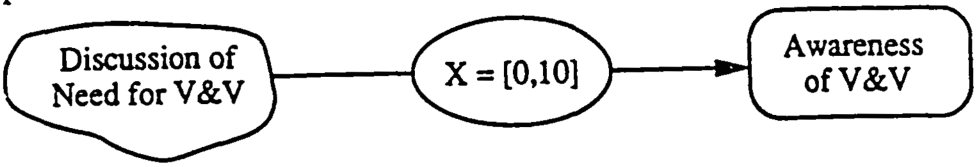
(Y1)
# of Distinct
Parameterless
Calls

(Y2)
# of Distinct
Calls

X = 10 - 10 (Y1/Y2)

Parameterless
Calls
(unclear realtion)

- Documentation
  Example



Discussion of
Need for V&V

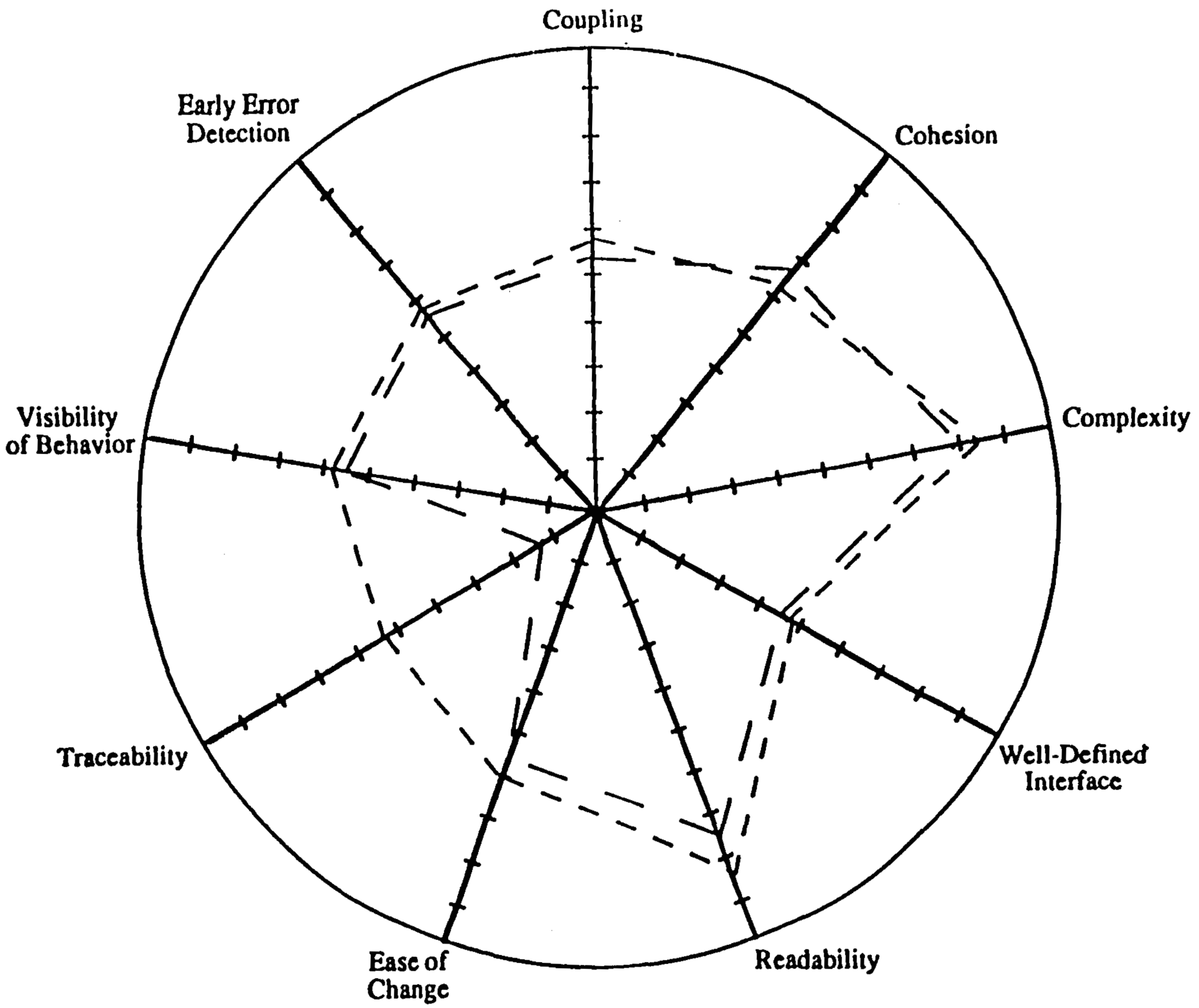
X = [0,10]

Awareness
of V&V

# ASSESSING "METHODOLOGICAL" EFFECTIVENESS (ATTRIBUTES)

| | A | B |
|---|---|---|
| Coupling | 5.4 | 5.9 |
| Cohesion | 6.8 | 6.4 |
| Complexity | 8.0 | 8.4 |
| Well-Defined Interfaces | 4.7 | 4.8 |
| Readability | 7.4 | 8.2 |
| Ease of Change | 5.6 | 6.0 |
| Visibility of Behavior | 5.6 | 5.8 |
| Early Error Detection | 5.6 | 5.8 |
| Traceability | 1.2 | 5.3 |

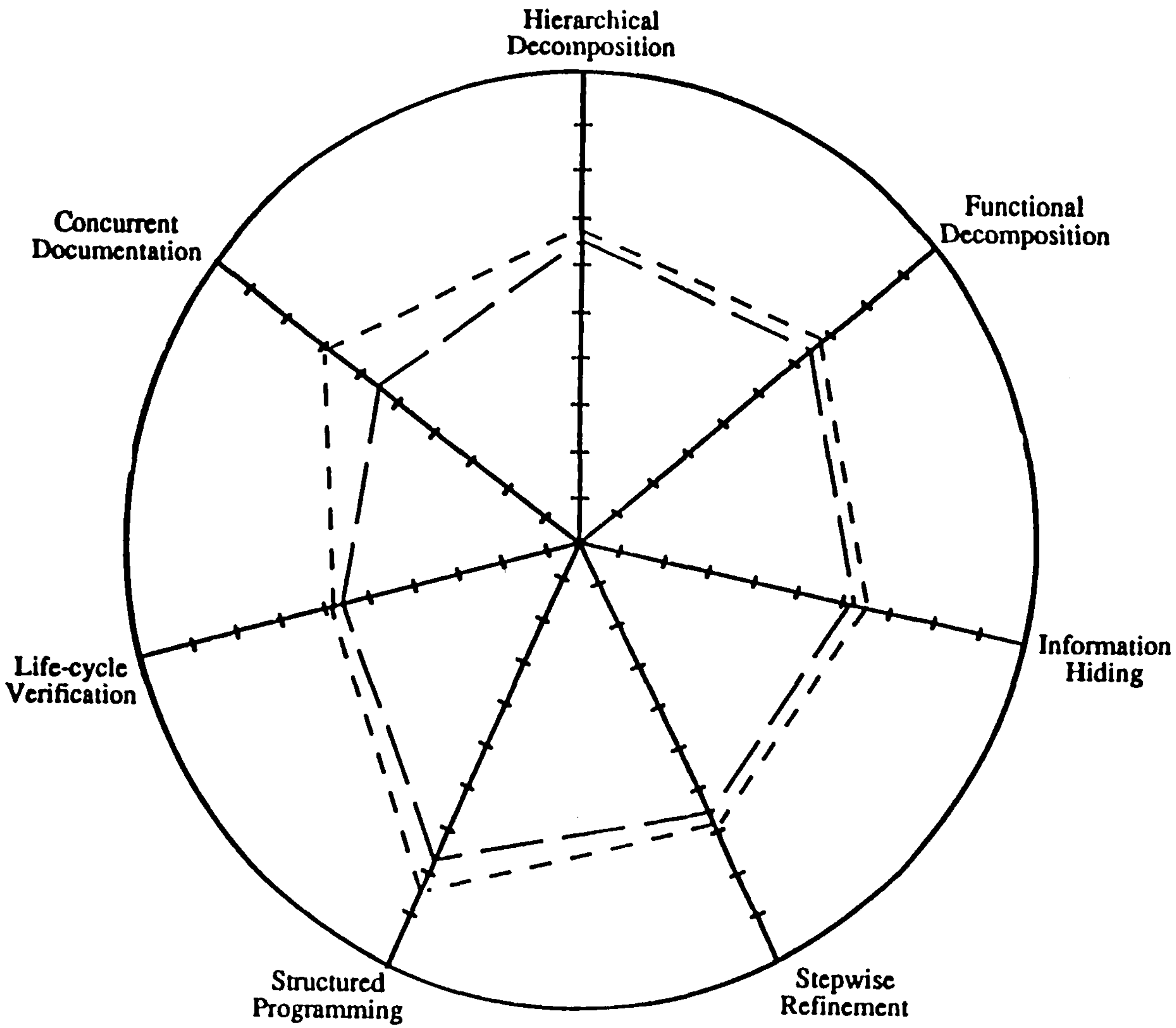
Both have difficulty with Traceability and Well-Defined Interfaces



KIVIAT GRAPH FOR ATTRIBUTES

Methodology A — — — — —

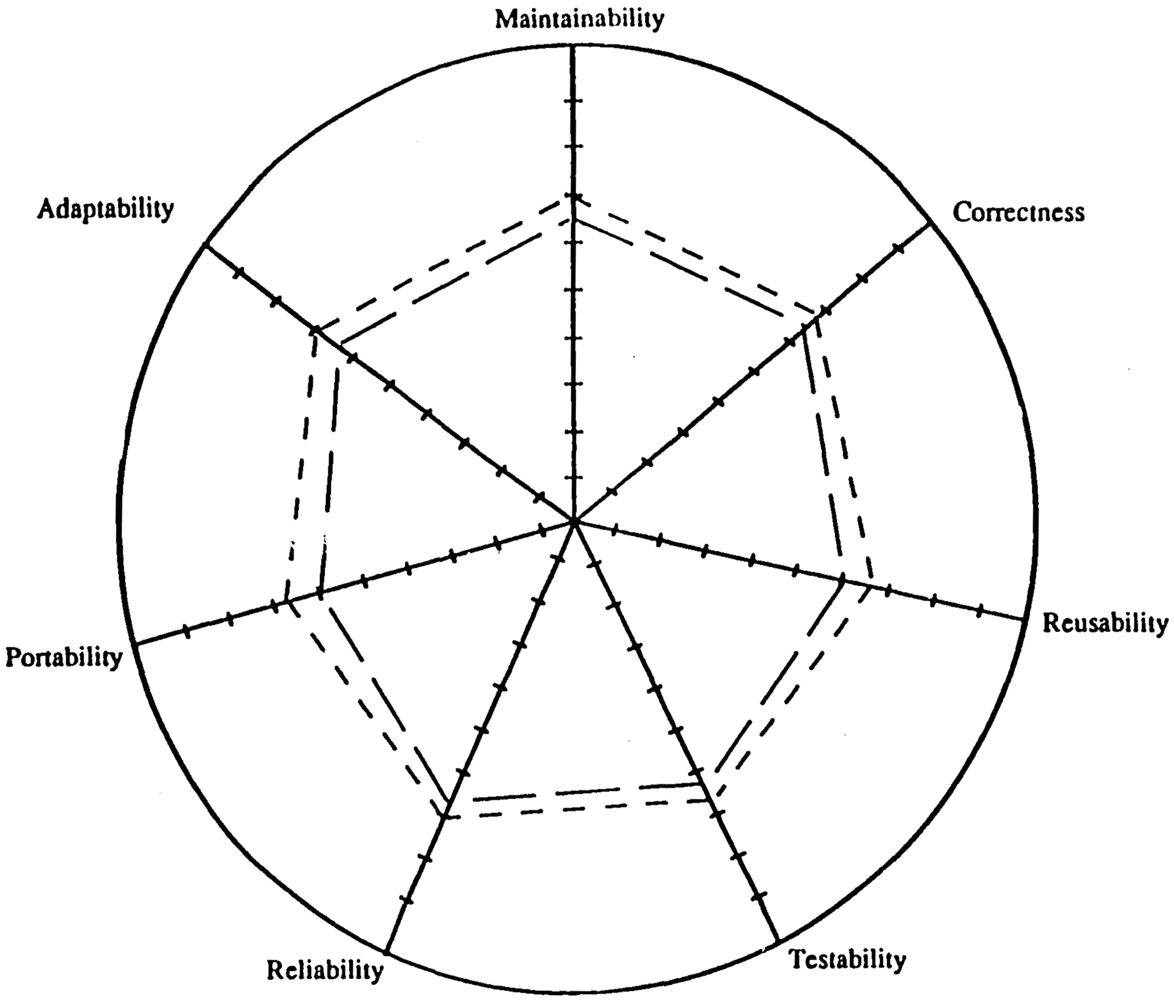
Methodology B - - - - - - - - -



KIVIAT GRAPH FOR PRINCIPLES

Methodology A — — — —

Methodology B - - - - - - -



KIVIAT GRAPH FOR OBJECTIVES

Methodology A — — — —

Methodology B - - - - - - -

# RESULTS OF PROCEDURE APPLICATION

## Assessing "Methodological" Adequacy

A: Stresses Objectives of Reliability and Correctness
Emphasizes Principle of Structured Programming

Methodology A was (and is) an "evolving methodology"

B: Stresses Objectives of Maintainability, Adaptability, Reliability, and Correctness
Emphasizes Principles of Modular Decomposition, Structured Programing and Concurrent Documentation

At the Objectives level, both “methodologies” support stated project objectives.

At the Principles level, both “methodologies” lack the enunciation of proper Principle usage.

No reference to desired Attributes is found

# FUTURE RESEARCH

## Applying the Evaluation Procedure to Software Quality Assurance

Predicting and/or assessing software quality necessitates a

- *Systematic* approach to
- *Assessing* product (or process) conformance with
- *Acceptance criteria* (standards and guidelines)

The Evaluation Procedure

- Currently supports a well-defined, systematic approach for evaluating software products, and
- Provides a rigorous framework for
  - Relating acceptance criteria based on attributes to software engineering principles and objectives, and
  - Defining acceptance levels based on measures reflecting the achievement of objectives, adherence to principles and realization of attributes.

N92
19425
UNCLAS

# A Method for Tailoring the Information Content of a Software Process Model

Dr. Sharon Perkins
University of Houston, Clear Lake

Mark B. Arend
839 Walbrook Dr.
Houston, TX 77062

## ABSTRACT

This paper will define the framework of a general method for selecting a necessary and sufficient subset of a general software life cycle's information products, to support new software development projects. Procedures for characterizing problem domains in general and mapping to a tailored set of life cycle processes and products will be given. An overview of the method is shown using the following steps:

1. During the problem concept definition phase, *perform standardized interviews and dialogs* between developer and user, and between developer and customer.
2. *Generate a quality needs profile* of the software to be developed, based on information gathered in step 1.
3. *Translate the quality needs profile* into a profile of quality criteria that must be met by the software to satisfy the quality needs.
4. *Map the quality criteria* to a set of accepted processes and products for achieving each criterion.
5. *Select the information products* which match or support the accepted processes and product of step 4.
6. *Select the design methodology* which produces the information products selected in step 5.

This paper will address every step, but will not attempt to generate a full-up methodology. A few of the more popular process models and design methodologies known today will be examined for their information content.

## TERMINOLOGY NOTES

The terms "software process model" and "life cycle" will be used interchangeably. The term "user" will always mean "customer and user".

## INTRODUCTION

The complete set of information products defined for common software process models and development methodologies is often too large for certain development efforts. In many cases, a subset of information products and the activities that produce them will suffice to administer the development of a software product. The act of selecting appropriate information products and activities to support the development effort is called "tailoring" the life cycle or development methodology. This tailoring process is currently an ad hoc method performed by managers and developers, in early meetings with the customer and user, as they begin to define some sort of Software Management or Development Plan. This paper explores a more formalized tailoring method to assist in the definition of such plans. It is hoped that such a formalization will both speed the process and help ensure the selection of a necessary and sufficient subset of information products (and by implication, the activities which produce them).

The cornerstone of this tailoring method uses Software Quality Assurance (SQA) techniques. Traditionally, SQA has dealt with the detection and prevention of defective software. New ideas in the field of SQA are concentrating on beginning the function much earlier in the life cycle, as early as problem concept and initial requirements definition It is hoped that SQA principles will assist the user and developer in creating complete, consistent and testable requirements. This assistance offers guidelines up front when we're scrambling to put some sensible words on paper

This paper developed under NASA cooperative agreement NCC-9-16, through the RICIS office of the University of Houston, Clear Lake. April 1990.

I believe that two quotes [5], [21] can illustrate the idea of "engineering in" quality to a software product.

> You can't achieve Quality...
> unless you specify it!

> Quality must be defined as conformance
> to requirements, not as "goodness"

## USING SQA TECHNIQUES TO SPECIFY QUALITY

### Quality Factors

This is a common SQA term. Quality Factors are characteristics which a software product exhibits that reflect the degree of acceptability of the product to the user. Since we're moving SQA up front, we'll restate this: Quality Factors are characteristics which the user *requires* the software to exhibit in order to reflect *the best possible* degree of acceptability.

Table 1 shows a list of Quality Factors which has been coming into general use for some time [21]. It was first proposed at the Rome Air Development Center (RADC) in 1977. I show a slightly expanded list, as it has evolved somewhat since then [5].

There are more detailed meanings of the quality factors which guide the user & developer in determining how important each factor is for their application.

Not every project requires all quality factors, which is good, because some quality factors are at conflicting purpose. Shown below is a list of factors whose characteristics cause conflicts of definition.

| *Quality Factor Conflict* | *Explanation of conflict* |
|---|---|
| Efficiency vs. Integrity | Overhead required to control access negates efficiency. |
| Efficiency vs. Usability | Overhead required to ease operations negates efficiency. |
| Efficiency vs. Maintainability | Optimized code negates maintainability. Modularization, instrumentation and well commented high-level code increases overhead. |
| Efficiency vs. Testability | Optimized code negates testability. |
| Efficiency vs. Portability | Optimized code is dependent on host processor services. |
| Efficiency vs. Flexibility | Overhead required to support flexibility negates efficiency. |
| *Efficiency vs. Reusability* | *Overhead required to support reusability negates efficiency.* |
| Efficiency vs Interoperability | Overhead required to support interoperability negates efficiency. |
| Integrity vs. Flexibility | Flexibility requires general and flexible data structures, increasing data security problems. |
| Integrity vs. Reusability | Generality required by reusable software introduces protection problems. |
| Integrity vs. Interoperability | Coupled systems allow more avenues of access. |
| Reusability vs. Reliability | Generality required by reusable software increases difficulty of providing error tolerance (anomaly management) and accuracy. |

The conflicts shown do not mean that the two factors are in strict mutual exclusion — extra effort may be expended to address the difficulties of specifying factors in conflict. Note that efficiency tends to conflict with many other factors. This is due to the tradeoff with the additional overhead required to satisfy other quality factors that does not necessarily apply to the algorithm's basic function. Efficiency issues may also be resolved by judicious hardware

| *Quality Factor* | *Meaning of factor in context of user needs for software product* |
|---|---|
| Correctness | Conformance of software design and implementation to stated requirements. |
| Efficiency | Economy of resources needed to provide the required functionality. |
| Expandability | Ease of maintaining the software to meet new functional or performance requirements. |
| Flexibility | Ease of maintaining the software to work in environments other than originally required. |
| Integrity | Security against unauthorized access to programs and data. |
| Interoperability | Ease of coupling the software with software in other systems or applications. |
| Maintainability | Ease of finding and fixing errors. |
| Manageability | Ease of administrating development, maintenance and operation of the software. |
| Portability | Ease of maintaining the software to execute on a processor or operating system other than that originally required. |
| Usability | Ease of learning & using the software, and of preparing input & interpreting output. |
| Reliability | The rate of failures in the software that render it unusable. |
| Reusability | Suitability of software modules for use in other applications. |
| Safety | Protection against loss of life or liability or damage to property. |
| Survivability | Continuity of reliable execution in the presence of a system failure. |
| Verifiability (testability) | Ease of verification of functionality against requirements. |

Table 1 – Quality Factors

selection. Note that there is also a reverse–matrix of quality factors (not shown) that tend to support one another, such as testability and maintainability -- similar sets of criteria support both factors.

So you get the idea of defining quality needs for specific applications. As this process of definition continues, a *profile* begins to emerge that describes the proposed software in terms of weighted quality factors.

## The Quality Profile

I introduce this term to describe the prioritized, weighted list of quality factors that the user & developer define for their software development effort. The Quality Profile is a "signature" or "fingerprint" of a project's quality needs. Humphrey [10] offers a common–sense example of what kinds of factors are important for different applications, based upon the "primary concern" of the application.

| | *Primary Concern* | *High Priority Quality Factors* |
|---|---|---|
| a. | Effect on human lives | Reliability, Correctness, Testability |
| b. | Long life Cycle | Maintainability, Flexibility, Portability |
| c. | Real time application | Efficiency, Reliability, Correctness |
| d. | In–house tool | Efficiency, Reliability, Correctness |
| e. | Classified Information | Integrity |
| f. | Communicating systems | Interoperability |

The High Priority Quality Factors shown for each type of application begin to define that application's quality profile. The profile of an application of type "a" is given by high degrees of reliability, correctness and testability, and lower degrees of the remaining factors. In practice, we define a more precise scale of degrees and assign a particular weight to each factor. The resultant set of quality factor weights defines the quality profile for the proposed software.

Another example, more generic, is given by Deutsch [5] to suggest an *initial prioritization* of Quality Factors by "software category".

| | *Software Category* | *High Priority Quality Factors* |
|---|---|---|
| a. | Critical | Safety, Survivability, Correctness, Maintainability, Efficiency |
| b. | Support | Maintainability, Verifiability, Interoperability, Portability, Usability, Correctness |
| c. | I/O | Correctness, Interoperability, Maintainability |
| d. | Data | Interoperability, Portability, Reusability |
| e. | Computational | Correctness, Maintainability |
| f. | Environment | Maintainability, Verifiability, Correctness, Interoperability, Portability, Reusability, Efficiency, Integrity |
| g. | MMI | Integrity, Usability |
| h. | Documentation | Correctness, Maintainability |
| i. | Design | Expandability, Flexibility, Interoperability, Maintainability, Portability, Reusability, Verifiability |

These two examples offer starting points for the development of a Quality Profile. Many applications will exhibit multiple concerns or cover several categories. It is the job of the user & developer to define the Quality Profile for the specific application.

## Defining the Quality Profile

Deutsch [5] suggests a metric for ranking or weighting quality factors.

| *Level of quality required* | | *What techniques should be used to ensure a quality factor of this rank* |
|---|---|---|
| E | Excellent | Exceptional techniques |
| G | Good | Better than average techniques |
| A | Average | Normal corporate practices |
| NI | Not an Issue | No special techniques |

He then extends the metric into the realm of cost and schedule prediction, using Jensen and COCOMO model relative cost and relative schedule analysis factors. Cost and schedule prediction will not be pursued further here.

Latter day SQA is also developing standardized means by which the user and developer discuss and come to an agreement of these factors for each application. These means often take the form of questionnaires that prompt the user to evaluate all needs for quality.

## Quality Criteria

This is a common SQA term. **Quality Criteria** are detailed subcharacteristics which the software exhibits that reflect the degree to which the Quality Factors are present. In other words, the planned presence of high–level quality *factors* implies the presence of a detailed set of quality *criteria.*

The Quality Factors are user–oriented; they are designed to map easily to a user's needs for the proposed software. The Quality Criteria are more software–oriented; they are designed to map easily to characteristics that may be evaluated by direct testing of the software. The relationship between quality factors and quality criteria is analogous to that between the two common stages of requirements definition. The analogy does *not* apply to the amount of effort needed to go from the early phase to the later — Quality Factors may be translated immediately to Quality Criteria. Table 2 shows a list of Quality Criteria [5], [21].

## Mapping Quality Factors to Quality Criteria

There is a direct translation from each Quality Factor to a subset of Quality Criteria which support the factor. The sets of criteria that support different factors may be disjoint or may intersect. Some criteria exhibit conflicts similar to

| *Quality Criterion* | *Meaning of criterion in context of software product* |
|---|---|
| **Accuracy** | Achievement of required precision in calculations and outputs |
| **Anomaly Mgmt** | Behavior for recovery from failures |
| **Augmentability** | Maintenance effort required to expand upon functions and data |
| **Autonomy** | Degree of decoupling from execution environment |
| **Commonality** | Use of standards to match "look and feel" of similar applications |
| **Communicativeness** | Appropriateness of inputs and outputs |
| **Completeness** | Degree to which all software is necessary and sufficient |
| **Conciseness** | Amount of code used to implement algorithm |
| **Consistency** | Use of standards to achieve uniformity within software |
| **Distributivity** | Physical (device) separation of function and data (addresses backup) |
| **Document Quality** | Access to complete, understandable information |
| **Communication Efficiency** | Usage of communication resources |
| **Processing Efficiency** | Usage of processing resources |
| **Storage Efficiency** | Usage of storage resources |
| **Functional Scope** | Range of applicability of software product's functions |
| **Generality** | Range of applicability of software's internal units |
| **Independence** | Degree of decoupling from support environment |
| **Instrumentation** | Amount of code devoted to usage measurement or error identification |
| **Modularity** | Cohesion & Coupling of software's modules (design & code) |
| **Operability** | Ease of operating the software |
| **Safety Management** | Degree to which the design addresses hazard avoidance |
| **Self-Descriptiveness** | Understandability of design & code |
| **Simplicity** | Degree to which algorithms map to the problem they solve |
| **Support** | Functionality that addresses the administration of maintenance |
| **System Accessibility** | Controlled access to functions, data and instructions |
| **System Compatibility** | Use of standards to match interfaces with hardware & communications |
| **Traceability** | Ease of finding links between requirements, design and code |
| **Training** | Provisions to help users learn the operation of the software |
| **Virtuality** | Separation of logical implementation from physical component |
| **Visibility** | Objectivity of evidence of correct functioning — ease of test verification |

Table 2 - Quality Criteria

those examined for quality factors. Table 3 shows a translation between Quality Factors and Quality Criteria that shows how the criteria support and influence the factors, either positively or negatively. The traditional direction of translation is from criteria to factor — the SQA or test team measures the criteria from the software, and reports on what quality factors the software thus exhibits. Our method will begin with the user definition of quality factors, and develop a set of criteria that the software must meet in order to satisfy our quality needs.

This table is merged from two different authors' approach to the factor/criteria map [5], [21]. Their perspectives overlap to a high degree, but each one shows a few more, different criteria than the other. I have included them all here in order to work with the most complete universe of factors and criteria possible. Detailed examination of the authors' text reveals that while some factors and criteria sound very similar, they actually do describe different characteristics of the software.

| Quality Factors / Quality Criteria | Correctness | Efficiency | Expandability | Flexibility | Integrity | Interoperability | Maintaianability | Manageability | Portability | Reliability | Reusability | Safety | Survivability | Usability | Verifiability |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Accuracy | | - | | | | | | | | ++ | | + | | + | |
| Anomaly Mgmt | + | - | | | | | | | | ++ | | + | + | + | |
| Augmentability | | -- | ++ | + | | | | | | | + | | | | |
| Autonomy | | - | | + | | + | | | ++ | | + | | + | | |
| Commonality | | | | | - | ++ | | | | | + | | | | |
| Communicativeness | | - | | + | | | + | | | | + | | | + | + |
| Completeness | ++ | | | | | | | | | + | | | | + | |
| Conciseness | + | + | | | | | ++ | | | | | | | | + |
| Consistency | ++ | | | + | | | ++ | | | + | + | | | | + |
| Distributivity | | | | | | | | | | | | + | + | | |
| Document Quality | | | | | | | | ++ | | | ++ | | | | |
| Communication Efficiency | | ++ | | | | | | | - | | | | | | |
| Proccesing Efficiency | | ++ | | | | | | | -- | | | | | | |
| Storage Efficiency | | ++ | | | | | | | -- | | | | | | - |
| Functional Scope | | | | | | + | | | | | + | | | | |
| Generality | | -- | + | ++ | - | + | | | | - | ++ | | | | |
| Independence | | - | | + | | ++ | | | ++ | | ++ | | | | |
| Instrumentation | | - | | | | | + | | | | | | | + | + |
| Modularity | | - | + | ++ | | ++ | ++ | | ++ | | ++ | | + | | ++ |
| Operability | | - | | | | | | | | | + | | | ++ | |
| Safety Mgmt | | | | | | | | | | | | ++ | | | |
| Self-Descriptiveness | | - | ++ | ++ | | | ++ | | + | | ++ | | | | ++ |
| Simplicity | + | + | + | ++ | | | ++ | | + | ++ | ++ | | | | ++ |
| Support | | | | | | | | ++ | + | | ++ | | | | ++ |
| System Access Control | | - | | - | ++ | -- | | | | | | + | | + | |
| System Compatibility | | - | | | | | | | | | | | | | |
| Traceability | ++ | | | + | | | + | | | | + | | | | ++ |
| Training | | | | | | | | | | | + | | | ++ | |
| Virtuality | | | + | | | | | | | | | | + | | |
| Visibility | | | | | | | + | | | | | | | | ++ |

**Table 3 - Quality Factors <=> Quality Criteria Map**

Symbols are used in the cells of the matrix in Table 3 to indicate the influence a criterion has on various factors. Another viewpoint is that they indicate which criteria are necessary to support each factor. A plus under a factor means that the software should be required to exhibit the corresponding criterion, but is subject to trade-off based on any conflicts that arise. A double plus means that the criterion is more important, and less subject to trade-off. A negative under a factor means that it would be wise not to require the software to exhibit the corresponding criterion, but is subject to trade-off based on the influence of other factors. A double negative means that extra effort must be expended to require the software to exhibit the corresponding criterion.

The assignment of pluses and minuses is a subjective process, but the concept has been refined over time by various authors [5], [8], [10], [21].

## SOFTWARE PROCESS MODELS

"The software process is the technical and management framework established for applying tools, methods and people to the software task" [10].

There are a handful of well-defined "process models" or "life-cycles" in the industry today. They each describe a set of activities and products designed to support the successful creation of a software product. The most widely used model is called the Waterfall model. Other models are coming into use that attempt to address the shortcomings of the Waterfall, but they tend to generate very similar information products. Appendix D offers a brief description of other common process models.

The Waterfall model is characterized by a *linear* set of activities and products such that each activity uses the output of previous activities as its input. Here we list general names of the primary technical products of a waterfall model.

| *Activity (phase)* | *Major products generated by activity (phase)* |
|---|---|
| Concept Definition | Feasibility Study, Concept document |
| User Req. Definition | Level-A Requirements Document, Software Management Plan, System Interface Control Document (ICD) |
| System Req. Definition | Level-B Requirements Document, Subsystem ICDs |
| System Design | System Design Document, System Test Plan |
| Implementation | Software, Test Case Document |
| Testing | Test Report |
| Maintenance | Upgraded Software, Maintenance Report |

Note that the waterfall model itself does not really define details of the information products that are to be produced. Most users of the waterfall model recommend a larger set of documentation; these recommendations are usually laid out in a documentation standard.

## SOFTWARE DOCUMENTATION STANDARDS

A Documentation Standard defines all information products that may be generated to support development of the software product. Usually, a documentation standard is packaged with a life-cycle standard. Two common standards are:

SMAP Information System Life Cycle & Documentation Standards [15]
DOD-STD-2167A [6]

For this study, we will use the document set defined by NASA's Information System Life Cycle Documentation Standard -- Appendix A shows the complete list. Our tailoring method will address which of these products are most important for a given set of quality factors.

## ANALYSIS & DESIGN METHODOLOGIES

Within the framework of the software process model, some method must be used to define the content of each product. Formalized methodologies address the complex definition of the requirements and design products of the software process. There are many different methodologies to choose from for use within any software process. The information content of the requirements document, then, may vary according to the technique used to produce it.

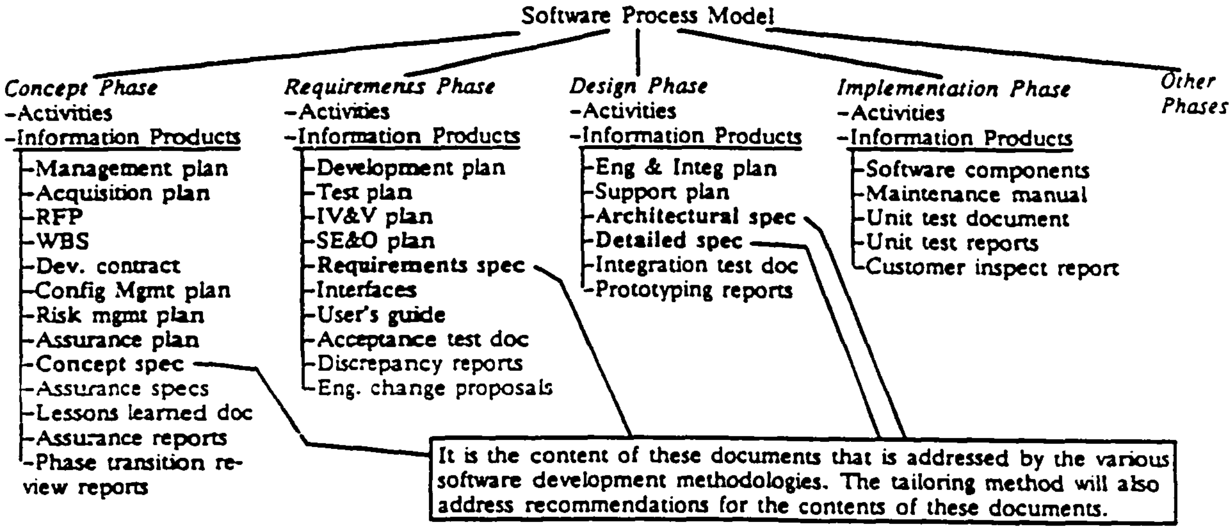
For example, one may choose to specify system requirements using:

a. a simple textual notation developed in an ad hoc manner, or from lessons learned during prototyping.
b. a functional decomposition hierarchy of diagrams, capturing the requirements in processes and data flows.
c. an information model, capturing the requirements in objects, relations and behavior diagrams.
d. a viewpoint/behavior model, capturing requirements in data/action maps and state diagrams.
e. a hybrid of the above techniques, or other techniques.

Appendix C gives a brief overview of some of the more popular methodologies in use today, and lists all the specific products they offer. Our tailoring method may eventually be used to select a meaningful subset of these products; the current version of the paper will not explore this.

## TAILORING INFORMATION PRODUCTS

The hierarchy of SMAP-recommended information products for the software development effort is shown in Figure 1.



It is the content of these documents that is addressed by the various software development methodologies. The tailoring method will also address recommendations for the contents of these documents.

**Figure 1** - SMAP Information Product Overview

Each Information Product shown will be analyzed to determine which quality criteria it best supports. The same analysis will be applied to the information products generated by various development methodologies. At this point, we will be ready to translate a set of 15 user defined Quality Factors into a recommended set of information products.

Tailoring will proceed on three levels:

1. A subset of the document universe will be selected for the specific quality profile. Example: recommend producing a Software Requirements Spec, among other documents.
2. For each selected information product, a subset of it's maximum table of contents will be selected. Example: recommend defining a Data Definition section in the Software Requirements Spec, among other sections.
3. For each recommendation from the table of contents, a set of suggestions will be given to characterize the nature of the information that should appear therein. Example: make the following recommendations for the contents of the Data Definition section: minimize the number of different data representations, minimize number of data conversions, use dynamic memory allocation, pack all data items, etc.

The user/developer then examine the lists of recommendations, and decide whether they make sense in the context of the project. There may still be some manual tailoring to do, but the bulk of the job will have been performed by this method.

## FUTURE WORK

The length of this study was not great enough to develop the full translation from Quality Criteria to Information Products. As a starting point, the requirements volume contents in Appendix B have been mapped to quality criteria. Areas that need more work are:

1. Develop the complete translation between Quality Criteria and all information products listed in the Appendices. This will include not only the selection of specific products, but recommendations for the character of that product's content.
2. Extend the tailoring method to include the tailoring of Management and Assurance activity products, as well as technical development products.
3. Define a weighting scheme for ranking Quality Factors that is consistent with Software Process Model and Design Methodology characteristics.
4. Analyze the list of information products generated by the outstanding process models in use today, and annotate with descriptions of the information content of each product. These descriptions should be compatible with the weighting scheme defined in area 3.

## Appendix A
## LIFE CYCLE PHASES & INFORMATION PRODUCTS: NASA'S SOFTWARE ACQUISITION STANDARD

This appendix lists the life cycle phases and information products for NASA's Software Acquisition Life Cycle as defined by the agency's Software Management and Assurance Program (SMAP). This set of documentation will serve as the universe from which a tailored set will be extracted.

The SMAP plan for volume roll–out describes a mechanism which allows the manager/developer to create information products as sections of one volume, or as separate individual volumes, or as a combination, depending upon the required complexity and management of the particular information product. The tailoring method will select a subset of these information products by recommending the "complexity" of each information product. It is recognized that there are considerations for tailoring other than the quality profile, especially as apply to the Management Plan. Initial tailoring guidelines will focus on the Product Specification, then the Assurance Specification.

### Life Cycle Phases

Concept Definition Phase (CD)

Requirements Definition Phase (Req): User requirements, System Requirements

Design Phase: Software Architectural Design (SAD), Software Detailed Design (SDD)

Implementation Phase.(Impl)

Integration and Test Phase: Integration & Unit Test (I&T), Acceptance Test (AT)

Maintenance, or Sustaining Engineering & Operations (SE&O)

### Information Products: Data Item Descriptions (DID)

#### Management Activity Products: the Management Plan

| *Product* | *Phase(s) during which product is generated, including updates* | | |
|---|---|---|---|
| Component Management Plan | CD | I&T | SE&O |
| Component Acquisition Plan | CD | | |
| Request for Proposal | CD | | |
| Work Breakdown Structure | CD | | |
| Software Development Contract | CD | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Configuration Management Plan | CD | Req | | | | | | |
| Risk Management Plan | CD | | | | | | | |
| Assurance Plan | CD | Req | SAD | | | | | |
| Component Development Plan | | Req | | | | | | |
| Test Plan | | Req | SAD | | | | | |
| Validation & Verification Plan | | Req | SAD | | | | | |
| Sustaining Engineering & Operations Plan | | Req | | | | I&T | | |
| Engineering and Integration Plan | | | SAD | SDD | Impl | | | |
| Product Support Plan | | | SAD | | | | | |

**Technical (Development) Activity Products: the Software Product Specification**

| *Product* | *Phase(s) during which product is generated, including updates* | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Concept Document | CD | | | | | | | |
| Software Requirements Spec (Level-A | CD | | | | | | | SE&O |
| Software Requirements Spec (Level-B | | Req | | | | | | SE&O |
| External Interface Requirements | | Req | | | | | | SE&O |
| User's Guide | | Req | | | Impl | I&T | AT | SE&O |
| Software Architectural Design Spec | | | SAD | | | | | SE&O |
| Software Detailed Design Spec | | | | SDD | | | | SE&O |
| Software Component | | | | | Impl | I&T | AT | SE&O |
| Software Maintenance Manual | | | | | Impl | | | SE&O |
| Version Description Document | | | | | | I&T | AT | SE&O |

**Assurance Activity Products: the Assurance Specification**

| *Product* | *Phase(s) during which product is generated, including updates.* | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Assurance Specs | CD | | | | | | AT | SE&O |
| Acceptance Test Document | | Req | SAD | SDD | Impl | I&T | AT | |
| Integration Test Document | | | SAD | | Impl | I&T | | |
| Unit Test Document | | | | | Impl | | | |

**Management Control & Status Reporting Activity Products**

| *Product* | *Phase(s) during which product is generated, including updates.* | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Lessons-Learned Document | CD | Req | SAD | SDD | Impl | I&T | AT | |
| Assurance Reports | CD | Req | SAD | SDD | Impl | I&T | AT | |
| Phase Transition Review Reports | CD | Req | SAD | SDD | Impl | I&T | AT | |
| Discrepancy Reports | | Req | SAD | SDD | Impl | I&T | AT | SE&O |
| Engineering Change Proposals | | Req | SAD | SDD | Impl | I&T | AT | SE&O |
| Prototyping Reports | | | SAD | | | | | |
| Unit Test Reports | | | | | Impl | | | |
| Customer Inspection Reports | | | | | Impl | | | |
| Integration Test Reports | | | | | | I&T | | |
| Certification Reports | | | | | | | AT | |
| Performance/Metrics Reports | | | | | | I&T | AT | SE&O |

## Appendix B
## INFORMATION CONTENT of the NASA-SMAP STANDARD SOFTWARE PRODUCT SPECIFICATION

This appendix lists the full table of contents for SMAP's Software Product Specification (SMAP-DID-P000-SW). This document package contains a Software Concept Document, a Software Requirements Spec, a Software Architectural Design Spec, a Software Detailed Design Spec, a delivery Version Description, a User's Manual and a Maintenance Manual. (from [15]). The contents have been extended to include a more complete list of information items that may be useful (from [1]). The extended items are italicized.

An initial pass at mapping document sections to quality criteria has been performed for the Requirements Volume -- the map uses abbreviations shown in the key below, and should be read "backwards" for each criterion. In other words, the map is to be used by selecting those document sections that show a reference to each criterion that is specified by the quality profile.

| | | | | | |
|---|---|---|---|---|---|
| Ac: | Accuracy | DQ: | Document Quality | Sf: | Safety Management |
| AM: | Anomaly Mgmt | EC: | Communication Efficiency | Sd: | Self-descriptiveness |
| Ag: | Augmentability | EP: | Processing Efficiency | Sm: | Simplicity |
| At: | Autonomy | ES: | Storage Efficiency | Sp: | Support |
| Cm: | Commonality | FS: | Functional Scope | SA: | System Accessibility |
| Cc: | Communicativeness | Gn: | Generality | SC: | System Compatibility |
| Cp: | Completeness | Ip: | Independence | Tc: | Traceability |
| Cn: | Conciseness | Is: | Instrumentation | Tr: | Training |
| Cs: | Consistency | Md: | Modularity | Vr: | Virtuality |
| Ds: | Distributivity | Op: | Operability | Vs: | Visibility |

Key: Quality Criteria Abbreviations

The Introduction and Related Documentation sections are recommended in their entirety for every software development effort. Content of the volumes following will be addressed by the tailoring method.-(At present, only the Requirements Volume is addressed).

**Introduction**
  Identification of Volume
  Scope of Volume
  Purpose and Objectives of Volume
  Volume Status and Schedule
  Volume Organization and Roll-Out

**Related Documentation**
  Parent Documents
  Applicable Documents
  Information Documents

**Concept Volume**
  Definition of Software
    Purpose and Scope
    Goals and Objectives
    Description
    Policies
    *Anticipated Uses of System*
    *Optional Configurations*
  User Definition

*Overview of the User Organization*
*Logical organization*
*Physical organization*
*Temporal organization*
*reporting cycles*
*scheduled events*
*Information flow organization*
Capabilities and Characteristics
Sample Operational Scenarios
*Anticipated Operational Strategy*
*System ownership*
*System administration*
*operational control*
*modification policy*
*change support*
*User administration*
*departments*
*skill levels*
*Funding strategy*
*Currently Used Procedures*

**Requirements Volume**
Requirements Approach and Tradeoffs————DQ, Tc
*Design Standards to be used*————Cm, Cs, Md, SC
*World Model (Information model) type A*————Ag, Cc, Md, Sd, Vr
*Entity–Relation summary (Data Requirements)*
*Entities: description, attributes, class size*
*Attributes: description, values, defaults, constraints, class size, retention/archive requirements*
*Relationships: description, size, components, constraints*
*Individuals (instantiations of entities)*
*World Model (Information model) type B*————Ag, Cc, Md, Sd, Vr
*Objects: description, allowed operations, class size*
*Allowed Operations: constructors, interrogators, iterators, etc.*
*Messages: sent, received*
External Interface Requirements————Cc, EC, SC
*Operational Resources & Resource Limitations*————EC, EP, ES, Vr
**Requirements Specification**
Process and Data Requirements
Function Input data & Source————Ac, Ag, AM, Cc, Cm, Gn, SC, Sd, Tc, Vs
Function Transactions and Algorithms————Ac, Ag, AM, Cp, Cs, EP, FS, Gn, Md
Function Output data & Destination————Ac, Ag, AM, Cc, Cm, Gn, SC, Sd, Tc, Vs
*Function Triggering mechanisms & conditions*————AM, Cm, EP
*Function Termination mechanisms & conditions*————AM, Cm, EP
*Function Expected demand*————EP
Data Definition————Ac, Ag, At
Data Relationships————Ac, Ag, At
Data Protection requirements————Op
Data Validity check requirements————Ac, AM, Gn, Ip, Op, SA
Data Parameterization requirements————Ac, Ag, Gn, Sd, Vr
Data Format or Implementation Restrictions————Ac, Ag, At
*System Behavior Requirements*
*Phases & Modes*————Ac, Ag, AM, Sf
*System Actions*————Ag, AM, Cm, Sf

| | |
|---|---|
| Performance and Quality Engineering Requirements | |
| Timing & Sizing requirements | EC, EP, ES |
| Sequencing & event timing requirements | EC, EP |
| Throughput & capacity requirements | EC, EP |
| Error Detection, Isolation, Recovery requirements | Ac, AM, Ds, Is, Sf |
| Quality Engineering requirements | ALL |
| *Quality factors required* | |
| Safety Requirements | AM, Sf, SA |
| Security and Privacy Requirements | |
| Access requirements | |
| *to functions* | Cm, Sf, SA |
| *to data* | Cm, Sf, SA |
| *to code* | Sf, SA |
| *Legal requirements* | Sf |
| *Audit requirements* | Vs |
| *Other policy-based requirements* | |
| Implementation Constraints | Ag, Ds, Ip |
| Site Adaptation | Ag, At, Gn |
| Design Goals | Cn, Cs, Gn, Sm |
| *Human Factors Requirements* | |
| *User type definition* | |
| *level of computer sophistication* | Op, Cc |
| *technical competence required* | Op, Cc |
| *Physical constraints* | |
| *response time* | Cm, Op |
| *special physical limitations/requirements* | Cm, Op |
| *On-line help requirements* | Op |
| *Robustness requirements* | AM, Gn, Sf, SA |
| *Failure message & diagnostic requirements* | AM, Cm, Cc, Gn, Is, Op |
| *Input/Output convenience requirements* | Cm, Cc, Is, Op |
| *defaults* | |
| *formats* | |
| Traceability to Parent's Design | Tc, Sm |
| Partitioning for Phased Delivery | DQ, Tc, Vs |

**Design Volume**
- Architectural Design
  - Design Approach and Tradeoffs
  - Architectural Design Description
  - External Interface Design
  - Requirements Allocation and Traceability
  - Partitioning for Incremental Development
- Detailed Design
  - Detailed Design Approach and Tradeoffs
  - Detailed Design Description
  - External Interface Detailed Design
  - Coding and Implementation Notes
  - Firmware Support Manual

**Version Description Volume**
- Product Description
- Inventory and Product
  - Materials Released
  - Product Content
- Change Status
  - Installed Changes

Waivers
Possible Problems and Known Errors

**User Documentation Volume**
User's Guide
Overview of Purpose and Function
Installation and Initialization
Startup and Termination
Functions and their Operation
Error and Warning Messages
Recovery Steps
User's Training Materials

**Maintenance Manual Volume**
Implementation Details
Modification Aids
Code Adaptation
*Standards*

**Abbreviations and Acronyms**

**Glossary**

**Notes**

**Appendices**

# Appendix C
# DESIGN METHODOLOGIES and their INFORMATION PRODUCTS

This appendix lists information products generated by the more popular analysis & design methodologies of the day (compiled from [3], [9]). These products make up a portion of the contents of the Software Product Spec as listed in Appendix A and Appendix B. It is hoped to extend the tailoring method to recommend an appropriate set of design methodology information products based on the quality profile.

## Functional Decomposition

### Structured Design (SD) — Constantine/Myers/Yourdon

This is the traditional data flow diagram methodology that has been in use since the early seventies. It's main products are a hierarchical set of data flow diagrams, process specifications and a data dictionary. State transition diagrams may also be used when deemed necessary by the analyst.

### Real Time Structured Analysis & Design (RTSAD)

This methodology is similar to SD, but includes the analysis and design of control flow between processes. State transition diagrams, decision tables and process activation tables are used with more regularity.

## Object Oriented Design (OOD)

### OOD — Booch

The objects defined in Booch's OOD have associated attributes and allowed operations. They use the concepts of visibility, class and inheritance, and they communicate with each other via message passing. One of Booch's goals in designing this methodology was to be compatible with the Ada language, and the objects map well to Ada constructs.

### GOOD (General OOD) — Seidewitz

The objects defined in this OOD have associated attributes only. They are tied to one another not by message passing, but by defined relationships. This is an attempt to model the real world more closely, and appiies well to non-real time applications.

## Other Methodologies

### Jackson Structured Design (JSD) — Jackson

This unique approach was an early contender on the requirements modeling scene, and is still going strong. As industry has developed the terms, we discover that JSD is a natural hybrid of Object Oriented and Functional Decomposition methodologies. JSD has its own set of information products which do not match 100% any of the traditional products in the map below, but I show what traditional products are most like those produced by JSD, rather than specifying and defining new product categories.

### Ada-based Design Approach for Real Time Systems (ADARTS) — Gomaa

This methodology is an Ada-based version of DARTS; it builds upon the SCR module structuring criteria, the Booch object structuring criteria, and the DARTS task structuring criteria to generate maintainable and reusable software components. It offers consideration of the concurrent nature of real-time systems. The analysis and design diagrams use the "Booch-gram" Ada notation.

### Software Cost Reduction (SCR) — Parnas

This real-time oriented methodology concentrates on the modules that will make up the software product, an information-hiding hierarchy into which they fall, and the interfaces which they use among themselves. Without trying, it is almost object oriented. The methodology offers strong support for software reuse.

### Software Productivity Consortium Methodology (SPCM) — Gomaa

This methodology is based on SCR. Its primary areas of focus are the inclusion of rapid prototyping techniques and the production of reusable software.

## Information Products of the Methodologies

| *Product* | *Methodologies which support generation of product* | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Context Diagram | SD | Rtsad | | | | | | |
| Data Flow Diagrams | SD | Rtsad | | GOOD | JSD | Adarts | | |
| Control Flow Diagrams | SD | Rtsad | | | | | | |
| Control Transformations (State Transitions) | SD | Rtsad | OOD | GOOD | JSD | Adarts | SCR | SPCM |
| Mini-Specs | SD | Rtsad | | | | | | |
| Data Dictionary | SD | Rtsad | | | JSD | | | |
| Structure Charts | SD | Rtsad | | | JSD | Adarts | | |
| Hardware Diagram | | | OOD | | | | | |
| Class Structure Diagram | | | OOD | | | | | |
| Architecture Diagram | | | OOD | | | | | |
| Ada Package Specs | | | OOD | | | | | |
| Object Diagram | | | OOD | GOOD | JSD | | | |
| Entity-Relation Diagrams | | Rtsad | | GOOD | | | | |
| Process Definitions | | | | GOOD | | | | SPCM |
| Object Composition | | | | GOOD | | | | |
| Object Descriptions | | | | GOOD | | | | |
| Task Structure Specs | | | | | | Adarts | SCR | SPCM |
| Module Guide | | | | | | Adarts | SCR | SPCM |
| Module Interface Specs | | | OOD | GOOD | | Adarts | SCR | SPCM |
| "Uses" Structure | | | | | | Adarts | SCR | SPCM |
| Module Internal Design Spec | | | | | | | SCR | SPCM |
| Subset Spec | | | | | | | SCR | SPCM |

# Appendix D
# OTHER SOFTWARE PROCESS MODELS

A sampling of Software Process Models other than the Waterfall Model are briefly described here. Recall that their associated information products are very similar to those described in Appendix A.

## Spiral

A management oriented model. Activities and products are almost identical to those of the waterfall model, but are interspersed with regular prototyping and risk analyses efforts to guide the process.

## Rapid Prototyping

This prototyping model covers the requirements definition phases of the waterfall or other similar model. It is generally recommended for never-before-attempted solutions, or when the user & developer deem areas of the problem concept to be technologically difficult.

A partial implementation of the system is constructed from *informal requirements*, usually of *poorly understood* areas. Users exercise of the prototype to better understand and define requirements. The prototype must then be discarded, and system design is begun from the requirements.

It is important to avoid temptations to keep and build upon the prototype, because the very nature of *rapid* prototyping causes generation of code that is inefficient, unsafe, unreliable, unmaintainable, etc. If, during development of the prototype, algorithms or designs are discovered that are particularly efficient, safe, reliable, maintainable, etc, they should be documented for consideration during the "real" design.

## Evolutionary Prototyping

This prototyping model is also recommended for technologically difficult problems, but covers a larger area of the life cycle. It is hoped that the evolutionary prototyping efforts will help guide and speed the requirements definition, system design and implementation phases.

A partial implementation of the system is constructed from *partially known, well defined requirements*, usually of *well understood* areas. Users exercise the prototype to better understand and define remaining requirements. The prototype forms a set of baseline software which will be built upon to complete the deliverable versions. At this point, the model *may* transition to the Iterative Enhancement model.

Development of an evolutionary prototype begins with well defined requirements. It takes longer than rapid prototyping, because good software engineering practices must be used to develop code that will eventually be part of the working product.

## Iterative Enhancement a.k.a. Incremental Development

This model is recommended for applications that have a basic, well understood core set of functions. The model is characterized by many releases of new versions which add new functionality. Many market-penetration schemes will use this model to get a product into the marketplace and generating revenue, to pay for later enhancements. A rather complete set of requirements is known up front, and the releases of new functions are planned in advance; of course, the model is adaptable to new requirements and relies on user feedback to improve the product.

## Software Reuse

This model may be used to cover the design portion of the waterfall or other similar model. It's design paradigm relies mostly on the incorporation of previously proven designs and code into new software products.

## Automated Software Synthesis

This is an advanced model that usually requires strict formulation of requirements using a regular grammar specification language. This model offers the direct (and hopefully, automatic) transformation of requirements and/or high level design into code, either algorithmically or using a knowledge based rule set. It is hoped to eliminate the middle portions of the documentation set, centering around the detailed design.

CASE tools currently exist that support this model to some degree. Typically, they will generate Ada package specs and the interface portions of package bodies from structure charts.

# REFERENCES

[1] Abbot, R., *An Integrated Approach to Software Development*, John Wiley & Sons, NY 1986.

[2] Basili, V.; Rombach, H., "Tailoring the Software Process to Project Goals and Environments", *9th International Conference on Software Engineering*, IEEE Computer Society, Washington, DC 1987.

[3] Davis, A., "A Comparison of Techniques for the Specification of External System Behavior", *Communications of the ACM*, 31,9 (September 1988).

[4] Davis, A.; Bersoff, E.; Comer, E., "A Strategy for Comparing Alternative Software Development Life Cycle Models" *IEEE Transactions on Software Engineering*, 14,10 (October 1988).

[5] Deutsch, M.; Willis, R., *Software Quality Engineering: A Total Technical and Management Approach*, Prentice-Hall, Englewood Cliffs, NJ 1988.

[6] DOD-STD-2167A, *Military Standard: Defense System Software Development*, Department of Defense, Washington, DC, 1988.

[7] Fox, G., "Performance Engineering as a Part of the Development Life Cycle for Large-Scale Software Systems" *11th International Conference on Software Engineering*, IEEE Computer Society, Washington, DC 1989.

[8] Gilb, T., *Software Metrics*, Winthrop Publishers, Cambridge, 1977.

[9] Gomaa, H.; Kirby, J.; Weiss, D., "Comparison of Software Development Methodologies",*Presentation at Software Productivity Consortium Methodology Workshop*, March 1989.

[10] Humphrey, W., *Managing the Software Process*, Addison-Wesley, Reading, MA 1989.

[11] Humphrey, W., "Software Process Modeling: Principles of Entity Process Models" *9th International Conference on Software Engineering*, IEEE Computer Society, Washington, DC 1987.

[12] IEEE, *Software Engineering Standards*, IEEE Computer Society, Washington, DC 1987.

[13] Jackson, M., *System Development*, Prentice-Hall, Englewood Cliffs, NJ 1983.

[14] Krasner, H.; Pore, M., "A Software Process Management Approach to Quality and Productivity", Lockheed Software Technology Center, 1989.

[15] NASA, *Software Management and Assurance Program (SMAP) Information System Life Cycle and Documentation Standards Release 4.3*, NASA Office of Safety, Reliability, Maintainability and Quality Assurance, 1989.

[16] Poore, J., "Derivation of Local Software Quality Metrics (Software Quality Circles)" *Software Practice and Experience*, 18,11 (November 1988).

[17] Pressman, R., *Making Software Engineering Happen: A Guide for Instituting the Technology*, Prentice-Hall, Englewood Cliffs, NJ 1988.

[18] Pressman, R., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, NY 1982.

[19] Rowen, R., "Software Project Management Under Incomplete and Ambiguous Specifications" *IEEE Transactions on Engineering Management*, 37,1 (February 1990).

[20] Tully, C., *Proceedings, 4th International Software Process Workshop*, ACM Press, NY 1989.

[21] Vincent, J.; Waters, A.; Sinclair, J., *Software Quality Assurance, Volume 1: Practice and Implementation*, Prentice-Hall, Englewood Cliffs, NJ 1988.

# VIEWGRAPH MATERIALS

# FOR THE

# M. AREND PRESENTATION

# A Method for Tailoring the Information Content of a Software Process Model

**REVIEW PACKAGE**
15th Annual Software Engineering Workshop
Goddard Space Flight Center, Greenbelt, MD
November 28, 1990

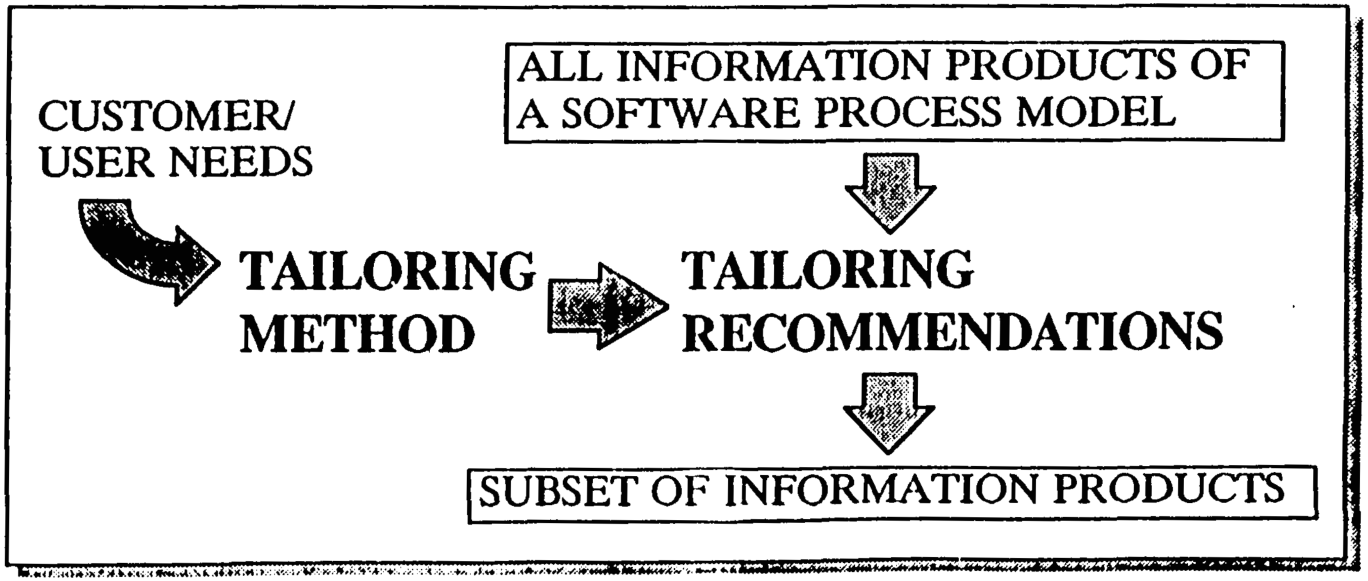
Mark Arend (McDonnell Douglas)
*for*
David Howes (NASA JSC)
*and*
Dr. Sharon Perkins (University of Houston, Clear Lake)

# DEFINITIONS

- SOFTWARE PROCESS MODEL (or LIFE CYCLE)
  - "The technical and management framework established for applying tools, methods and people to the software task."
  - Applies to the entire development cycle of the software, from concept to maintenance.
- SOFTWARE METHODOLOGY
  - Definition of a means for capturing requirements and design.
  - Applies to one or more portions of the development cycle, usually requirements analysis, specification or design.
- TAILORING
  - Selecting a *subset* of a Process Model or a Methodology for practical application.
- SOFTWARE QUALITY
  - The degree to which software matches customer/user needs.

# INTRODUCTION

- MANY SOFTWARE PROCESS MODELS AND SOFTWARE METHODOLOGIES RECOMMEND TAILORING.
- TAILORING IS USUALLY GUIDED BY PERSONAL EXPERIENCE, ABILITY, AND TRADITION.
- WE WILL DESCRIBE A *METHOD* FOR TAILORING.



ALL INFORMATION PRODUCTS OF A SOFTWARE PROCESS MODEL

CUSTOMER/ USER NEEDS

**TAILORING METHOD**

**TAILORING RECOMMENDATIONS**

SUBSET OF INFORMATION PRODUCTS

M. Arend
McDonnell Douglas
Page 20 of 31

# CHARACTERIZING CUSTOMER/USER NEEDS

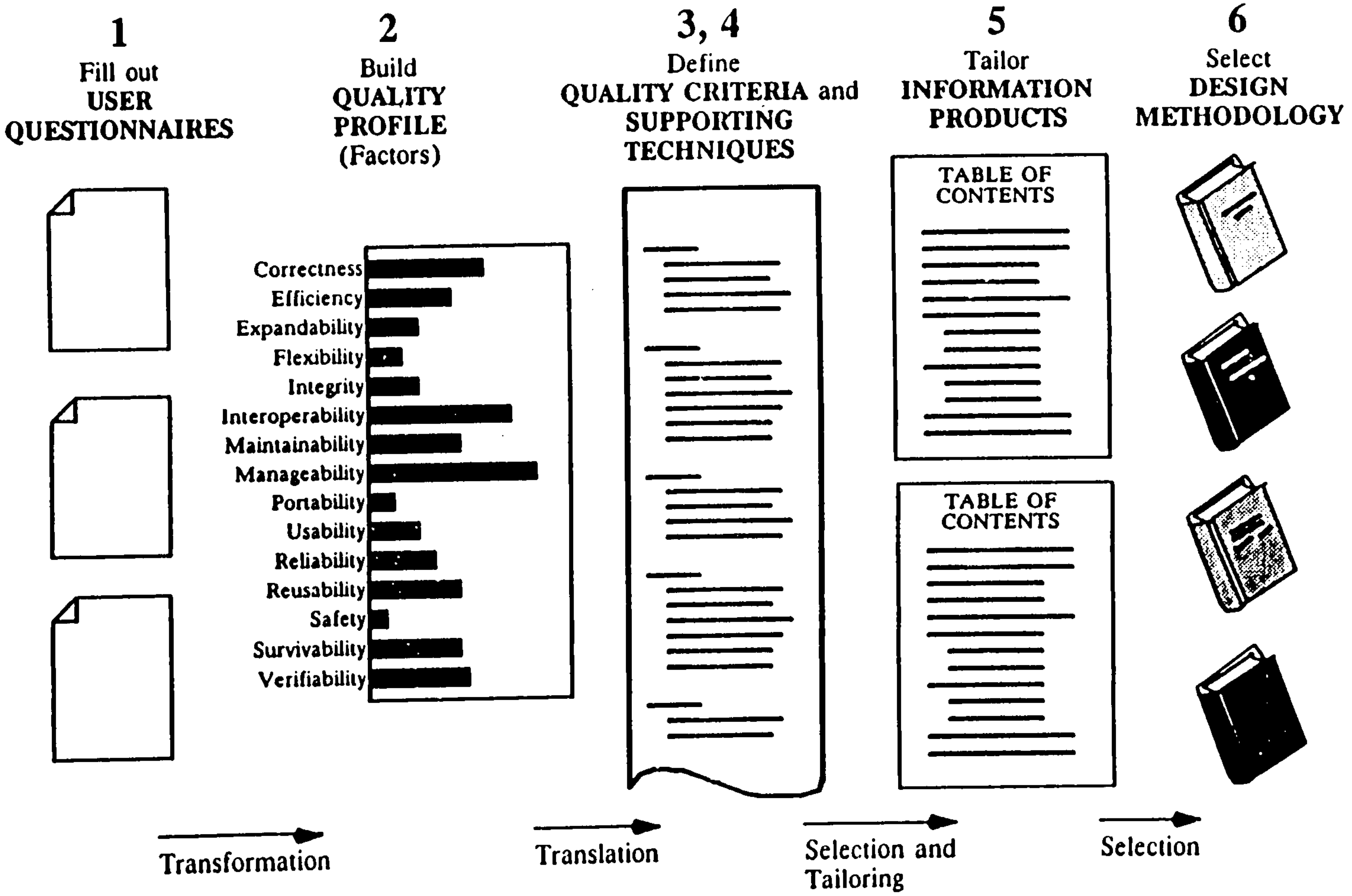
- WE WILL USE CONCEPTS FROM SOFTWARE QUALITY ASSURANCE (SQA) TO EXPLORE CUSTOMER NEEDS:
  - What constitutes appropriate fitness for use of this software?
  - What attributes must this software exhibit to be considered of high quality?
  - Remember, software quality is more than "goodness", it is a measure of how well the software matches the needs of the customer and user.
- SQA SHOWS HOW TO OBJECTIFY A QUALITY RATING OF SOFTWARE, BY EVALUATING *QUALITY FACTORS*.
  - Capture Quality Factors through Customer/User interviews.
- SQA SHOWS HOW TO TRANSLATE QUALITY FACTORS TO *QUALITY CRITERIA*, WHICH ARE MORE DIRECTLY RELATED TO SOFTWARE TESTABILITY.
  - Derive Quality Criteria from Quality Factors
  - Derive development techniques to enforce Quality Criteria

M. Arend
McDonnell Douglas
Page 21 of 31

# THE METHOD'S STEPS

1. PERFORM STANDARD INTERVIEWS AND DIALOGS BETWEEN DEVELOPER AND CUSTOMER/USER.
2. GENERATE A PROFILE OF QUALITY FACTORS OF THE SOFTWARE TO BE DEVELOPED.
3. TRANSLATE THIS QUALITY–NEEDS PROFILE INTO A SET OF QUALITY CRITERIA THAT MUST BE MET BY THE SOFTWARE.
4. MAP THE CRITERIA TO A SET OF REQUIREMENT AND DEVELOPMENT TECHNIQUES.
5. SELECT AND TAILOR THE INFORMATION PRODUCTS WHICH MATCH OR SUPPORT THOSE TECHNIQUES.
6. SELECT AND TAILOR DESIGN METHODOLOGY(S) TO PRODUCE THESE INFORMATION PRODUCTS.

# THE METHOD'S STEPS



1 Fill out **USER QUESTIONNAIRES**

2 Build **QUALITY PROFILE** (Factors)

3, 4 Define **QUALITY CRITERIA** and **SUPPORTING TECHNIQUES**

5 Tailor **INFORMATION PRODUCTS**

6 Select **DESIGN METHODOLOGY**

Transformation

Translation

Selection and Tailoring

Selection

M. Arend
McDonnell Douglas
Page 23 of 31

# Step 1

*PERFORM STANDARD INTERVIEWS AND DIALOGS BETWEEN DEVELOPER AND CUSTOMER/USER*

- QUESTIONNAIRES DESIGNED TO PROBE THE USER'S NEEDS FOR QUALITY.
- IMPORTANT TO DEFINE BOUNDARY OF SPECIFICATION, TO PREVENT OVER– OR UNDER–SPECIFICATION OF QUALITY NEEDS.
- DEVELOPER WRITES QUESTIONNAIRES, USING A GREAT DEAL OF BOILERPLATE AND HELPS CUSTOMER/USER THROUGH THE PROCESS.
- EXAMPLE QUESTIONS
  - ✓ How many users will want to use the system simultaneously?
  - ✓ What level of user training is acceptable?
  - ✓ Will other computer systems rely on this one?

# Step 2

*GENERATE A PROFILE OF QUALITY FACTORS OF THE SOFTWARE TO BE DEVELOPED*

- QUANTIFY RESPONSES TO USER QUESTIONNAIRES.
- THE TAILORING METHOD DEFINES A TRANSFORMATION BETWEEN POSSIBLE RESPONSES AND QUALITY FACTORS.
- THE TRANSFORMATION WILL APPLY WEIGHTED VALUES TO EACH RESPONSE, BASED UPON THE EFFECT THE ISSUE PROBED BY THE QUESTION HAS UPON ITS RELATED FACTOR(S). (Most questions will deal with decisions that influence several factors to varying degrees, even positively for some and at the same time negatively for others).
- SINCE SOME FACTORS CONFLICT WITH OTHERS, A SECOND USER INTERVIEW MAY BE NECESSARY TO AMPLIFY RELATIVE IMPORTANCE. Factor conflict may assist risk identification and management.

# Step 3

*TRANSLATE THE QUALITY–NEEDS PROFILE INTO A SET OF QUALITY CRITERIA THAT MUST BE MET BY THE SOFTWARE*

- PRE–DEFINED GUIDELINES MAP FACTORS TO CRITERIA.

- THIS TRANSLATION BRINGS US CLOSER TO WHAT QUALITY MEANS IN TERMS OF A SOFTWARE PRODUCT, RATHER THAN IN TERMS OF THE USER.

- SOME CRITERIA ALSO CONFLICT WITH ONE ANOTHER. THIS TRANSLATION WILL ASSIGN RELATIVE WEIGHTS TO THE CRITERIA TO HELP REDUCE CONFLICTS.

- REMEMBER, CONFLICTS ARE NOT IMPOSSIBILITIES, THEY MERELY IDENTIFY AREAS REQUIRING EXTRA EFFORT AND EXCEPTIONAL TECHNIQUES – RISK MANAGEMENT.

# Step 4

*MAP THE CRITERIA TO A SET OF REQUIREMENT AND DEVELOPMENT TECHNIQUES*

- TECHNIQUES OF DEVELOPMENT AND MANAGEMENT MAY BE USED TO ENSURE THE PRESENCE OF VARIOUS QUALITY CRITERIA.
- TYPES OF TECHNIQUES
  - ✓ Product Recommendation
  - ✓ Method Recommendation
  - ✓ Standards Recommendation
  - ✓ General Guidelines
- EXAMPLES
  - ✓ Produce a traceability matrix to ensure *Completeness.*
  - ✓ Use prototyping to ensure *Usability.*
  - ✓ Adhere to interface standards to ensure *Commonality.*
  - ✓ Separate critical & non–critical functions to ensure *Safety Management.*

M. Arend
McDonnell Douglas
Page 27 of 31

# Step 5

*SELECT AND TAILOR THE INFORMATION PRODUCTS WHICH MATCH OR SUPPORT THE TECHNIQUES*

- INFORMATION PRODUCTS ACT AS SPECIFIC GOALS WHICH FORCE US TO RECOGNIZE, FORMALIZE AND ADHERE TO TECHNIQUES TO SPECIFY, DESIGN AND IMPLEMENT SOFTWARE OF APPROPRIATE QUALITY.

- INFORMATION PRODUCTS DOCUMENT REQUIREMENTS AND DESIGNS, PROVIDING FOR CONTINUITY OF DEVELOPMENT AND MAINTENANCE.

- WE WISH TO SELECT THE APPROPRIATE SUBSET OF ALL POSSIBLE INFORMATION PRODUCTS.

- THE TAILORING METHOD WILL DESCRIBE A UNIVERSE OF INFORMATION PRODUCTS, AND WILL OFFER A DIRECT TRANSLATION FROM QUALITY CRITERIA TO RECOMMENDED SUBSET OF THAT UNIVERSE.

# Step 6

*SELECT AND TAILOR THE DESIGN METHODOLOGY WHICH PRODUCES THESE INFORMATION PRODUCTS*

- MANY METHODOLOGIES ARE AVAILABLE FOR SOFTWARE REQUIREMENTS SPECIFICATION, SOFTWARE DESIGN AND IMPLEMENTATION.

- THE TAILORING METHOD WILL DESCRIBE A UNIVERSE OF METHODOLOGIES, AND WILL CATEGORIZE THEM BY THE INFORMATION PRODUCTS THEY PRODUCE.

- THE MATCHUP BETWEEN INFORMATION PRODUCTS PRODUCED BY A METHODOLOGY AND THOSE RECOMMENDED TO ACHIEVE THE QUALITY PROFILE FACILITATES THE SELECTION OF AN APPROPRIATE METHODOLOGY.

# CURRENT STATUS AND FUTURE WORK

- THIS PHASE OF THE RESEARCH EFFORT DEALT WITH DISCOVERY OF CONCEPTS AND ASSEMBLY OF DATA.

- AREAS ALREADY DEVELOPED TO SOME EXTENT
  - ✓ Translation from Quality Profile to Quality Criteria
  - ✓ List of Techniques sorted by Quality Criteria
  - ✓ Universe of Information Products (enhanced NASA SMAP standard)
  - ✓ Universe of Methodologies

- AREAS FOR DEVELOPMENT
  - ✓ User Questionnaire boilerplates
  - ✓ Response weighting scheme
  - ✓ Transformation of weighted responses to Quality Profile
  - ✓ List of Information Products sorted by Quality Criteria

# CONTACTS

David B. Howes
(Manager for Engineering IRM, under Information Systems Directorate, Service Management Division)
Code PS2
Lyndon B. Johnson Space Center
National Aeronautics and Space Administration
Houston, TX 77058
(713) 483-8381

Mark Arend
839 Walbrook Drive
Houston, TX 77062
(713) 480-7332

Dr. Sharon Perkins
Assistant Professor of Computer Science and Information Systems
University of Houston, Clear Lake
2700 Bay Area Boulevard
Houston, TX 77058-1098
(713) 488-7170

M. Arend
McDonnell Douglas
Page 31 of 31

N92
19426
UNCLAS

N92-19426

P- 25

# SOFTWARE TECHNOLOGY INSERTION: A STUDY OF SUCCESS FACTORS

Submitted to:
**The Fifteenth Annual Software Engineering Workshop**
November 28-29, 1990
NASA/Goddard Space Flight Center
Greenbelt, MD

by

**Tom Lydon**
**Raytheon MSD**
50 Apple Hill Drive
Tewksbury, MA 01876

Managing software development in large organizations has become increasingly difficult due to constantly increasing technical complexity, stricter government standards, a shortage of experienced software engineers, competitive pressure for improved productivity and quality, the need to co-develop hardware and software together, and the rapid changes in both hardware and software technology.

The "software factory" approach to software development minimizes risks while maximizing productivity and quality through standardization, automation, and training. However, in practice, this approach is relatively inflexible when adopting new software technologies. How can a large multi-project software engineering organization increase the likelihood of successful **software technology insertion (STI)**, especially in a standardized engineering environment?

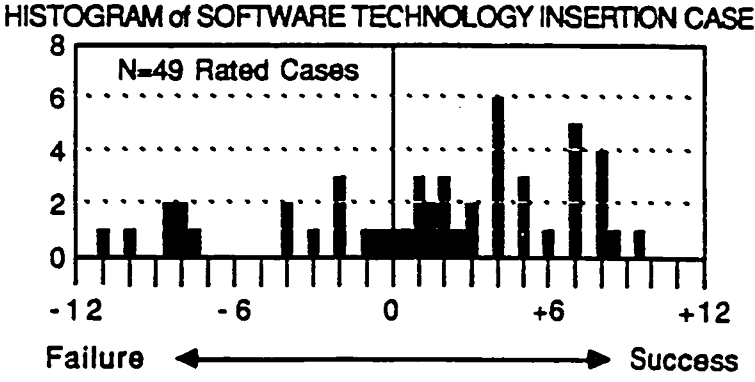


Figure 1 - Distribution of scores from 49 rated STI Cases

In an attempt to correlate various success factors with levels of success, 59 cases of "new software technology insertion" in thirteen recent projects at a large U.S. Defense electronics contractor were identified and categorized according to several criteria. The relative success or failure of 49 of these cases (see **Figure 1**) was determined by having key project personnel (Lead Engineer, Dept Manager, and tool supporters) rate 6 aspects (added together for total rating) of the software technology insertion results. Maximum success was scored as +12, and maximum failure as -12 on the rating scale. *The histogram in Figure 1 illustrates the distribution of scores from the 49 rated cases.*

There were 21 different **new software technologies** studied, most of them new **tools or methods**, including (in approximate lifecycle order):

- The use of DoD-STD-2167 or 2167A
- Structured analysis CASE tools
- Rapid-Prototyping in requirements or design
- In-House requirements traceability tool
- In-House program design language (PDL) tools
- Reusable Software in design or coding
- The use of Ada® as an implementation language
- The use of M68020 assembly language
- Microprocessor Development Stations (MDS) for integration testing
- In-House configuration management (CM), source code control tool
- Workstation-based engineering documentation tools
- The use of workstations as primary development platforms

Though meaningful statistical correlations were not possible due to the limited sample size, ratings were compiled and empirically compared with several technology factors measured for each STI case, including:

- Technology Type (Competence-Enhancing or -Destroying)
- Support Type (In-House or External)
- Maturity of the Technology (Young, Mature, and Old)
- Project Size (SLOC)
- Prior Expectations (for success or failure)
- Reasons (for using the new software technology)
- Methods (of inserting the new software technology)
- Perceived Time Savings
- Perceived Labor Savings
- Perceived Computer Cost Savings
- Perceived Quality Improvement
- Met Expectations? (for success or failure)

A closer look at the **"Top Eleven"** cases of successful STI (ratings ≥ +7), and the **"Bottom Seven"** cases of unsuccessful STI (ratings ≤ -7) shows that:

1. Perceived **Time Savings** and perceived **Labor Savings** are the <u>most significant real indicators</u> of successful or unsuccessful STI.

2. Though users often complain about increased computer costs, **saving computer cost** is <u>not an indicator of STI success</u>, because it is not usually a goal or a motivator for the use of new technology.

3. Perceived **Quality Improvement** is a <u>strong indicator</u> of STI success, but <u>not an indicator</u> of STI failure.

4. Even in successful STI cases, users' **Prior Expectations** about what a new technology can/cannot do are not managed effectively.

In addition to the success ratings, **on-site structured interviews** were used to profile each new technology, and collect other qualitative information that was used to clarify and complete the data.

Tushman[1] describes new technology types as: (1) **competence-enhancing** - incrementally different, building on existing know-how, and substituting for older technologies without rendering their skills obsolete, or (2) **competence-destroying** - fundamentally different, requiring new skills, abilities, and knowledge for use. The main types of technology support are: (1) **In-House**, where the supporters work in the same organization as the users, or (2) **Outside**, where the supporters work in a different organization than the users.

A sample of the distribution of successful STI cases over these two combined factors (technology type and support type) is show in **Figure 2:**

Ratings of New Technology Types Across Two Dimensions

| | IN-HOUSE<br>Support | OUTSIDE<br>Support |
|---|---|---|
| Competence<br>ENHANCING | #Total= 16<br>#Rated= 13<br>Tot Rating= 47.0<br>Median= +5.0<br>Ave Rating= +3.6<br>BEST | #Total= 9<br>#Rated= 8<br>Tot Rating= 11.5<br>Median= +4.0<br>Ave Rating= +1.4<br>OK |
| Competence<br>DESTROYING | #Total= 11<br>#Rated= 8<br>Tot Rating= -2.0<br>Median= +1.5<br>Ave Rating= -0.2<br>Poor | #Total= 23<br>#Rated= 20<br>Tot Rating= 14.5<br>Median= +0.7<br>Ave Rating= +0.7<br>Marginal |

RATING SCALE: +12 = Maximum Success, -12 = Maximum Failure

Figure 2 - Distribution of Success/Failure across two factors

The new software technologies that had the **most successful** STI experience (though across a very limited set of cases) are summarized below:

| #Cases | Ave Rating | New Software Technology |
|---|---|---|
| 1 | +9.5 | In-House Automated Build Tool |
| 2 | +7.5 | Microprocessor Development Stations for Integration |
| 6 | +4.8 | In-House Software Problem Reporting Tool |
| 3 | +3.3 | In-House Configuration Management (CM) Tool |
| 7 | +2.1 | In-House Program Design Language Tool |

The new software technologies that had the **least successful** STI experience are:

| #Cases | Ave Rating | New Software Technology |
|---|---|---|
| 1 | -11.0 | In-House Automated Code Documentation Tool |
| 2 | -8.8 | Workstation-based Engineering Documentation Tool |
| 4 | -4.8 | Workstation-based CASE Tool for Req'ts and Design |

Among the overall conclusions from the study are:

1. Saving **schedule time** and **labor costs** are necessary and sufficient conditions for successful STI

2. **Improving quality** is a necessary, but not sufficient condition for successful Software Technology Insertion (STI)

3. Success with new software technology insertion (STI) is much greater for **competence-enhancing** than for competence-destroying technologies

4. Success with STI is somewhat greater for **in-house** supported technologies than for outside supported technologies

5. Success with STI is greater for **mature** technologies than for either young or old technologies (mature is >1 year after release, <5 years after release)

6. Success with STI is greater when users' expectations about "new technology" are controlled to avoid expecting too much -- exceeding users' expectations is not necessary for successful STI, but not meeting expections (i.e., disappointing them) is a sufficient condition for failure

7. Success with STI can be increased when there is **synergy** between multiple new technologies, such as Ada and workstations

These and other results and conclusions, along with some recommendations for large software development organizations, will be covered at the workshop.

Reference [1] Tushman, M., and Anderson, P., "Technological Discontinuities and Organizational Environments", Administrative Science Quarterly, Sept 1986.

## 13 Software Projects

| Project ID# | Language | SLOC | Current Phase |
|---|---|---|---|
| 1 | Assembly<br>Fortran<br>C | 49200<br>6400<br>2900 | Integration Test |
| 2 | C | 9100 | Design & Code |
| 3 | Assembly<br>C | 4000<br>4500 | Maintenance |
| 4 | C | 8500 | Integration Test |
| 5 | Assembly | 10850 | Design & Code |
| 6 | Assembly | 7100 | System Test |
| 7 | Assembly | 16000 | Integration Test |
| 8 | Fortran | n/a | Cancelled |
| 9 | Ada | 25000 | Design |
| 10 | Ada | n/a | Integration Test |
| 11 | Assembly | 4850 | Integration Test |
| 12 | C | 13700 | Maintenance |
| 13 | Assembly | 18000 | Design & Code |

## 21 New Software Technologies

(most of them new tools, methods, languages)

A - The use of Ada® as an implementation language
B - In-House automated build tool(s)
C - In-House automated code documentation tool
D - In-House program design language (PDL) tools
E - In-House metrics tools for automatic data collection
G - In-House standard test reporting tool based on RDBMS
I - Workstation-based engineering documentation tool
J - The use of M68020 assembly language
K - Microprocessor Development Stations (MDS) for integration testing
L - In-House project scheduling and reporting tool
M - In-House configuration management (CM), source code control tool
N - In-House Vax/Unix documentation package using troff
P - In-House Software Problem Reporting Tool based on RDBMS
R - Rapid-Prototyping in requirements or design
S - Structured analysis graphical CASE tool
T - Structured analysis graphical CASE tool
U - Reusable Software in design or coding
W - The use of workstations as primary development platforms
X - Workstation-based engineering documentation tool
Y - In-House requirements traceability tool
Z - The use of DoD-STD-2167 or 2167A

# Project/Technology Matrix

New Technology ID

Project ID#

| | A | B | C | D | E | G | I | J | K | L | M | N | P | R | S | T | U | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | ● | | | ● | | | | | | | ● | | ● | | | ● | | | ● | | |
| 2 | | | | | ○ | | | | | | | | | | | | | | | | |
| 3 | | ● | | ○ | | | | | | | ● | | ● | ● | | | ● | | | | |
| 4 | | | | | | ● | | | | | | | | | | | | | | | |
| 5 | | | | ○ | | | ○ | | | | | | ● | | | | | | | ○ | ○ |
| 6 | | | | ● | | | | | | ○ | | ○ | ● | | | | | | ● | | ● |
| 7 | | | ● | ● | | | ● | | | | | | ● | | ● | | | | | | ● |
| 8 | | | | | | | | | | | | | | | ● | | | | | | |
| 9 | ● | | | | ● | | | | | ● | | | | | | | ● | | | | ● |
| 10 | ● | | | ● | | | ● | | | | | | ● | | ● | | | ● | | | ● |
| 11 | | | | ● | | | | ● | ● | | ● | | | ● | | | | | | ● | ● |
| 12 | | | | | | | | ● | ● | | | | | ● | | | | | | | |
| 13 | ○ | | | | | | ● | | | | | | | ○ | ● | | | | | | ● |

**59 STI Cases:** ● Rated ○ Not Rated

# Measuring Perceived STI Success

- For each STI Case, 6 **Questions** were asked of:
  (1) Lead Engineer (Project/Matrix)
  (2) Dept Manager (Functional/Matrix)

| ***For Each STI Case:***<br>Statement (Agree or Disagree?) | **Agree.....Disagree**<br>**+2** | <br>**+1** | <br>**0** | <br>**-1** | <br>**-2** |
|---|---|---|---|---|---|
| 1. I would use the new method/tool again | | ✓ | | | |
| 2. The new method/tool saved **schedule time** | | | ✓ | | |
| 3. The new method/tool saved **labor cost** | ✓ | | | | |
| 4. The new method/tool saved **computer cost** | | | | | ✓ |
| 5. The new method/tool **improved quality** | | ✓ | | | |
| 6. The new method/tool met my expectations | | | | ✓ | |

- **Total Rating** for each STI Case is **sum** (example =+1)
  i.e., maximum = +12, minimum = -12

(Note: Questions not weighted)

## References

[1] Tushman, M., and Anderson, P., "Technological Discontinuities and Organizational Environments", Administrative Science Quarterly, Sept 1986.

[2] Scacchi, W., and Babcock, J., "Understanding Software Technology Transfer", MCC Technical Report STP-309-87, October 1987.

## Acknowledgements

• To the Raytheon Company, for helping to sponsor this work at M.I.T. during 1989-1990.

• To Dr. Ralph Katz, for his technical (and non-technical) advice on this thesis work.

• To the M.I.T. Management of Technology Program, in which the work was performed.

## VIEWGRAPH MATERIALS

## FOR THE

## R. LYDON PRESENTATION

The Fifteenth Annual Software Engineering Workshop
NASA/Goddard Space Flight Center, Greenbelt, MD
November 28-29, 1990

# Software Technology Insertion: A Study of Success Factors

Tom Lydon
Raytheon MSD, Mailstop T3ML19
50 Apple Hill Drive, Tewksbury, MA 01876

# Software Technology Insertion (STI)

**Software Technology Insertion**
"New" Software Technology
+ Opportunity to Insert

**"New" Software Technology**
Tool or Method that is unfamiliar to the majority of a Project Team, usually replacing a more familiar one

**Opportunity to Insert**
A software development activity on a new (most likely) or ongoing (less likely) software project

# Successful STI

| | |
|---|---|
| **Perceived STI Success** | User's sense of **Labor** Cost Savings +<br>User's sense of **Computer** Cost Savings +<br>User's sense of Elapsed **Time** Savings +<br>User's sense of **Quality** Improvement |
| **Real STI Success** | Measured **Labor** Cost Savings +<br>Measured **Computer** Cost Savings +<br>Measured Elapsed **Time** Savings +<br>Measured **Quality** Improvement |

# STI "Cases" Overview

**STI Case**
A single incident of STI
on a single project, usually
within a single development phase

**59 STI Cases Identified**
Across 13 different projects;
from 1 to 7 STI Cases per project

**49 STI Cases Rated for Perceived Success**
Some of the 59 identified cases
were not able to be rated

**13 Different SW Projects**
Some ongoing, some just completed;
using Ada, C, Fortran, Assembly;
ranging in size from 2900 to 49200 SLOC

**21 Different SW Technologies**
Most new tools, methods, languages (e.g.,
CASE, 2167A, Ada, Rapid-Proto, Reuse,...)

# Software Technology Insertion: Success Factors

## 13 Software Projects

| Project ID# | Language | SLOC | Current Phase |
|---|---|---|---|
| 1 | Assembly<br>Fortran<br>C | 49200<br>6400<br>2900 | Integration Test |
| 2 | C | 9100 | Design & Code |
| 3 | Assembly<br>C | 4000<br>4500 | Maintenance |
| 4 | C | 8500 | Integration Test |
| 5 | Assembly | 10850 | Design & Code |
| 6 | Assembly | 7100 | System Test |
| 7 | Assembly | 16000 | Integration Test |
| 8 | Fortran | n/a | Cancelled |
| 9 | Ada | 25000 | Design |
| 10 | Ada | n/a | Integration Test |
| 11 | Assembly | 4850 | Integration Test |
| 12 | C | 13700 | Maintenance |
| 13 | Assembly | 18000 | Design & Code |

Software Technology Insertion: Success Factors

## 21 New Software Technologies
(most of them new tools, methods, languages)

**A** - The use of Ada® as an implementation language
**B** - In-House automated build tool(s)
**C** - In-House automated code documentation tool
**D** - In-House program design language (PDL) tools
**E** - In-House metrics tools for automatic data collection
**G** - In-House standard test reporting tool based on RDBMS
**I** - Workstation-based engineering documentation tool
**J** - The use of M68020 assembly language
**K** - Microprocessor Development Stations (MDS) for integration testing
**L** - In-House project scheduling and reporting tool
**M** - In-House configuration management (CM), source code control tool
**N** - In-House Vax/Unix documentation package using troff
**P** - In-House Software Problem Reporting Tool based on RDBMS
**R** - Rapid-Prototyping in requirements or design
**S** - Structured analysis graphical CASE tool
**T** - Structured analysis graphical CASE tool
**U** - Reusable Software in design or coding
**W** - The use of workstations as primary development platforms
**X** - Workstation-based engineering documentation tool
**Y** - In-House requirements traceability tool
**Z** - The use of DoD-STD-2167 or 2167A

UNCLASSIFIED

Missile Systems Division

Software Technology Insertion: Success Factors

**Raytheon**

Software Laboratory

Nov 28, 1990

# Project/Technology Matrix

New Technology ID

| Project ID# | A | B | C | D | E | G | I | J | K | L | M | N | P | R | S | T | U | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | ● |  |  | ● |  |  |  |  |  |  | ● |  | ● |  |  | ● |  |  | ● |  |  |
| 2 |  |  |  |  | ○ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 |  | ● |  | ○ |  |  |  |  |  |  | ● |  | ● | ● |  |  | ● |  |  |  |  |
| 4 |  |  |  |  |  | ● |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  | ○ |  |  | ○ |  |  |  |  |  | ● |  |  |  |  |  |  | ○ | ○ |
| 6 |  |  |  | ● |  |  |  |  |  | ○ |  | ○ | ● |  |  |  |  |  | ● |  | ● |
| 7 |  |  | ● | ● |  |  | ● |  |  |  |  |  | ● |  | ● |  |  |  |  |  | ● |
| 8 |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ● |  |  |  |  |  |  |
| 9 | ● |  |  |  | ● |  |  |  |  | ● |  |  |  |  |  |  | ● |  |  |  | ● |
| 10 | ● |  |  | ● |  |  | ● |  |  |  |  |  | ● |  | ● |  |  | ● |  |  | ● |
| 11 |  |  |  | ● |  |  |  | ● | ● |  | ● |  |  | ● |  |  |  |  |  | ● | ● |
| 12 |  |  |  |  |  |  |  | ● | ● |  |  |  |  | ● |  |  |  |  |  |  |  |
| 13 | ○ |  |  |  |  |  | ● |  |  |  |  |  |  | ○ | ● |  |  |  |  |  | ● |

59 STI Cases: ● Rated ○ Not Rated

## Other Measured Factors

- **Technology Type** (Competence-Enhancing or -Destroying)
- **Support Type** (In-House or External)
- **Maturity** of the Technology (Young, Mature, and Old)
- **Project Size** (SLOC)
- **Prior Expectations** (for success or failure)
- **Reasons** (for STI choice)
- **Methods** (of STI insertion)
- Perception of **Time** Savings
- Perception of **Labor** Savings
- Perception of **Computer** Cost Savings
- Perception of **Quality** Improvement
- **Result vs. Prior Expectations** (for success or failure)

## Measuring Perceived STI Success

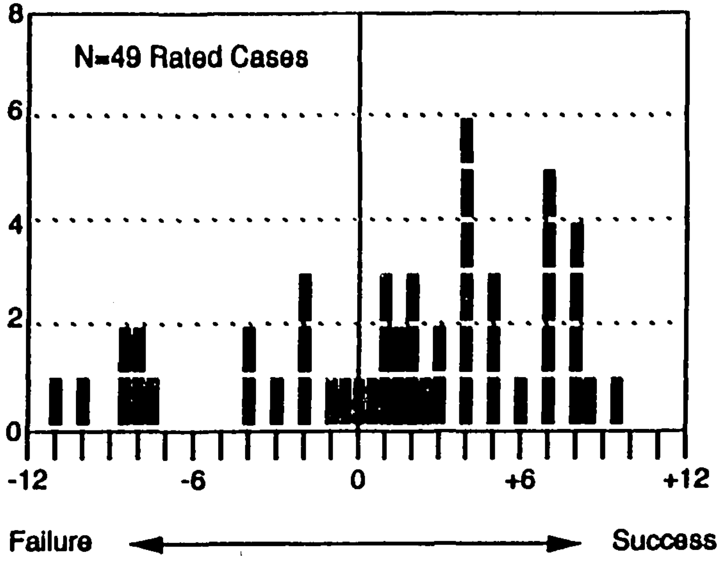
- For each STI Case, **6 Questions** were asked of:
  (1) Lead Engineer (Project/Matrix)
  (2) Dept Manager (Functional/Matrix)

| ***For Each STI Case:*** | **Agree.....Disagree** | | | | |
|---|---|---|---|---|---|
| **Statement** (Agree or Disagree?) | +2 | +1 | 0 | -1 | -2 |
| 1. I would use the new method/tool again | | ✓ | | | |
| 2. The new method/tool saved **schedule time** | | | ✓ | | |
| 3. The new method/tool saved **labor cost** | ✓ | | | | |
| 4. The new method/tool saved **computer cost** | | | | | ✓ |
| 5. The new method/tool **improved quality** | | ✓ | | | |
| 6. The new method/tool met my expectations | | | | ✓ | |

- **Total Rating** for each STI Case is **sum** (example =+1)
  i.e., maximum = +12, minimum = -12

(Note: Questions not weighted)

## Histogram Of Software Technology Insertion Cases



R. Lydon
Raytheon
Page 17 of 24

UNCLASSIFIED

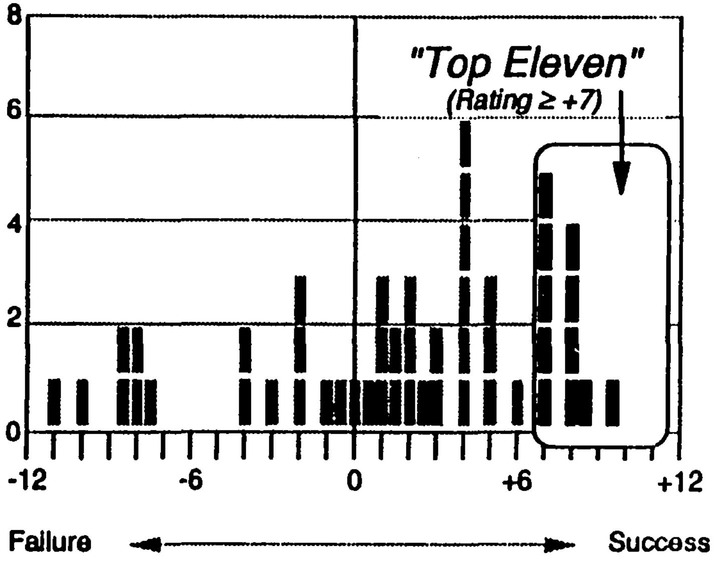
Missile Systems Division

# Software Technology Insertion: Success Factors

**Raytheon**

Software Laboratory

Nov 28, 1990



## Top Eleven STI Cases

**Main reasons for "success":**

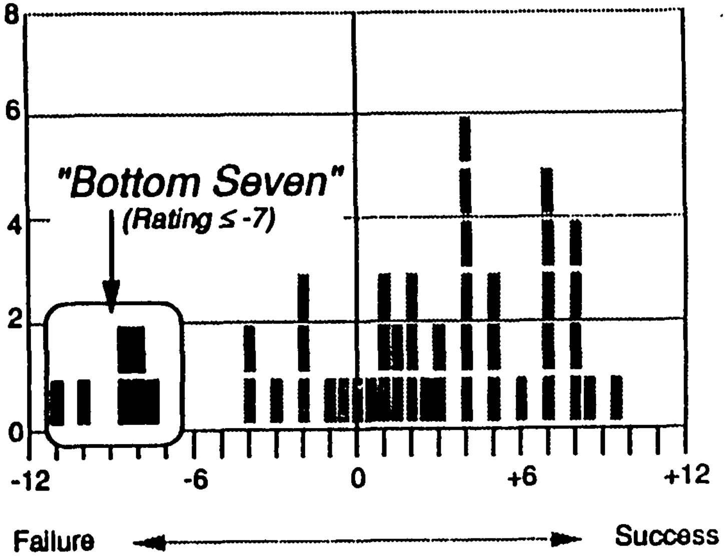
- "Synergy" within a project (4)
- Critical need for a capability (3)
- "Synergy" between two techr.ologies (2)
- Mature and powerful tool (1)

May or may not **"Save Computer Costs"** (+0.2)

**"Met Expectations"** (+0.5) not as critical as:

- "Save Time" (+1.8)
- "Save Labor" (+1.7)
- "Improve Quality" (+1.6)

# Software Technology Insertion: Success Factors



## Bottom Seven STI Cases

**Main reasons for "failure":**

- Immature technology (3)
- Interface problems (3)
- Technology not "needed" by LE (2)
- Wrong technical solution (1)

May or may not **"Improve Quality"** (-0.4)

**"Save Computer Costs"** (-1.1) not as critical as:

- "Save Time" (-1.8)
- "Save Labor" (-1.9)
- "Met Expectations" (-1.9)

UNCLASSIFIED
Missile Systems Division

**Raytheon**

## Software Technology Insertion: Success Factors

Software Laboratory

Nov 28, 1990

### **Competence-Enhancing** vs **Competence-Destroying**

**Competence-Enhancing** technology - major improvement in price/performance that builds on existing know-how; a substitute for older technology, but does not render old skills obsolete; increase efficiency.

**Competence-Destroying** technology - new way of making a given product; requires new skills, abilities, and knowledge for use; may combine previously discrete steps into continuous flow, or be a completely different process

### Maturity of a New Software Technology

**Young** technology - Released < 1 year, or prior to 2nd major release (V1.x)

**Mature** technology - Released > 1 year, and after 2nd major release (V2.x+)

**Old** technology - Released > 5 years, or after end of formal support

UNCLASSIFIED
Missile Systems Division

Raytheon

# Software Technology Insertion: Success Factors

Software Laboratory

Nov 28, 1990

## Ratings of New Technology Types Across Two Factors

| | IN-HOUSE Support | OUTSIDE Support | |
|---|---|---|---|
| **Competence ENHANCING** ("incremental") | #Total= 16<br>#Rated= 13<br>Tot Rating= 47.0<br>Median= +5.0<br>Mean Rating= +3.6<br>BEST | #Total= 9<br>#Rated= 8<br>Tot Rating= 11.5<br>Median= +4.0<br>Mean Rating= +1.4<br>OK | **Mean = +2.8** |
| **Competence DESTROYING** ("radical") | #Total= 11<br>#Rated= 8<br>Tot Rating= -2.0<br>Median= +1.5<br>Mean Rating= -0.2<br>Poor | #Total= 23<br>#Rated= 20<br>Tot Rating= 14.5<br>Median= +0.7<br>Mean Rating= +0.7<br>Marginal | **Mean = +0.4** |
| | **Mean = +2.1** | **Mean = +0.9** | |

**RATING SCALE:** +12 = Maximum Success, -12 = Maximum Failure

UNCLASSIFIED

Missile Systems Division

# Software Technology Insertion: Success Factors

**Raytheon**

Software Laboratory

Nov 28, 1990

## Summary of Results

*(Focus on success **factors** rather than successful **technologies**)*
*(Focus on **perceived** rather than **real** STI success)*

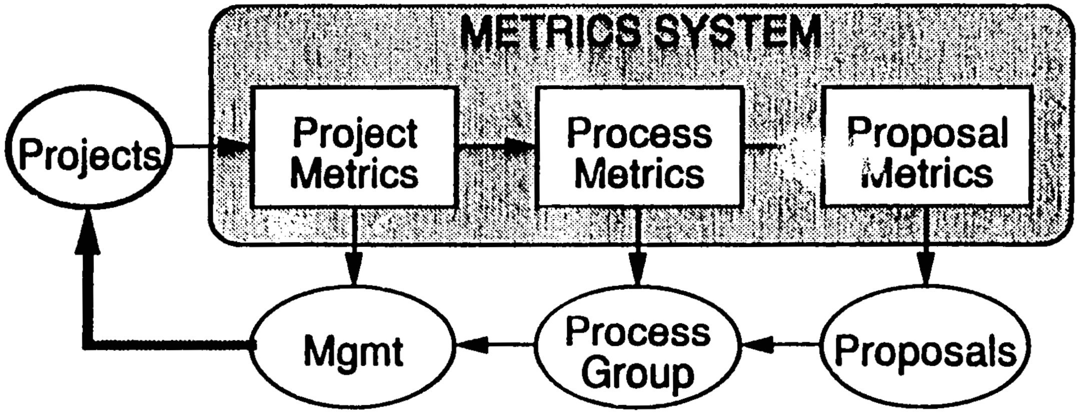
- Saving **schedule time** and **labor costs** drive successful STI (obvious?)
- **Improving quality** is necessary, but not sufficient for successful STI
- Exceeding users' expectations not necessary for successful STI, but **not meeting expectations** is sufficient for failure (i.e., must control)
- Much greater success for **competence-enhancing** vs competence-destroying technologies
- Greater success for **mature** vs young or old technologies
- Somewhat greater success for **in-house** vs outside supported

Software Technology Insertion: Success Factors

## Next Step:
## Linking **Perceived Success** with **Real Success** via **Software Metrics Collection**

- Corporate-wide effort to implement automatic collection of software metrics as a by-product of development - **MSD** is Lead Division
- **10** current software metrics defined (similar to Mitre Metrics)
- Based mainly on **DoD-STD-2167A**
- AutoCollection in development for both **project-specific** and **process-level** (across multiple projects) software metrics

## Overview of Raytheon MSD's Software Metrics Collection



# SESSION 3—MEASUREMENT

W. E. Royce, TRW

R. E. Loesh, NASA/JPL

W. W. Agresti, MITRE

N92

19427

UNCLAS

N92-19427

63/20

p-31

# Pragmatic Quality Metrics For Evolutionary Software Development Models

Walker Royce
TRW Space and Defense Sector
Redondo Beach California

T60126

## ABSTRACT

Due to the large number of product, project and people parameters which impact large custom software development efforts, measurement of software product quality is a complex undertaking. Furthermore, the absolute perspective from which quality is measured (customer satisfaction) is intangible. While we probably can't say what the absolute quality of a software product is, we can determine the relative quality, the adequacy of this quality with respect to pragmatic considerations, and identify good and bad trends during development. While no two software engineers will ever agree on an optimum definition of software quality, they will agree that the most important perspective of software quality is its ease of change. We can call this flexibility, adaptability or some other vague term, but the critical characteristic of software is that it is *soft*. The easier the product is to modify, the easier it is to achieve any other software quality perspective.

This paper presents objective quality metrics derived from consistent lifecycle perspectives of *rework* which, when used in concert with an evolutionary development approach, can provide useful insight to produce better quality per unit cost/schedule or to achieve adequate quality more efficiently. The usefulness of these metrics is evaluated by applying them to a large, real world, Ada project (CCPDS-R).

These measures can be automated, consistent, and easy to use. Along with subjective interpretation to account for the lifecycle context, objective insight into product quality can be achieved early where correction or improvement can be instigated more efficiently.

*Index Terms*- Evolutionary Development, Software Quality Metrics, Ada, Maintainability, Process Improvement.

## BACKGROUND

There have been many attempts to define measures of software quality in the past 20 years. For many reasons, none of these has caught on as accepted practice in the software industry. [2] discusses many of the problems and tradeoffs associated with defining and measuring software quality. One of the recurring themes in this work was the need for subjectivity and expensive human resources in both the collection and interpretation of quality metrics. Furthermore, the concept of a technology independent set of metrics, although an acknowledged desire, was not well understood. [8] provides an excellent discussion of the need for objective, measurable software quality metrics which remain technology independent. [9] defines a complete company metrics program with actual data that provides some valuable experience and lessons learned. [10] describes the most current motivation for measuring software quality, process improvement.

After three years of successful software development on the Command Center Processing and Display System - Replacement (CCPDS-R) project using modern Ada software engineering techniques ([12], [13] and [15]), TRW has derived a subset of software quality metrics which are measurable, objective, and useful in providing a basis for improving downstream quality of products and processes. One of the problems with typical government contracted systems like CCPDS-R is that most are one of a kind projects. This characteristic provides added complexity to measurement since the experience may be only partially useful between different project domains. The metrics presented herein have been formulated to be as useful as possible while remaining relatively domain independent so that comparisons between different projects are possible. This is not as simple as it sounds and the literature on software quality metrics reinforces this experience. After many iterations, the data presented herein has demonstrated objective and valuable insight in its application to CCPDS-R and it provides a credible basis from which better metrics can be derivea.

Software Quality Metrics Objectives. Software quality metrics should be simple, easy to use, and hard to misuse. They should be useful to project management, stimulate continuous improvement of our development process, and low cost to administer consistently across different projects.

Usefulness. Conventional testing techniques exist for assessing the *functionality*, *reliability* and *performance* of a software product, however, there are no accepted methods for assessing its flexibility (*modularity*, *changeability*, *or maintainability*). While there are many other perspectives of quality (e.g., portability, interoperability, etc.), our experience in executing an evolutionary development process has demonstrated that its flexibility aspects are the most important. The easier the product is to modify, the easier it is to achieve any other software quality perspective except perhaps performance. The tradeoff between flexibility and performance is highly dependent on the application domain as well as many other architectural issues and for the purposes of this discussion we will assume that performance is achieved through proper hardware selection and that the project is prioritized "software first". A project which is prioritized more towards performance (i.e., 1750A flight program), may not interpret these metrics in the same fashion as a project prioritized towards continuous lifecycle modification (i.e., ground based $C^3$ System). This paper will attempt to provide useful, objective definitions for modularity, changeability and maintainability. The intent of this metrics program is to provide a mechanism for quantifying both end-product quality as well as in-progress development trends toward achieving that quality.

Development Language. Ada has proven to support increased quality and the evolutionary process model in large software development efforts. Furthermore, Ada appears to be the language of choice for the majority of current and future large government projects. While this paper assumes that Ada is the language for design and implementation of software development projects which use these software quality metrics, it should be straightforward to adapt this approach to other languages through a suitable redefinition of a Source Line of Code (SLOC).

Development Approach. An evolutionary development approach as prescribed in the Ada Process Model [12] is necessary to maximize the usefulness of these metrics across a broader range of the life cycle. The metrics are derived from controlled configuration baselines. Therefore, an approach with early incremental baselines will see an increased benefit. As a prerequisite to understanding the derivation of the software quality metrics, the following section provides an overview of the Ada Process Model employed on CCPDS-R.

## Ada PROCESS MODEL

An Evolutionary Process Model is fundamental to this approach for Software Quality Assessment. Without tangible intermediate products, software quality assessment would be ineffective and inaccurate. Conventional experience has repeatedly seen projects sequence through highly successful preliminary and critical design phases (as perceived by conventional Design Review assessment of design quality) only to have the true quality problems surface in the integration and test phases with little or no time for proper resolution. An Evolutionary Process Model provides a systematic approach for achieving early insight into product quality and a uniform lifecycle measure for its assessment. It also avoids the inevitable degradations in quality due to late breakage and rapid fixes which are shoehorned into the product without adequate software engineering.

TRW's Ada Process Model is, in simplest terms, a uniform application of incremental Ada product evolution coupled with a demonstration-based approach to design review for continuous and insightful thread testing and risk management. The techniques employed within this process are derived from the philosophy of the Spiral Model [7] with emphasis on an evolutionary design approach. The use of Ada as the life cycle language for design evolution provides the vehicle for uniformity and provides a basis for consistent software progress and quality metrics.

TRW's Ada Process Model recognizes that all large, complex software systems will suffer from design breakage due to early unknowns. It strives to accelerate the resolution of unknowns and correction of design flaws in a systematic fashion which permits prioritized management of risks. *The dominant mechanism for achieving this goal is a disciplined approach to incremental development.* The key strategies inherent in this approach are directly aimed at the three main contributors to software diseconomy of scale: minimizing the overhead and inaccuracy of interpersonal communications, eliminating rework, and converging requirements stability as quickly as possible in the lifecycle. These objectives are achieved by:

1. requiring continuous and early convergence of individual solutions in a homogeneous life cycle language (Ada).

2. eliminating ambiguities and unknowns in the problem statement and the evolving solution as rapidly as practical through prioritized development of tangible increments of capability.

Although many of the disciplines and techniques presented herein can be applied to non-Ada projects, the expressiveness of Ada as a design and implementation language and support for partial implementation (abstraction) provide a strong platform for creating a uniform approach.

Many of the Ada Process Model strategies (summarized in Figure 1) have been attempted, *in part*, on other software development efforts; however, there are fundamental differences in this approach compared to conventional software development models.

| Process Model Strategy | | Conventional Counterpart |
|---|---|---|
| Uniform Ada Lifecycle Representation | ⟹ | PDL/HOL |
| Incremental Development | ⟹ | Monolithic Development |
| Design Integration | ⟹ | Integration and Test |
| Demonstration Based Design Review | ⟹ | Documentation Based Design Review |
| Total Quality Management | ⟹ | Quality by Inspection |

Figure 1: New Techniques vs. Conventional Techniques

Uniform Ada Lifecycle Representation. The primary innovation in the Ada Process Model is the use of a single language for the entire software lifecycle, including, to some degree, the requirements phase. All of the remaining techniques rely on the ability to equate design with code so that the only variable during development is the level of abstraction. This provides two essential benefits:

1. *The ability to quantify units of software (design/development/test) work in* one dimension, *Source Lines of Code (SLOC).* While it is certainly true that SLOC is not a perfect absolute measure of software, with consistent counting rules, it has proven to be the best normalized measure and does provide an objective, consistent basis for assessing relative trends across the project life cycle.

2. *A formal syntax and semantics for lifecycle representation with automated verification by an Ada compiler.* Ada compilation does not provide complete verification of a component. It does go a long way, however, in verifying configuration consistency, and ensuring a standard, unambiguous representation.

Incremental Development. Although risk management through incremental development is emphasized as a key strategy of the Ada Process Model, it was (or always should have been) a key part of most conventional models. Without a uniform lifecycle language as a vehicle for incremental design/code/test, conventional implementations of incremental development were difficult to manage. This management is simplified by the integrated techniques of the Ada Process Model.

Design Integration. In this discussion, we will take a simple minded view of "design" as the structural implementation or partitioning of software components (in terms of function and performance) and definition of their interfaces. At the highest level of design we could be talking about conventional requirements definition, at the lowest level, we are talking about conventional detailed design and coding. Implementation is then the development of these components to meet their interfaces while providing the necessary functional performance. *Regardless of level, the activity being performed is Ada coding.* Top level design means coding the top level components (Ada main programs, task executives, global types, global objects, top-level library units , etc.). Lower level design means coding the lower level program unit specifications and bodies.

The postponement of all coding until after CDR in conventional software development approaches also postponed the primary indicator of design quality: integrability of the interfaces. The Ada Process Model requires the early development of a Software Architecture Skeleton (SAS) as a vehicle for early interface definition. The SAS essentially corresponds to coding the top level components and their interfaces, compiling them, and providing adequate drivers/stubs so that they can be executed. This early development forces early baselining of the software interfaces to best effect smooth evolution, early evaluation of design quality and avoidance of downstream breakage. In this process, we have made integration a design activity rather than a test activity. To a large degree, the Ada language forces integration through its library rules and consistency of compiled components. It also supports the concept of separating structural definition (specifications) from runtime function (bodies). The Ada Process Model expands this concept further by requiring structural design (SAS) *prior to* runtime function (executable threads). Demonstrations provide a forcing function for broader runtime integration to augment the compile time integration enforced by the Ada language.

Demonstration Based Design Review. Many conventional projects built demonstrations or benchmarks of standalone design issues (e.g., user system interface, critical algorithms, etc.) to support design feasibility. However, the design baseline was represented on paper (PDL, simulations, flowcharts, vugraphs). These representations were vague, ambiguous and not amenable to configuration control. The degree of freedom in the design representations made it very difficult to uncover design flaws of substance, especially for complex systems with concurrent processing. Given the typical design review attitude that a design is "innocent until proven guilty", it was quite easy to assert that the design was adequate. This was primarily due to the lack of a tangible design representation from which true design flaws were unambiguously obvious. Under the Ada Process Model, design review demonstrations provide some proof of innocence and are far more efficient at identifying and resolving design flaws. The subject of the design review is not only a briefing which describes the design in human understandable terms, but also a demonstration of important aspects of the design *baseline* which verify design quality (or lack of quality).

Total Quality Management (TQM). In the Ada Process Model there are two key advantages for applying TQM. The first is the common Ada format throughout the lifecycle which permits consistent software metrics across the software development work force. Although these metrics don't all pertain to quality (many pertain to progress), they do permit a uniform communications vehicle for achieving the desired quality in an efficient manner. Secondly, the demonstrations serve to provide a common goal for the software developers. This "integrated product" is a reflection of the complete design at various phases in the life cycle for which all personnel have ownership. Rather than individually evaluating components which are owned by individuals, the demonstrations provide a mechanism for reviewing the team's product. This team ownership of the demonstrations is an important motivation for instilling a TQM attitude.

## SOFTWARE QUALITY METRICS APPROACH

In essence, the approach we are taking is similar to that of [8] who proposes to measure software quality through the *absence of spoilage.* While his definitions are purposely vague (to remain technology and project independent), ours are quite explicit. The key to this metrics approach is similar to conventional cost estimation techniques such as COCOMO [3] where quantifiability and consistency of application are important. Note that software cost estimation has subjective inputs and objective outputs. Our approach will define objective inputs which may require subjective interpretation for project context.

Our primary metric for software quality will be rework as measured by changed SLOC in configured baselines. This metric will also need to be adjusted for project context to accommodate the product characteristics, the life cycle phase, etc. The software quality assessment derived from this objective collection of rework metrics will require subjective analysis in some cases. The subjectivity here is in the fact that we are trying to assess quality during development (this requires subjective analysis) using the same metrics used to assess quality following development (objective analysis). For example, the volume of rework following product delivery is an objective measure of quality, or lack of quality. The amount of rework following the first configuration baseline during development is a subjective measure. Zero rework might be interpreted as a perfect baseline (unlikely), an inadequate test program, or an unambitious first build. The following paragraphs define some of the foundations in this approach:

Software Quality Definition. *Software quality is the degree of compliance with the customer expectations of function, performance, cost and schedule.* This is an incredibly difficult concept to make objective. The only mechanisms available for defining "customer expectations" are Software Requirements Specifications for function and performance, and an approved expenditure plan which quantifies cost and schedule goals (basically, this corresponds to the "contract"). These two mechanisms are traditionally the lowest quality products produced by a project since they are required to be agreed upon with numerous unknowns far too early in the lifecycle. The evolutionary process model and software quality metrics should provide better insight into the degree of compliance with customer expectations in the above four perspectives.

Software Change Order (SCO). A Software Change Order constitutes direction to proceed with changing a configured software component. This change may be needed to 1) rework a component with bad quality (a fix), or 2) rework a component to achieve better quality (an enhancement) or 3) accommodate a customer directed change in requirements. The difference between the first two types of rework is inherent in the *necessity* for the change. If the change is *required* for compliance with product specifications, then the rework is type 1. If the change is *desired* for cost-effectiveness, increased testability, increased usability, or other efficiency reasons (assuming the unchanged component is compliant), then the rework is type 2. In both cases, the rework should result in increased end product quality (requirements compliance per dollar), however, type 1 also indicates inadequate quality in a current baseline. In practice, differentiating between type 1 and type 2 may be quite subjective. As discussed later, most of the metrics are insensitive to the categorization, but if the differentiation is consistently applied, it can provide useful insight. Conventionally, SCOs were called Software Problem Reports (SPRs). To avoid confusion ("problem" has a negative connotation, and not all changes are necessarily problems), we have changed the terminology. The software quality metrics collection and analysis will use type 1 and type 2 SCOs in an appropriate manner. Type 3 SCOs need to be separated since they do not reflect any change in quality, they do however, redefine the customer expectations. Furthermore, Type 3 SCOs typically reflect a change which is of more global impact thereby requiring various levels of software and system engineering as well as high level regression testing. These types of SCOs will not be used in these metrics due to this wide range of variability. Rather, the data derived from type 1 and type 2 SCOs should provide a solid basis for estimating maintainability and the effort required for type 3 SCOs.

Source Lines of Code (SLOC). There has always been a controversy as to whether SLOC provides a good metric for measuring software volume (DeMarco calls this *bang*). [11] identifies some of the precautions necessary when dealing with SLOC. Upon reading open literature which discusses project productivities (SLOC/MM), it is easy to see that there is little, if any, comparability between projects within the same company no less projects from different companies. [4] identifies the pros and cons of various measures and comes to the conclusion that there is nothing better. Everyone agrees however, that whatever one uses, it must be defined objectively and consistently to be of value for comparison. How we define the absolute unit of SLOC is not as important as defining it consistently across all projects and all areas of a specific project. Therefore, the preferred way to define a SLOC is the following:

> The number of SLOC for a given set of Ada program units is defined as the output of a SLOC Counting Tool.

Enforcing this definition is simple to achieve by providing a portable tool. By accepting certain noncontroversial and simple standards for program unit headers and program layout the tool can provide more valuable outputs than simply SLOC counts (e.g., static hierarchies, and complexity ratings).

Ada/COCOMO [5], [6] defines SLOC for Ada programs as: Within an Ada specification part, each carriage return counts as one SLOC. Specifications shall be coded with the following standards (rationale is provided in *italics*):

1. each parameter of a subprogram declaration be listed on a separate line (*The design of a subprogram interface is done in one place and generally the effort associated with the interface design is dependent on the number of parameters.*)

2. for custom enumeration types (e.g., system state, socket names, etc.) and record types each enumeration or field should be listed on a separate line. (*Custom types usually involve custom design and engineering, hence an increased number of SLOC.*)

3. for predefined enumeration types (e.g., keyboard keys, compass directions), enumerations should be listed on as few lines as possible without loss of readability. (*These kinds of types generally require no custom engineering.*)

4. Initialization of composite objects (e.g., records or arrays) should be listed with one component per line. (*Frequently, each of these assignments represents a custom statement, an others clause is typically used for the non-custom assignments.*)

Within Ada bodies each semi-colon counts as one SLOC. Generic instantiations count one line for each generic parameter (spec or body).

The definition above treats declarative (specification) design much more sensitively than it does executable (body) design. It also does not recognise the declarative part of a body as the same importance as a specification part. Although these and other debates can surface with respect to the "optimum" definition of a SLOC, the optimum *absolute* definition is far less important than a consistent *relative* definition.

Quality Control Board. The QCB constitutes the governing body responsible for authorizing changes to a configured baseline product (conventionally known as a configuration control board - CCB). This body is composed, at a minimum, of the development manager, customer representative, each product manager, systems effectiveness representative and the test manager. The QCB decides on all proposed changes to configured products and approves all SCOs. The QCB is responsible for collecting the Software Quality metrics, objectively and subjectively analyzing trends, and proposing changes to the development process, tools, products or personnel to improve future quality.

Configured Baseline. A configured baseline constitutes a set of products which are subjected to change control through a Quality Control Board (QCB). Configured baselines usually represent intermediate products which have completed design, development, and informal test and final products which have completed formal test.

## METRICS DERIVATION

The remainder of this paper provides substantial detail in the definition and description of the necessary statistics to be collected, the metrics derived from these statistics and their interpretation. This section provides a simple overview of how these metrics were derived, the necessity of some of the collected statistics and their raison d'être. The following derivations are not an obvious top down progression, rather, they resulted from substantial trial and error, numerous dead end analyses, intuition and heuristics.

The fundamental hypothesis was that their was significant information content in the character of rework being performed over the project lifecycle. The obvious raw statistics to collect include number and type of software changes, SLOC damaged, and SLOC fixed. The problem was to find the right filtering techniques for the raw rework statistics which identify useful trends and to uncover objective measures which quantify product attributes both during development and as an end-product. Our original intent was to provide a quantification of the product's modularity, changeability, and maintainability. The first two are intuitively simple to define as a function of rework, the third is more subtle:

**Modularity** ($Q_{mod}$): The average extent of breakage. This identifies the need to quantify *extent of breakage* (we will use volume of SLOC damaged) and number of instances of rework (Number of SCOs). In effect we are defining modularity as a measure of breakage localization.

**Changeability** ($Q_C$): The average complexity of breakage. This identifies the need to quantify *complexity of breakage* (we will use effort required to resolve) and number of instances of rework (Number of SCOs).

**Maintainability** ($Q_M$): Theoretically the maintainability of a product is related to the productivity with which the maintenance team can operate. Productivities however, are so difficult to compare between projects that this definition was intuitively unsatisfying. If we ratio the productivity of rework to the productivity of development, we end up with a value which is independent of productivity but yet a reflection of the complexity to change a product in relation to the complexity to develop it. This normalises out the project productivity differences and provides a relatively comparable metric. Maintainability then, will be defined as the ratio of rework productivity and development productivity. Intuitively, this value identifies a product which can be changed three times as efficiently ($Q_M = .33$) as it was developed as having a better (lower) maintainability than a product that can be changed twice as efficiently ($Q_M = .5$) as it was developed, independent of the absolute maintenance productivity realised. The statistics needed to compute these values are the total development effort, total SLOC, total rework effort and total reworked SLOC.

While the values above provide useful end-product objective measures, their intermediate values as a function of time would also provide insight during the development process into the expected end-product values. Furthermore, once we have gained some experience with maintenance of early increments, this experience should be useful for predicting the rework inherent in remaining increments.

The above brief derivation is starting to push the limits of our first goal (simplicity) and the following sections, on the surface, will appear to be somewhat complex. A few remarks about this are in order. First, there will always be a tradeoff between simplicity and real insight. Surface insight is usually attained very simply, detailed insight requires added knowledge and complexity. We have chosen a set of metrics which range from simple to moderately complex to cover the multiple perspectives needed by project management to ensure accuracy. It is not necessary to deal with these metrics as a complete set. Subsets, or different sets are also useful. Secondly, most of the analysis, mathematics and data collection inherent in these metrics should be automated so that managers need only interpret the results and understand their basis.

The above values were determined through extensive analysis, trial and error, and intuition. There are certainly other metrics derivable from rework statistics which would also provide useful insight. The following sections provide more detailed descriptions and notations for the collected statistics (Table 1), in-progress indicators (Table 2), and end-product quality metrics (Table 3). Hypothetical expectations are provided in Figure 2 for the in-progress indicators and collected statistics.

## Collected Statistics

Table 1 identifies the necessary statistics which must be collected over the lifecycle to implement our proposed metrics.

**Total Source Lines** The $SLOC_T$ metric tracks the estimated total size of the product under development. This value may change significantly over the life of the development as early requirements unknowns are resolved and as design solutions mature. This total should also include reused software which is part of the delivered product and subject to contractor maintenance.

**Configured SLOC** This metric simply tracks the transition of software components from a maturing design state into a controlled configuration. For any given project, this metric will provide insight into progress and stability of the design/development team. [12] discusses some of the tradeoffs and risk management philosophy inherent in laying out an incremental build approach. For projects with reused software, there will be an early contribution to $SLOC_C$ and thus "immediate progress" and quality metrics as defined below.

**Errors** Real errors (type 1 SCOs) constitute an important metric from which many of the following are derived. The expectation is that the highest incidence of uncovering errors happens immediately after the turnover and decreases with time (i.e., the software matures).

**Improvements** The other stimulus for changing a baseline, improvements (type 2 SCOs), are also key to the assessment of quality and progress towards producing quality. The expectation for improvements is approximately inversely proportional to errors, in that as the error rate starts off high and damps out, the improvements start off low (the focus is on errors) and increase. This phenomenon is basically derived from the assumption that a fixed team is working the Test/Maintenance program and:

| Statistic | Definition | Insight |
|---|---|---|
| Total Source Lines | $SLOC_T$ = Total Product SLOC | Total Effort |
| Configured Source Lines | $SLOC_C$ = Standalone Tested SLOC | Demonstrable Progress |
| Errors | $SCO_1^o$ = No. of Open Type 1 SCOs<br>$SCO_1^c$ = No. of Closed Type 1 SCOs<br>$SCO_1$ = No. of Type 1 SCOs | Test Effectiveness<br>Test Progress<br>Reliability |
| Improvements | $SCO_2^o$ = No. of Open Type 2 SCOs<br>$SCO_2^c$ = No. of Closed Type 2 SCOs<br>$SCO_2$ = No. of Type 2 SCOs | Value Engineering<br>Design Progress |
| Open Rework | $B_1$ = Damaged SLOC Due to $SCO_1^o$<br>$B_2$ =Damaged SLOC Due to $SCO_2^o$ | Fragility<br>Schedule Risks |
| Closed Rework | $F_1$ =SLOC Repaired after $SCO_1^c$<br>$F_2$ =SLOC Repaired after $SCO_2^c$ | Maturity<br>Changeability |
| Total Rework | $R_1 = F_1 + B_1$<br>$R_2 = F_2 + B_2$ | Design Quality<br>Maintainability |

Table 1: Collected Raw Data Definitions

$$Effort_{Errors} + Effort_{Improvements} = Constant$$

The actual differentiation between Type 1 and Type 2 is somewhat subjective. The metrics defined herein are not particularly sensitive to either type since they rely on the sum of the impacts from both types. However, the difference between Type 1 damage and Type 2 damage may provide useful insight as demonstrated on CCPDS-R.

**Open Rework** Theoretically, all rework corresponds to an increase in quality. Either the rework is necessary to remove an instance of "bad" quality ($SCO_1$), or to enhance a component for life cycle cost effectiveness ($SCO_2$). The dynamics of the rework coupled with the project schedule context must be evaluated to provide an accurate assessment of quality trends. A certain amount of rework is a necessity in a large software engineering effort. In fact, early rework is considered a sign of healthy progress in the evolutionary process model. Continuous rework, late rework, or zero rework due to the non-existence of a configured baseline are generally indicators of negative quality. Interpretation of this metric requires project context. In general however, the rework must ultimately go to zero at product delivery. In order to provide a consistent and automatable collection process, rework is defined as the number of SLOC *estimated* to change due to an SCO. The absolute accuracy of the estimates is generally unimportant and since open rework is tracked with an estimate and closed rework (see below) is tracked separately with actuals, the values continually correct themselves and remain consistent.

**Closed Rework** Whereas the breakage metrics estimated the damage done, the repair metrics should identify the actual damage which was fixed. Upon resolution, the corresponding breakage estimate should be updated to reflect the actual required repair that remains in the baseline. The actual SLOC *fixed* will clearly never be absolutely accurate. It will, however, be relatively accurate for assessing trends inherent in these metrics. Since *fixed* can take on several different meanings depending on what is added, deleted and changed, a consistent set of guidelines is necessary. Changed SLOC will increase $R_1$ without a change to $SLOC_C$. Added code will increase $R_1$ and $SLOC_C$, although not necessarily in the same proportion. Deleted code (not typically a problem) with no corresponding addition could reduce both $R_1$ and $SLOC_C$. A conventional differences tool with an appropriate preprocessor which converts properly formatted source files into a format which contains no comments and 1 SLOC per compared record would be the best method for computing changed SLOC. A simpler method (and the one used here) would be to simply estimate the magnitude of the fixed SLOC. Given the volume of changes and the need for only roughly accurate data for identifying trends, the accuracy of the raw data is relatively unimportant.

### In-Progress Indicators

Table 2 defines the in-progress indicators and Figure 2 identifies relative expectations. It is difficult to define the absolute expectations for the in-progress metrics without comparable data from other projects. Relative expectations are described in the following paragraphs.

| Indicator | Definition | Insight |
|---|---|---|
| Rework Ratio | $RR = \frac{R_1+R_2}{SLOC_C}$ | Future Rework |
| Rework Backlog | $BB = \frac{B_1+B_2}{SLOC_C}$ | Open Rework |
| Rework Stability | $SS = (R_1 + R_2) - (F_1 + F_2)$ | Rework Trends |

Table 2: In Progress Indicator Definitions

**Rework Ratio** The sum of the currently broken product $(B_1 + B_2)$ and the already repaired breakage $(F_1 + F_2)$ corresponds to the mass of the current product baseline which has needed rework $(R_1 + R_2)$. The rework ratio (RR) identifies the current ratio of $SLOC_C$ which is expected to undergo rework prior to maturity into an end product. The expectation for RR shown in Figure 2 is to increase to a stable value with minor discontinuities following the initial delivery of each build.

**Rework Backlog** The current backlog of rework is defined as the percentage of the current $SLOC_C$ which is currently in need of repair. In general, one would expect that the rework backlog should rise to some level and remain stable through the test program until it drops off to zero. Large changes from month to month should clearly be investigated.

**Rework Stability** The difference between *total* rework and closed rework provides insight into the trends of resolving issues. The important use of this metric is to ensure that the breakage rate is not outrunning the resolution rate. Figure 2 identifies an idealised case where the resolution rate does not diverge (except for short periods of time) from the breakage rate. Note also that the breakage rate somewhat tracks the $SLOC_C$ delivery rate. A diverging value of SS would indicate instability of rework activities. A stable value of SS would indicate systematic and straightforward resolution activities.

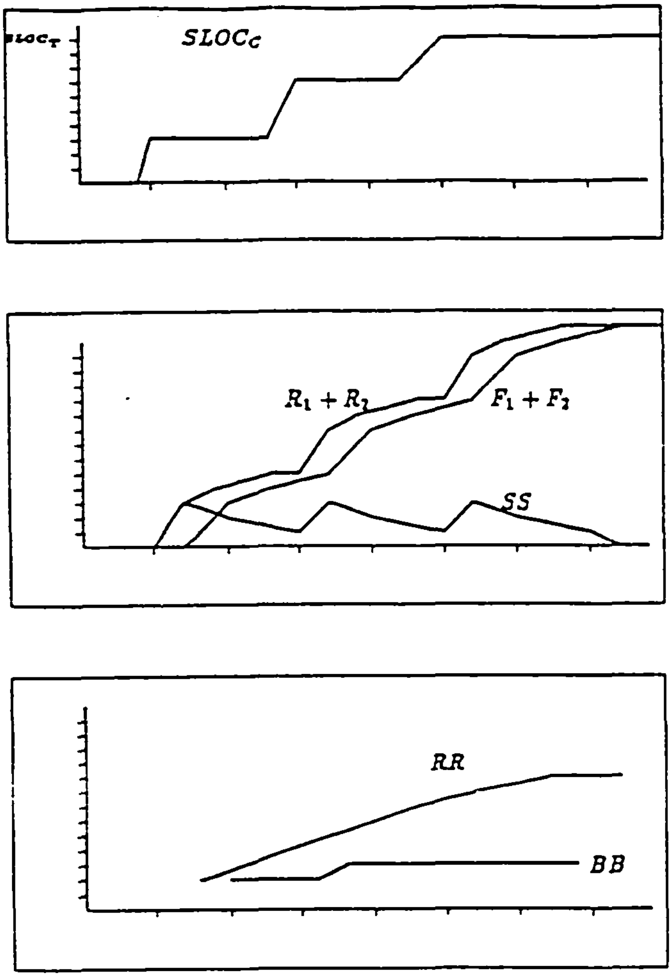


Figure 2: In-Progress Indicators Example Expectations

## End-Product Quality Metrics

The end-product metrics reflect insight into the maintainability of the software products with respect to type 1 and type 2 SCOs. Type 3 SCOs are explicitly not included since they redefine the inherent target quality of the system and tend to require more global system and software engineering as well as some major re-verification of system level requirements. Since these types of changes are dealt with in extremely diverse ways by different customers and projects, they would tend to cloud the meanings and comparability of the data. However, the metrics data below should be very helpful in determining and planning the expected effort for implementing type 3 SCOs.

**Rework Proportions** The $R_E$ value identifies the percentage of effort spent in rework compared to the total effort. In essence, it probably provides the best indicator of productivity. The activities included in these efforts should only include the technical requirements, software engineering, design, development, and functional test. Higher level system engineering, management, configuration control, verification testing and higher level system testing should be excluded since these

| Metric | Definition | Insight |
|---|---|---|
| Rework Proportions | $R_E = \frac{Effort_{SCO_1}+Effort_{SCO_2}}{Effort_{Total}}$<br>$R_S = \frac{(R1+R2)_{Total}}{SLOC_{Total}}$ | Productivity<br>Rework<br>Project Efficiency |
| Modularity | $Q_{mod} = \frac{R_1+R_2}{SCO_1+SCO_2}$ | Rework Localization |
| Changeability | $Q_C = \frac{Effort_{SCO_1}+Effort_{SCO_2}}{SCO_1+SCO_2}$ | Risk of Modification |
| Maintainability | $Q_M = \frac{R_E}{R_S}$ | Change Productivity |

Table 3: End-Product Quality Metrics Definitions

activities tend to be more a function of the company, customer or project attributes independent of quality. The goal here is to normalize the widely varying bureaucratic activities out of the metrics. $R_S$ provides a value for comparing with similar projects, future increments, or future projects as well as other in progress analyses. Basically, it defines the proportion of the product which had to be reworked in its lifecycle. Note that the actual value could be greater than 100% .

**Modularity** This value identifies the average SLOC broken per SCO which reflects the inherent ability of the integrated product to localize the impact of change. To the maximum extent possible, QCBs should ensure that SCOs are written for single source changes.

**Changeability** This value provides some insight into the ease with which the products can be changed. While a low number of changes is generally a good indicator of a quality process, the magnitude of effort per change is sometimes even more important.

**Maintainability** This value identifies the relative cost of maintaining the product with respect to its development cost. For example, if $R_E = R_S$, one could conclude that the cost of modification is equivalent to the cost of development from scratch (not highly maintainable). A value of $Q_M$ much less than 1 would tend to indicate a very maintainable product, at least with respect to development cost. Since we would intuitively expect maintenance costs of a product to be proportional to its development cost, this ratio provides a fair normalization for comparison between different projects. Since the numerator of $Q_M$ is in terms of effort and its denominator is in terms of SLOC, it is a ratio of productivities (i.e., effort per SLOC). Some simple mathematical rearrangement will show that $Q_M$ is equivalent to:

$$Q_M = \frac{Productivity_{Maintenance}}{Productivity_{Development}}$$

**Expectations** It is difficult to define the expectations for the end-product metrics without comparable data from other projects. Now that we have solid data for CCPDS-R, we can form expectations for future increments of CCPDS-R as well as other projects.

The above descriptions identify idealized trends for these metrics. Undoubtedly, real project situations will not be ideal. Their differences from ideal, however, are important for management and customer to comprehend. Furthermore, the application of these metrics on project increments as well as the project as a whole, should be useful.

## APPLICATION RESULTS

Figure 3, Figure 4 and Table 4 provide the actual data to date for the CCPDS-R project. The Command Center Processing and Display System Replacement (CCPDS-R) project will provide display Information used during emergency conferences by the National Command Authorities; Chairman, Joint Chiefs of Staff; Commander in Chief North American Aerospace Command; Commander in Chief United States Space Command; Commander in Chief Strategic Air Command; and other nuclear capable Commanders in Chief. It is the missile warning element of the new Integrated Tactical Warning/Attack Assessment System developed by North American Aerospace Defense Command/Air Force Space Command.

The CCPDS-R project is being procured by Air Force Systems Command Headquarters Electronic Systems Division (ESD) at Hanscom AFB and was awarded to TRW Defense Systems Group in June 1987. TRW will build three subsystems. The first, identified as the Common Subsystem, is 30 months into development. The Common Subsystem consists of 350,000 source lines of Ada with a development schedule of 38 months. It will be a highly reliable, real-time distributed system with a sophisticated User Interface and stringent performance requirements implemented entirely in Ada. CCPDS-R Ada risks were originally a very serious concern. At the time of contract definition, Ada host and target environment, along with Ada trained personnel availability were questionable.

The data provided in this paper was collected by manually analysing 1500+ CCPDS-R SCOs maintained online and in hard copy notebooks. Most of the data defined in the previous section was available in the SCOs. Each problem description and resolution was evaluated to determine whether the SCO was type 1 or type 2 and whether the SCO was relevant to the operational product (out of the 1500 SCOs, 910 were relevant, the remainder were SCOs for initial turnovers, support tools, test software or commercial software). Furthermore, each SCO opened contained an estimate of the effort to fix and each closed SCO provided the actual (technical) effort required for the fix. The statistic which was not present, unfortunately, was the actual breakage assessment in SLOC. For each relevant SCO, the SLOC breakage estimate was based on experience with the fix, the detailed description of the resolution, the hours of analysis and the hours required for implementing the fix. While not perfectly accurate in all cases, these estimates are at least consistent relative to each other and given the large sample space, relatively accurate for the intended use. Again, it is not that important to be absolutely exact when the metrics and trends are derived from a large sample and only useful to at most 1 or 2 digits of accuracy.

### CCPDS-R Common Subsystem Analysis

The following paragraphs discuss the quality metrics results for the CCPDS-R common subsystem as a whole with conclusions drawn where applicable. Figure 3 provides CCPDS-R actuals with the incremental build sequence ($SLOC_C$) overlayed for comparison.

Configured SLOC. The CCPDS-R installments of $SLOC_C$ delivered small initial builds (A0/A1 and A2) with the highest risk components. The middle build (A3), while less risky, was bulky and a substantial portion of the build was produced by (somewhat immature) automated tools. Nevertheless, it was installed in two increments (A31 and A32).

SCOs. As expected, the SCO rate is proportional to the $SLOC_C$ rate. The actuals also suggest that the state of the first two builds was higher quality at delivery than the third build. The feeling of the development managers on the project concurs with this assessment but also added that it was during the A3-A4 timeframe when substantial requirements volatility occurred in the user interface and external interface definitions. The number of open SCOs has remained fairly constant with respect to the number generated and hence indicative that the rework is being resolved in a timely fashion.

Rework Resolution. The total rework ($R_1 + R_2$) has also grown at a rate proportional to $SLOC_C$ growth but its rate of growth is decreasing. Now that the software is all configured and turnovers are complete, breakage should start damping out rapidly. The resolved rework ($F_1 + F_2$) tracked the total rework closely with little, if any divergence. The last three months indicate that the rate of resolution is exceeding the rate of breakage. This should indicate to the management team that no serious problems are lurking in the future.

Rework Ratio. The rework rate has grown from the initial builds to an apparently stable value of .15. This would imply that the initial build was more mature at delivery than the second and third builds. With over 98% of the software in $SLOC_C$, this value should be expected to be fairly stable and a

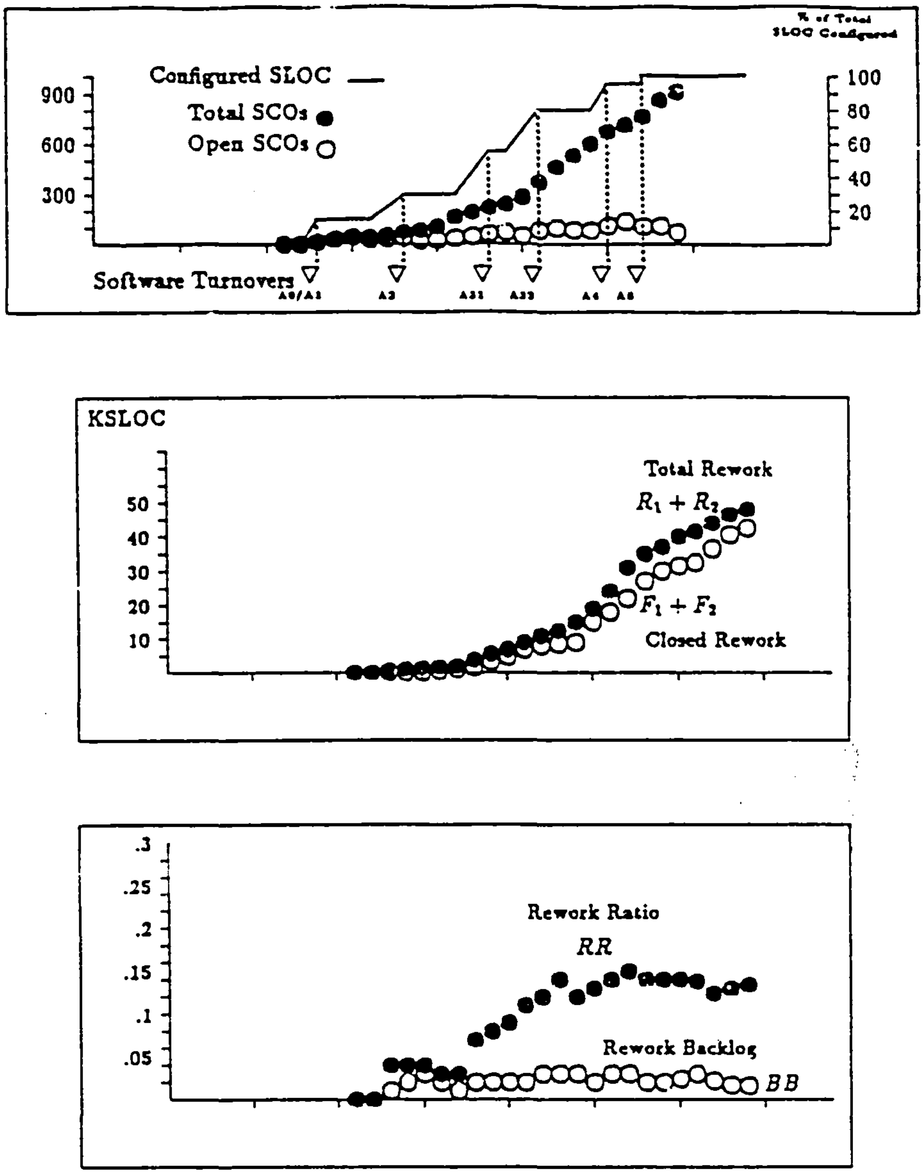


Figure 3: CCPDS-R Collected Statistics

good predictor of future rework. The amount of rework backlog in proportion to $SLOC_C$ has remained fairly constant and implies that the divergence of breakage rate and resolution rate should correct itself shortly. The situation here is that substantial increments are being added to $SLOC_C$ and an increase in breakage vs resolution is expected since the development team is likely focusing on installing baseline components rather than fixing components.

SCO Effort Distributions. Figure 4 identifies the distribution of SCOs by the effort required for resolution. This graphic also suggests that the software is generally easy to modify. A deeper analysis of the data shows that the majority of complex SCOs occurred in the more complex early builds.

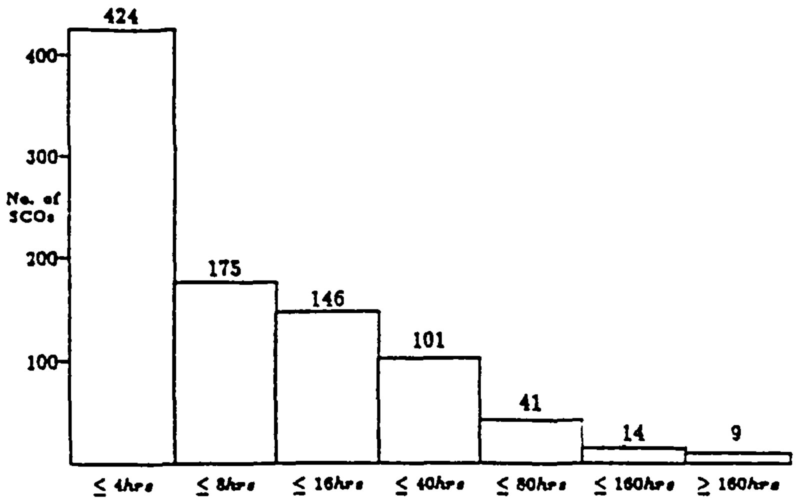


Figure 4: SCO Effort Distribution

Rework Proportions. $R_E$ (Table 4) defines the percent of the development efforts devoted to rework. Since we only tracked the technical effort in analyzing and implementing resolutions, we have compared it to the software development effort devoted to the same, namely, the requirements, design, development and test effort. In both cases we eliminate the cost of management, facility, secretarial, configuration management, quality assurance, and other level of effort administrative activities. Note that we have included the software requirements analysis effort since, in our evolutionary approach, there is only a subtle difference between requirements and design. $R_S$ defines the percentage of source code which has undergone rework. CCPDS-R is currently projecting a rework ratio of 14% .

| Metric | Definition | CCPDS-R Value |
|---|---|---|
| Rework Proportions | $R_E = \frac{Effort_{SCO_1}+Effort_{SCO_2}}{Effort_{Total}}$<br>$R_S = \frac{(R1+R2)}{SLOC_C}$ | 6.7%<br>13.5% |
| Modularity | $Q_{mod} = \frac{R_1+R_2}{SCO_1+SCO_2}$ | 53 $\frac{SLOC}{SCO}$ |
| Changeability | $Q_C = \frac{Effort_{SCO_1}+Effort_{SCO_2}}{SCO_1+SCO_2}$ | 15.7 $\frac{Hrs}{SCO}$ |
| Maintainability | $Q_M = \frac{R_E}{R_S}$ | .49 |

Table 4: End-Product Quality Metrics Definitions

Modularity. This value characterises the extent of damage expected for the average SCO. A value of 53 SLOC implies that the average SCO only affected the equivalent of one program unit. Since most of the trivial errors get caught in standalone test and demonstration activities, this value indicates the average impact for the non-trivial errors which creep into a configuration baseline. This value suggests that the software design is flexible but with no basis for comparison, this is purely conjecture. An additional metric which would be useful in assessing modularity would be the number of files affected per change, This would provide insight into the locality of change as well as the extent. This information was not available in the CCPDS-R historical data, but it is being collected in future data.

Changeability. The average effort per SCO provides a mechanism for comparing the complexities of change. As a project average, 16 hours suggests that change is fairly simple. When change is simple, a project is likely to increase the amount of change thereby increasing the inherent quality.

Rework Improvement. Figure 5 identifies how the changeability ($Q_C$) evolved over the project schedule to date. While conventional experience is that changes get more expensive with time, CCPDS-R demonstrates that the cost per change improves with time. This is consistent with the goals of an evolutionary development approach [12] and the promises of a good layered architecture [13] where the early investment in the foundation components and high risk components pays off in the remainder of the life cycle with increased ease of change. The trend of this metric would indicate that the CCPDS-R software design has succeeded in providing an integrable component set with effective control of breakage. Had the trend of this metric showed growth in effort per SCO without stabilization, management may be concerned about the design quality and downstream risks in reworking an increasingly hard to change product. Note that $Q_C$ metrics do not include the cost of downstream re-verification of higher level requirements since the broad range of these activities would corrupt the intent of the metric. $Q_C$ has been purposely defined to reflect the technical risk of change, not the cost of reverification in a larger context or the management risk. For example, a late change of minor complexity could result in regression test by inspection or a complete reverification of numerous performance threads. This range of effort varies with the context of the change, the customer/contractor paranoia and a variety of other issues which are not reflective of the ease of change. The technical cost of change is not closed out however, until this reverification is complete since it may result in reconsideration.

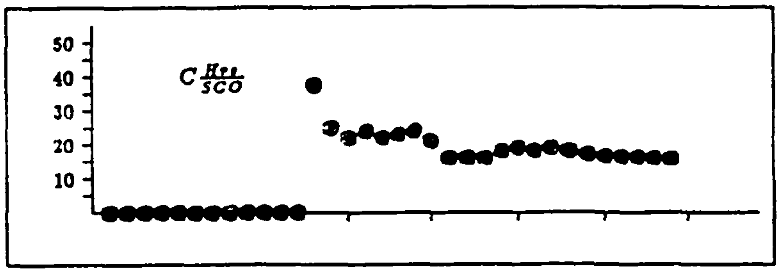


Figure 5: Rework Improvement: Changeability Evolution

Maintainability. The ratio of $R_E$ to $R_S$ characterizes the cost of reworking CCPDS-R components compared to developing them from scratch. This value along with the change traffic experienced during the last phase of the life cycle could be used to predict the maintenance productivity expected from the current development productivity being experienced. The overall change traffic during development should not be used to predict operational maintenance since it is overly biased by immature product changes. The FQT phase change traffic (likely a lower value than the complete development lifecycle traffic), is a more accurate measure. A value of .49 seems like a good maintainability rating, but further project data would permit a better basis for assessment.

This value requires some caveats in its usage. First, this maintenance productivity was derived from small scale maintenance actions (fixes and enhancements) as opposed to large scale upgrades where system engineering and broad redesign may be necessitated. Secondly, the data is derived from the development lifecycle, therefore, it should be treated as more of an upper bound in planning the expectations during the maintenance phase of a product where the existence of defects should be less than that experienced during development. The personnel performing the maintenance actions however, were *knowledgeable developers* which may bias the maintainability compared to the expertise of the maintenance team. The message here, is that this data, like any productivity data, must be used carefully by people cognizant with its derivation to ensure proper usage.

Functional CSCI Analysis. A complete lower level analysis was performed to analyze the various contributions to the values in Table 4 by the individual CSCIs. While the evaluation of this lower level data will not be discussed here in detail, they did uncover some interesting phenomena which have since been incorporated into the plans of future subsystems. There were significant differences in the various CSCI level values which provided insight into various levels of quality and the need for perturbations to future plans. The $Q_M$ varied from .12 to .85 across 6 CSCIs. For example, relatively low values were observed for algorithm (.12) and display (.27) software where ease of change was a clear design goal. Higher values were observed for the external communications software (.51) and system services software (.85) where changes in an external message set for example, could result in broader system impacts. The range of values clearly identifies the *relative* difference in risk associated with changing various aspects of the design. The absolute risk associated with these changes is difficult to assess without further data from other similar projects..

Global Summary. In general, the CCPDS-R program appears to be converging towards a very high quality product with high probability. This assessment is implied from the visible stability in the quality metrics. The fact that these metrics are stable generally implies that the remaining efforts are predictable. If the predictions do not extrapolate to better than required performance, action can be taken. The key to optimising the value of these metrics is to achieve stabilization as early as possible so that if predicted performance does not match expectations, management can instigate improvement actions as early in the life cycle as possible. Some characteristics of CCPDS-R which are important to keep in mind when interpreting the above metrics include:

1. Many changes incurred by the project were really type 3 (true requirements change). However, since most of these were small it was easier to incorporate them rather than go through the formal ECP process. In retrospect, the sum of all these little changes was quite substantial.

2. These metrics are derived from the development phase, comparison with other project's maintenance phase metrics is misleading. The metrics available in the final 3 months prior to delivery (as opposed to the lifecycle averages presented here) however, should be fairly comparable.

Operational Concept. The concept of operations for the software quality metrics program is to provide insight for the purposes of managing product development with minimum interference to the development team. This will be accomplished by integrating the standards for metrics collection into the tools and QCB procedures. The responsibilities of this initiative are allocated as follows:

**Software Developers:** Follow the core Ada Design/Development Standards

**Software Development Managers:** Follow the evolutionary process model, adhere to core software quality metrics policy, coordinate with project systems effectiveness any project unique policies, interpret systems effectiveness SQM analysis and be accountable for issues and resolutions.

**Corporate Systems Effectiveness:** Define the SQM policy/tools/procedures, evaluate project implementations, improve the policies/tools/procedures and ensure consistent usage across different projects. This is the same function proposed by [8] as the standards group.

**Project Software Engineering:** Flowdown the SQM policy/tools/procedures into a project implementation, implement project QCB, SQM collection, SQM analysis, SQM reporting, evaluate project implementations, and propose candidate improvements to the policies/tools/procedures. Note that we are putting this function in the hands of knowledgeable project personnel (as opposed to conventional independent QA personnel) since the administrators of these metrics should be motivated for effective use through ownership in both the process and the products.

We would foresee SQM metrics reporting on a monthly or quarterly basis depending on project phase, size, risks, etc. Furthermore, the entire SQM initiative should be relatively dynamic during its infancy as real project applications determine what is most useful and feedback is incorporated.

## SUMMARY

By itself, CCPDS-R is perhaps a bad example for testing these metrics. In general, the project has performed as planned and has a high probability of delivering a quality product. It would be useful to examine a less successful project to illustrate the tendencies which every project manager should be looking for as indicators of trouble ahead.

Furthermore, none of these metrics by themselves, provides enough data to make an assessment of a project's quality. They must be examined as a group in conjunction with other conventional measures to arrive at an accurate assessment. They also do not represent the only set of useful metrics possible from the collected statistic on SCOs and rework. There are many other ways to examine this data and present it for trend analysis. With further automation, these other views would be simple to produce.

Although not fully implemented on a large project to date, subsets of the metrics presented herein have proven useful in the long term planning and development process improvement on CCPDS-R. TRW is currently in the process of expanding these concepts into a uniform practice across its Ada software development projects supported by automated tools. With the broad acceptance of Ada and evolutionary development techniques, this approach has the potential of providing a uniform technique for quality metrics collection, reporting and history. This data is paramount to the implementation of a consistent TQM approach to software development for enhanced software product quality and more efficient software production. The following activities still need to be performed to provide a complete initiative:

1. Enhance the standard SCO form with definitions, standards and procedures for usage.

2. Develop a portable SLOC Counting Tool (the current CCPDS-R Metrics Tool would satisfy this with minor modifications).

3. Identify Ada standards (which would be mandatory across all Ada projects) necessary to guarantee consistent metrics collection across projects and within projects. This primarily involves standards for program unit headers and program layout which are not controversial.

4. Develop an SCO data base management system with supporting tools for automated collection, analysis and reporting in the formats defined above and other, as yet undiscovered, useful formats.

5. Define QCB procedures, guidelines for metrics analysis and candidate reporting formats.

6. Incorporate this initiative into corporate policy.

As a conclusion, we should evaluate the approach presented herein with our original goals:

1. Simplicity. The number of statistics to be maintained in an SCO database to implement this approach is 5 (type, estimate of damage in hours and SLOC, actual hours and actual SLOC to resolve) along with the other required parameters of an SCO. Furthermore, metrics for $SLOC_C$ and $SLOC_T$ need to be accurately maintained. If automated in an online DBMS, the remaining metrics could be computed and plotted from various perspectives (e.g., by build, by CSCI) in a straightforward manner. Depending on the extent of discipline already inherent in a project's CCB and development metrics, the above effort could be viewed as very simple (as in the case of CCPDS-R) to complex (undisciplined, management by conjecture projects).

2. Ease of Use. The metrics described herein were easy to use by CCPDS-R project personnel and managers familiar with the project context. Furthermore, they provide an objective basis for discussing current trends and future plans with outside authorities and customers. Most trends are obvious and easily explained. Some trends require further analysis to understand the underlying subtleties. End-product metrics provide simple to understand indicators of different software quality aspects for the purposes of comparison and future planning as well as assessment of process improvement.

3. **Probability of Misuse** There are enough perspectives that provide somewhat redundant views so that misuse should be minimized. Without further experience, however, it is not clear that contractor and customer will always interpret them correctly. Although correct interpretation could never be guaranteed, it would be beneficial to obtain more experience to evaluate where misinterpretation is most likely.

## BIOGRAPHY

**Walker Royce** is currently the Principal Investigator for an Independent Research and Development Project directed at expanding distributed Ada architecture technologies proven on CCPDS-R. He received his BA in Physics at the University of California, Berkeley in 1977, MS in Computer Information and Control Engineering at the University of Michigan in 1978, and has 3 further years of post-graduate study in Computer Science at UCLA. Mr. Royce has been at TRW for 12 years, dedicating the last six years to advancing Ada technologies in both research and practice. He served as the Software Chief Engineer responsible for the software process, the foundation Ada components and the software architecture on the CCPDS-R Project from 1987-1990. From 1984-1987, he was the Principal Investigator of SEDD's Ada Applicability for $C^3$ Systems Independent Research and Development Project. This IR& D project resulted in the foundations for Ada COCOMO, the Ada Process Model and the Network Architecture Services Software, technologies which earned TRW's Chairman's Award for Innovation and have since been transitioned from research into practice on real projects.

## REFERENCES

[1] Andres, D. H., "Software Project Management Using Effective Process Metrics: The CCPDS-R Experience" *APCEA Military/Government Computing Conference Proceedings*, Washington D.C., January 1990.

[2] Boehm, B. W., J. R. Brown, H. Kaspar, M. Lipow, G. MacLeod, and M. J. Merrit, "Characteristics of Software Quality," *TRW Series of Software Technology*, Volume 1, TRW and North Holland Publishing, 1978.

[3] Boehm, B. W., *Software Engineering Economics*, Prentice Hall, 1983.

[4] Boehm, B. W., "Improving Software Productivity", *Computer*, September 1987.

[5] Boehm, B. W., Royce, W. E., "TRW IOC Ada COCOMO: Definition and Refinements", *Proceedings of the 4th COCOMO Users Group*, Pittsburgh, November 1988.

[6] Boehm, B. W., Royce, W. E., "COCOMO Ada et le Modele de Developpement Ada", *Genei Logiciel*, December 1989, pp. 36-53.

[7] Boehm, B. W., "The Spiral Model of Software Development and Enhancement", *Proceedings Of The International Workshop On The Software Process And Software Environments*, Coto de Caza, CA, March 1985.

[8] DeMarco, T., *Controlling Software Projects*, Yourden Press, 1982.

[9] Grady, R. B., and Caswell, D. L., *Software Metrics: Establishing a Company Wide Program*, Prentice Hall, 1986.

[10] Humphrey, W., *Managing the Software Process*, SEI Series in Software Engineering, Addison Wesley, 1989.

[11] Jones, C., *Programming Productivity: Issues for the Eighties*, IEEE Computer Society Press, 1981.

[12] Royce, W. E., "TRW's Ada Process Model For Incremental Development of Large Software Systems", *Proceedings of the 12th International Conference on Software Engineering*, Nice, France, March 26-30 1990.

[13] Royce, W. E., "Reliable, Reusable Ada Components For Constructing Large, Distributed Multi-Task Networks: Network Architecture Services (NAS)", *TRI-Ada Proceedings*, Pittsburgh, October 1989.

[14] Shooman, M. L., *Software Engineering*, McGraw Hill, 1983.

[15] Springman, M. C., "Incremental Software Test Methodology For A Major Government Ada Project ", *TRI-Ada Proceedings*, Pittsburgh, October 1989.

[16] Springman, M. C., "Developing Maintainable & Reliable Ada Software, A Large Military Application's Experience", *Ada Europe Proceedings*, Dublin, Ireland, June 1990.

VIEWGRAPH MATERIALS

FOR THE

W. ROYCE PRESENTATION

# Pragmatic Quality Metrics
# For Evolutionary Software Development Models

W. E. Royce

November 1990

TRW

# Large Software Project Issues

Primary Contributors To Software Diseconomy of Scale (Boehm):

- Rework
- Interpersonal Communications
- Requirements Volatility

Ada COCOMO (Boehm/Royce) Speculates That Economy of Scale Is Possible

- Use an Evolutionary Development Approach
- Use Ada as a Lifecycle Language

TRW

# Evolutionary Development Objectives

Optimize Rework

- Minimize Ineffective Rework
  - Do Hard Parts First
  - Compartmentalized Breakage
- Maximize Rework Efficiency
  - Fix it Early
  - *Design for Change*

Minimize Interpersonal Communications

- Small Expert Design Team
- Layered Architecture
- Self Documenting Lifecycle Language (Ada)

Optimize Requirements Volatility

- Stablize Necessary Primitives Early
- Change Requirements As Product Matures
- *Design for Change*

TRW

# SQM Focus

Most Important Software Quality is that it is SOFT

- **Modularity:**
  - Breakage Extent When Changed
- **Changeability:**
  - Complexity of Effort to Analyze/Implement Change
- **Maintainability:**
  - Productivity of Change

Assumptions:

- Evolutionary Process Model
- Consistent SLOC Counting
- Configuration Control Board For Change Assessment

TRW

# SQM Definition

Quality: *Degree of compliance with customer expectations of function, performance, cost and schedule*

Quality Metrics Derived from Measurement of Rework

- Type 1 Rework: Fix Bad Quality Instance
- Type 2 Rework: Improve Quality
- Type 3 Rework: Requirements Change

All Rework Corresponds to Quality Increase

Quality Metrics Approach

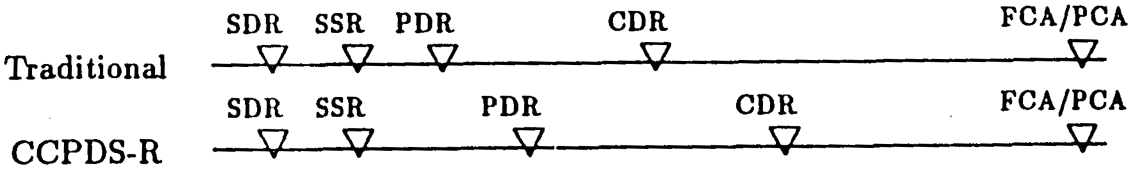
- Collect Statistics on Rework over Project Lifecycle
- Quantify Meaningful Metrics
- Plot Processed Statistics Over Time
  - Quality Progress Trends

TRW

# CCPDS-R CDR STATUS

TRW

- Evolutionary Approach Permits Tangible Insight into End-product



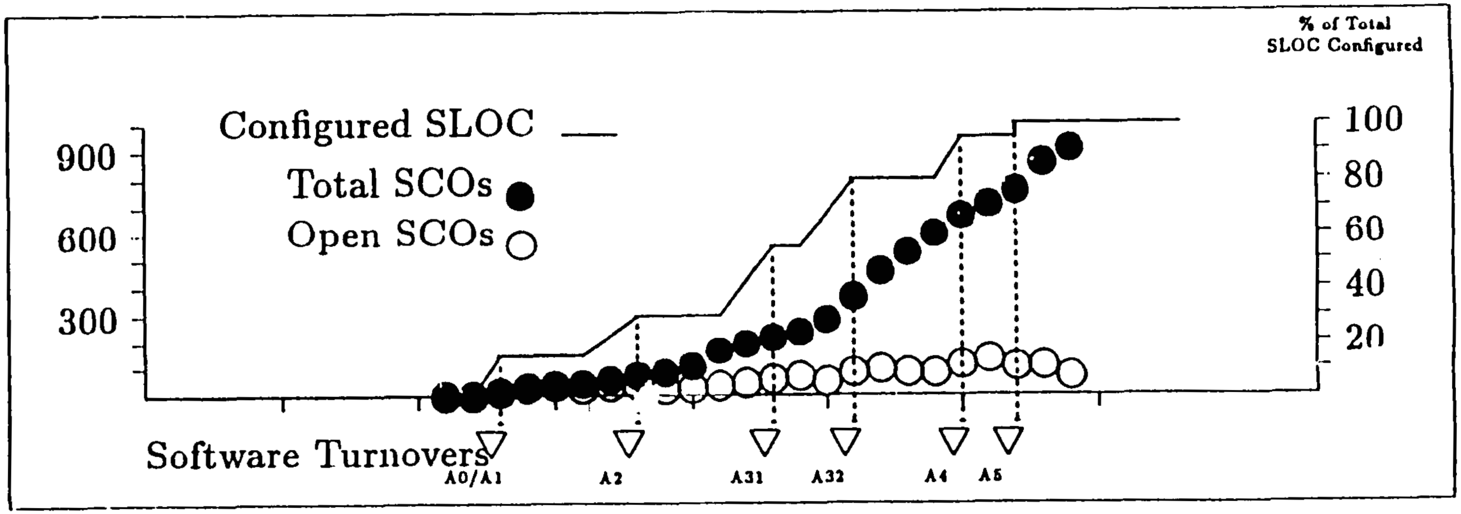
| | Traditional Approach | CCPDS-R Approach |
|---|---|---|
| Software Design | Complete | Complete |
| Code development | ≈10% | 94% |
| Software Integration | Negligible | 74% |
| Formal Test | 0% | 12% |
| Performance Assessment | Modeling | 80% of Operational S/W Demonstrated |

W. Royce
TRW
Page 24 of 30

This New Approach Enables Early Software Quality Assessment

TRW

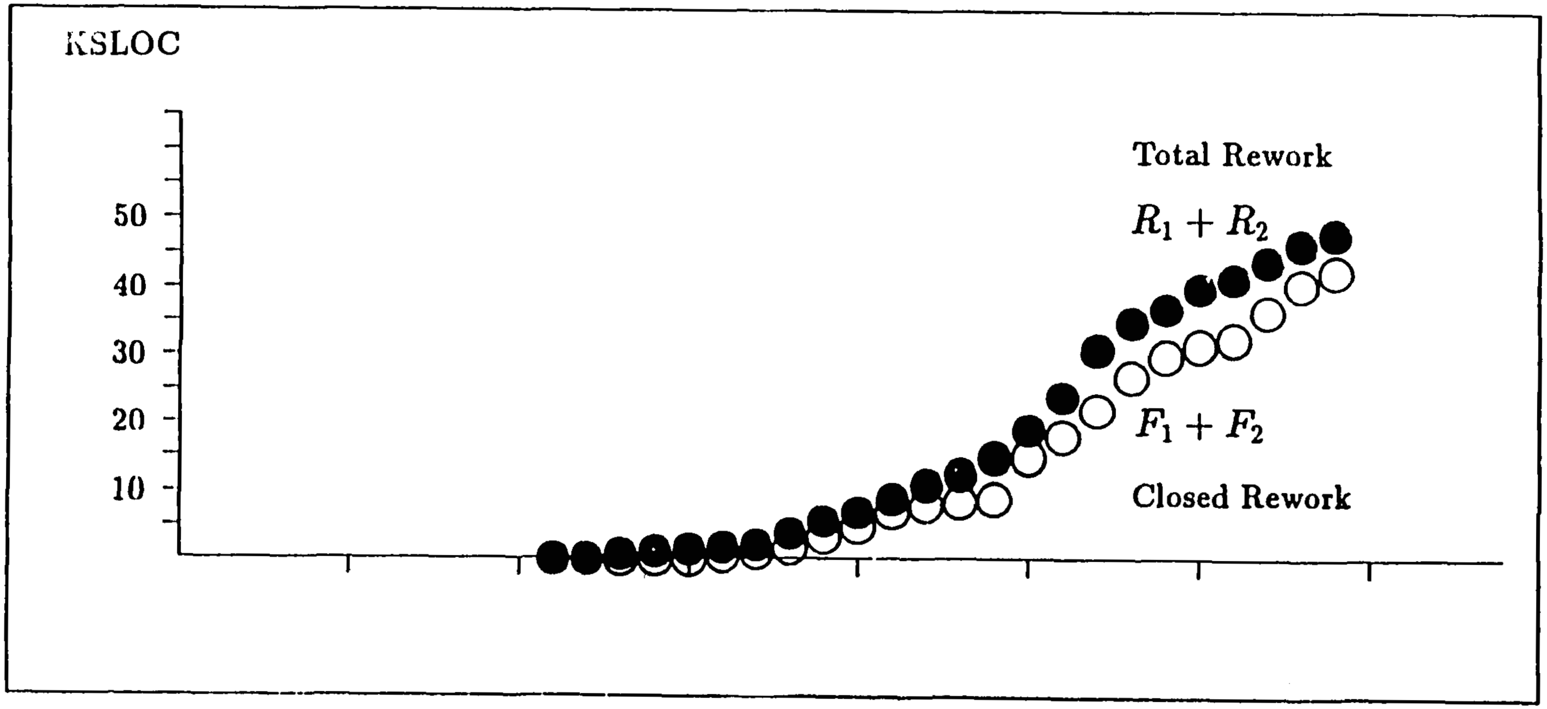
# CCPDS-R Experience



TRW

# CCPDS-R Experience

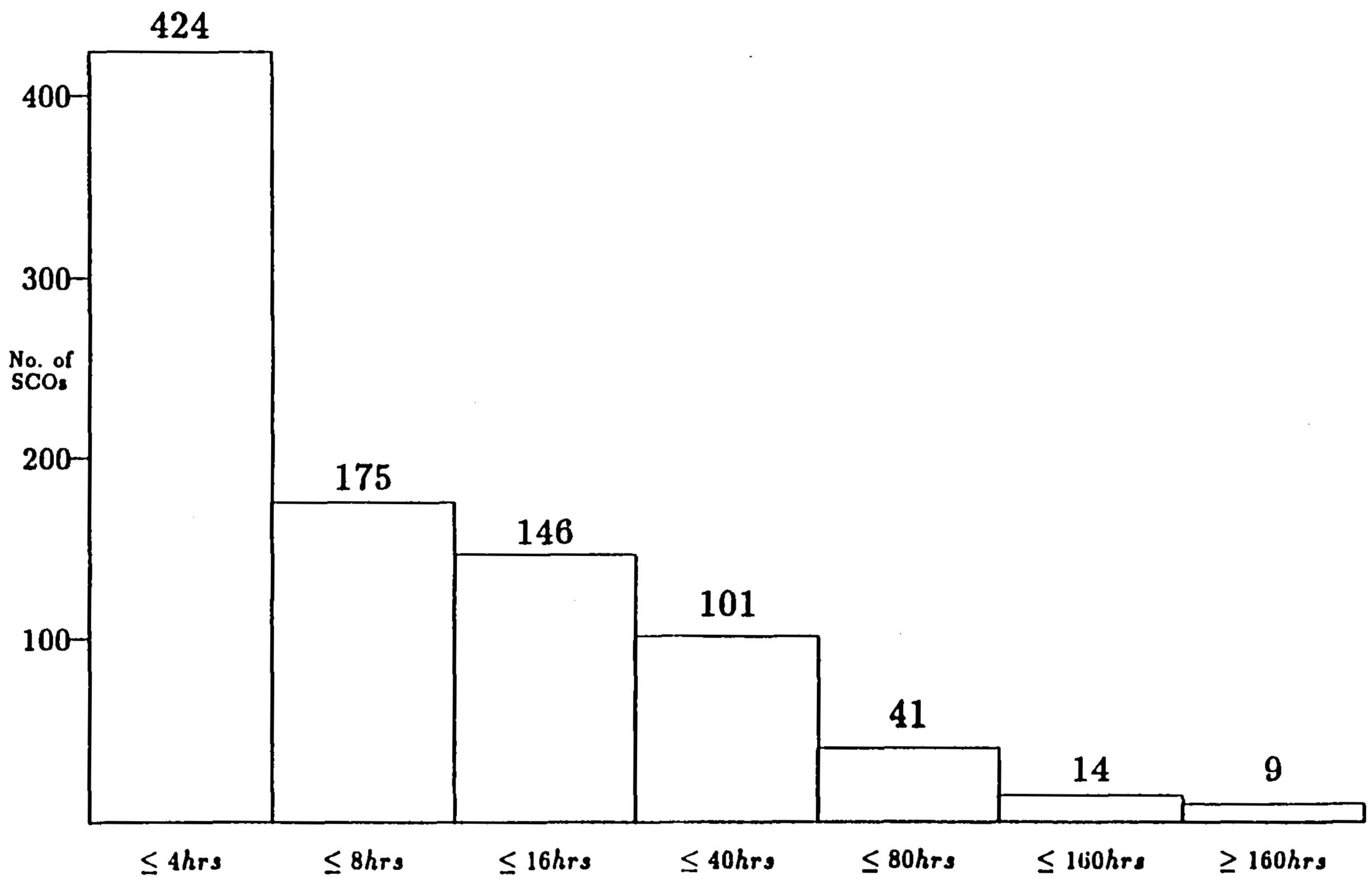
TRW



W. Royce
TRW
Page 26 of 30

TRW

# CCPDS-R SCO Experience

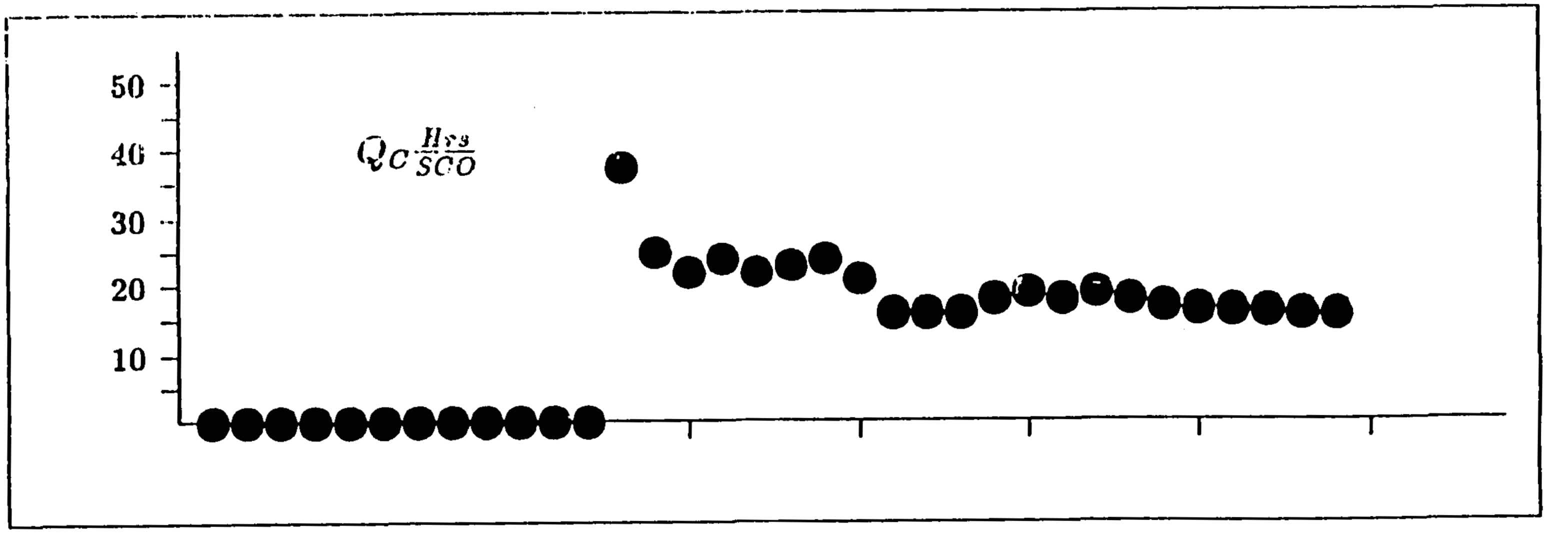


| TRW | CCPDS-R Quality Metrics Actuals | TRW |
|---|---|---|

| Metric | Definition | CCPDS-R Value |
|---|---|---|
| Rework Proportions | Rework: $R_E$ =% of Effort<br>Rework: $R_S$ =% of Product | 6.7%<br>13.5% |
| Modularity | $Q_{mod}$ =Average Breakage per Change | 53 $\frac{SLOC}{SCO}$ |
| Changeability | $Q_C$ =Average Effort per Change | 15.7 $\frac{Hrs}{SCO}$ |
| Maintainability | $Q_M$ =Normalized Rework Productivity | .49 |

TRW

# CCPDS-R Changeability Evolution



# Conclusions

Important Needs For Successful Use:

- Consistency of Application
- Automated Tools
- Management And Practitioner Acceptance

Advantages:

- Quantitative Data For Decision Making
- Quantitative Data For Subjective Requirements Compliance
  - Maintainability, Modularity, Adaptability, etc.
- Historical Data for Better Future Planning

Disadvantages:

- No Existing Multi-project Historical Database
  - Only CCPDS-R Data Exists Now
- No Existing Project Independent Toolsuite

Quality Metrics *Can* Be Used Effectively

N92

19428

UNCLAS

# JPL's Real-Time Weather Processor Project (RWP)

## Metrics and Observations at System Completion

## Build 3

by Robert E. Loesh (RWP Project Office)
Robert A. Conover (RWP Project Manager)
Shan Malhotra (System Analysis Section)

Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, California 91109

This work was performed by the Jet Propulsion Laboratory, California Institute of Technology, for the FAA through an agreement with NASA (NAS 7-918).

This presentation is an update to the November 1988 GSFC First Ada Symposium presentation which provided preliminary data reflecting the RWP Project at the Build-3 Preliminary Design Review. This presentation is based upon the completion of the RWP Build-3 development and the associated Metrics Report draft. The RWP Build-3 Metrics Report will be completed in March 1991 and will be submitted for public release which may take 3-5 months. Because this presentation is based on the draft Metrics Report, prior to complete validation of all the data, minor corrections may result after the Final Metrics Report is completed.

The development of the RWP System is sponsored by the Federal Aviation Administration (FAA). The RWP is one of several weather information programs the FAA has identified in the FAA's National Airspace System (NAS) Plan, which describes all programs planned for modernizing and improving air traffic control and airway facilities services by the year 2000.

An integral part of the overall upgraded NAS, the objective of the RWP is to improve the quality of weather information and the timeliness of its dissemination to system users. To accomplish this, an RWP will be installed in each of the Center Weather Service Units (CWSUs), located in 21 of the 23 Air Route Traffic Control Centers (ARTCCs). The RWP System is a Prototype System. It is planned that the software will be GFE and that production hardware will be acquired via industry competitive procurement.

The ARTCC is a facility established to provide air traffic control service to aircraft operating on Instrument Flight Rules (IFR) flight plans within controlled airspace, principally during the en route phase of flight. Beginning in 1993, and continuing to 1998, the ARTCCs will be reconfigured to include both en route and approach control functions. The reconfigured facilities will be called Area Control Facilities (ACFs).

RWP will process up to 27 Next Generation Radar (NEXRAD) weather data simultaneously in real-time and create mosaic displays. The processed NEXRAD data is disseminated directly to meteorologists and FAA aircraft controllers. This information is updated every three to five minutes.

The RWP project was started in November of 1987 which resulted from the descoping of the Central Weather Processor Project (CWP). At the time of the descoping the CWP was in detailed design and planned for the "C" programming language development environment. RWP is following DOD-STD-2167A and the software will be coded in the DOD standard ADA programming language. RWP is composed of 3 incremental development builds (Build-1, Build-2 and Build-3). Build-3 contains all of the capabilities specified in the RWP System Specification. There was one Preliminary Design Review (PDR) for the entire system and an individual Critical Design Review (CDR) for each Build. The Coding and Unit Testing (CUT) was completed in February 1990. System Testing was completed in June 1990. FAA Prototype (FAA Users) Test & Evaluation (PT&E) was completed in July 1990. Following PT&E several changes were made to improve the Man-Machine Interface and System Reliability. This was followed by the FAA Formal System Acceptance Test (FSAT) completed in October 1990. Final as-built documentation and the FSAT Test Report are scheduled for mid January 1991.

The system is composed of one CSCI developed by JPL that has 704 Computer Software Units (CSUs) and is composed of 97,687 Ada Statements, number of semicolon ";" delimiters, (or 213,961 Source Lines of Code ((SLOC)), carriage return delimiters less comments and blanks, but including specifications and data, type, declarations). In addition it has 4,330 of "C" SLOC.

In addition to the software developed by JPL there are two areas where Commercial-Off-The-Shelf (COTS) software is used:

- Communications Protocols
- Man-Machine Interface (DECWindows and DECs Forms Management System)

Following are some of the metrics and observations.

Requirements Metrics and Observations

The RWP System Specification contains a total of 223 requirements within 70 pages of the document. On the average there are about 3 requirements per page. This does not include the specification of the external RWP System-to-System interfaces. These are contained in a series of Interface Control Documents (ICDs). The System Specification was approved May 1988. Any System Specification questions, clarifications, or additions were reviewed and negotiated by the RWP System Design Teat (SDT) which was composed of key technical lead staff from each area (Project Office, System Engineering, Software Development, Hardware Development, Test and Operations, Product Assurance and Configuration Management). Results of these meetings were processed using Project Configuration Management procedures and documented in the SDT minutes as "Open Issues." During the development 222 Open Issues were discussed by the SDT and approximately 40% were external system interface issues relating to ICDs. The 222 Open Issues resulted in 52 Engineering Change Requests (ECRs) and 34 Request for Deviation/Waivers, (RDWs) to the System Specification. RDWs were used as the interim method for correcting wording in the RWP System Specification.

Approximately 2/3 of the Open Issues were generated by the Test and Operation Organization (TOO) resulting primarily from the preparation of the System Integration and Test Descriptions and Procedures. The other 1/3 were generated by the Software Development Organization (SDO).

The 52 ECRs and 34 RDWs caused a significant rework impact late in the development life cycle.

The conclusion we have drawn is that if the System Integration Test Descriptions and Procedures had been prepared earlier in the life cycle, most of the Open Issues would have been initiated and resolved before much of the development was completed or even started and the amount of rework would have been minimized (significantly less).

SPFR Metrics and Observations

1,266 Software Problem Failure Reports (SPFRs) were generated which were based upon requirements (Priority 1,2,3); see DOD-STD-2167A error classification.

SPFRs reflect all errors reported during software (CSCI) or system related requirements testing. The only exception is that any errors found during Coding and Unit Testing and CSC Integration Testing still outstanding at the start of CSCI Requirements Testing were turned into SPFRs at that time.

Most notable is the small number of SPFRs (18%) that existed at the start of CSCI Requirements Testing and the large % (40%) of SPFRs found during System Integration and Testing (SIT). Because of schedule pressures the CSCI Requirements Testing (9% of errors) was deleted for the third incremental Build. This explains the small number of errors found (9%) during CSCI Requirements Testing and likely contributed to the large number found during system level testing (SIT, FSAT-1, FSAT-3 = 51%).

While there are no specific comparisons or conclusions we are prepared to make on the SPFR code growth. It may serve as an important point of reference to note the code growth per SPFR for embedded systems where the memory utilization and margin is critical. Our experience over six interim error correction Builds is that we had approximately 8.4 Ada statements of increase for each SPFR corrected. This does not provide any detail of number of specific amounts of code deleted, changed and added; only the net result.

During SIT there were 5 errors reported per 1000 Ada statements. A more useful number is the error density per SLOC which allows for comparison to numerous density reports on previous other developments. It is typical in this phase to see error density rates in the 3 to 10 errors per thousand range with the median falling around 5-6 errors per thousand. Comparing the RWP Project error densities with other Fortran, "C" type developments it is our observation that there were fewer (approximately 50%) errors during the RWP, SIT that some previous projects. Some of this probably is due to the use of Ada. However, other factors also contributed such as quality of staff, low attrition of staff, etc.

Based upon the number of work years of effort for CSCI Requirements Testing versus the number of work years for SIT, SIT was 51% more productive in error generation. This is probably exaggerated somewhat due to the deletion of the Build-3 CSCI Requirements Testing.

The metrics of the number of work days to fix an SPFR is between 1.9 and 2.3 work days. The average is 2.1 work days per SPFR correction. This includes any design, coding, unit testing, CSU and CSC integration and delivery of the code to the Project Software Library.

The % of SPFRs fixed that were incorrect or created other problems attributable to the fixed code was 3% or less. This allowed us to use the 4-6 week period prior to release of Builds for various system level tests (SIT, FSAT, PT&E) to continue to be used SPFR correction rather verification of the SPFRs fixed. With 2-3 months centers for Build deliveries and version updates, it provided us with 1/3 more time to fix SPFRs and a higher overall SPFR correction productivity rate given a fixed period.

Ada Portability Metrics and Observations

Ada portability was established as a Project high priority design goal. The object was to minimize the various code constructs that may need to changed using the same programming language and software design but different hardware. The following specific design decisions were made to meet the portability goal:

- Ada Programming Language & Standard
- Ada Tasking to minimize Operating System function uniqueness
- DEC Windows (X-windows subset) to minimize the Man-Machine Interface rework
- Object Oriented Design Methodology to localize external interface dependencies and rework
- Other Engineering Principles and Standards to minimize rework

A tool was developed and used by the Product Assurance staff to analyze the code to identify each non-portable construct and provide summary statistics. Because of the still less than stable industry standards on X-windows the tool produced the portability results both with DECWindows portable and non-portable.

Portability can be measured any number of ways. One of the most useful is to measure the number of specific code constructs that run a risk of needing to be modified for execution on a different computer than that used for the RWP system (i.e., DEC). This measure could then be compared to the number of code constructs existing in the developed RWP software. The tool does provide the number of non-portable constructs (i.e. 12,267). However there is no measurement of the number of total code constructs in the RWP developed code. There is a count of the number of Ada statements (i.e. 97,687). There may be 1 or more constructs per Ada statement but it is still a useful number to quantify the ratio or metric of % portable. If we divide the total Ada statements into the non-portable constructs we get the answer approx. 12.5%. Therefore, on a construct basis, the RWP system is at least 87.5% portable. This does not include any changes needed to accommodate word size or reformatting to accommodate storage devices that are unique. It should be cautioned that using the difficulty classification to compute work hours to port the system should not be done. Since many of the porting changes for one type of construct is mechanically repeatable and represents a single instance, worst case, the estimation of porting effort needs to consider repeatability. In addition, not all of the constructs identified as a porting risk may need to be ported.

However, the metrics and analysis should set an industry reference point for specifying design requirements for portability.

# VIEWGRAPH MATERIALS

# FOR THE

# R. LOESH PRESENTATION

# RWP METRICS AND OBSERVATIONS AT SYSTEM COMPLETION (BUILD-3)

ROBERT E. LOESH (RWP)
ROBERT A. CONOVER (RWP)
SHAN MALHOTRA (SYSTEMS ANALYSIS SECTION)

NOVEMBER 28, 1990

JPL

# RWP METRICS AND OBSERVATIONS

RWP

- DATA IS PRELIMINARY
- MINOR CORRECTIONS MAY RESULT AFTER VALIDATION PROCESS
- METRICS REPORT TO BE COMPLETE IN MARCH 1991
  - WILL START PROCEDURE FOR PUBLIC RELEASE OF METRICS REPORT
  - RELEASE MAY TAKE 3 - 5 MONTHS

RCL-2

# RWP METRICS AND OBSERVATIONS

JPL

## AGENDA

- WHAT IS THE RWP SYSTEM?
- REQUIREMENTS ISSUES
- TESTING EFFECTIVENESS
- ERROR DENSITY AND DISCOVERY RATE
- ADA ERROR CORRECTION RATES
- PORTABILITY ISSUES

# RWP METRICS AND OBSERVATIONS

JPL

## WHAT IS THE RWP SYSTEM?

- SPONSOR: FEDERAL AVIATION ADMINISTRATION (FAA)

- PROTOTYPE DEVELOPMENT; EVENTUALLY PART OF NATIONAL AIRSPACE SYSTEM UPGRADE

- RWP WILL PROCESS WEATHER DATA IN REAL-TIME BY CREATING A MOSAIC DISPLAY OF UP TO 27 RADARS SIMULTANEOUSLY. THE DATA WHICH IS DISSEMINATED DIRECTLY TO THE FAA AIRCRAFT CONTROLLERS AND METEOROLOGISTS IS UPDATED EVERY THREE TO FIVE MINUTES

- PROJECT MILESTONES:
  - PROJECT START - NOVEMBER 1987
  - CODING COMPLETE - FEBRUARY 1990
  - SYSTEM TESTING COMPLETE - JUNE 1990
  - FAA OPERATIONAL TEST AND EVALUATION - JULY 1990
  - FAA FORMAL SYSTEM, ACCEPTANCE TEST - OCTOBER 1990

- 1 RWP SYSTEM AT 21 OF 23 AREA CONTROL FACILITIES; 7 EXTERNAL INTERFACES

JPL

# RWP METRICS AND OBSERVATIONS

RWP

## WHAT IS THE RWP SYSTEM? (CONT'D)

o S/W INTENSIVE; H/W OFF-THE-SHELF

- 1 COMPUTER S/W CONFIGURATION ITEM

  - DEVELOPED BY JPL: 97,687 (ADA STATEMENTS)
    213,961 (CARRIAGE RETURNS (COMMENTS AND BLANKS))
    4,330 (C SLOC)

  - COMMERCIAL OFF-THE-SHELF: 280,238 (C SLOC)

    -- COMMUNICATIONS PROTOCOLS
    -- DEC-WINDOWS
    -- DEC FORMS MANAGEMENT SYSTEM

- ADA, DOD-STD-2167, REVISION A: TAILORED

# RWP METRICS AND OBSERVATIONS

JPL

RWP

## WHAT IS THE RWP SYSTEM? (CONT'D)

- DISTRIBUTED H/W ARCHITECTURE

  - 10 MICRO VAX IIS, 3 MICRO VAX 3600S, 1 MICRO VAX 3200

  - VAXELN AND VAX/VMS OPERATING SYSTEMS, DECNET, ISO PROTOCOLS

- THE RWP SYSTEMS ARE SCHEDULED TO BE INSTALLED IN THE FAA CONTROL CENTERS BY 1994 BY A FAA SYSTEM CONTRACTOR WHO IS SCHEDULED FOR SELECTION IN 1992. JPL IS PLANNING TO PROVIDE TECHNICAL SUPPORT TO THE FAA THROUGH 1994 FOR THE INSTALLATION OF THE FIRST THREE OF 23 SITES.

# RWP METRICS AND OBSERVATIONS

JPL

RWP

## REQUIREMENTS ISSUES

- o SYSTEM SPECIFICATION (WRITTEN BY JPL AND FAA)
  - 205 FUNCTION AND PERFORMANCE REQUIREMENTS
  - + 18 PERFORMANCE (COUNTED AS 1)

    223

- o SYSTEM SPECIFICATION FUNCTIONAL AND PERFORMANCE REQUIREMENTS PAGES = 70 APPROXIMATELY 0.3 PAGES/REQUIREMENT

- o PROJECT SYSTEM DESIGN TEAM (SDT) ADDRESSED REQUIREMENTS ISSUES AT WEEKLY MEETINGS

- o ISSUE RESOLUTIONS WERE DOCUMENTED IN DESIGN TEAM MINUTES AND PROCESSED VIA CONFIGURATION MANAGEMENT:
  - ENGINEERING CHANGE REQUESTS (ECRs) TO SRS, ICDs AND SYSTEM/SEGMENT DESIGN DOCUMENT
  - REQUEST FOR DEVIATION/WAIVER (RDW) TO SYSTEM SPECIFICATION

# RWP METRICS AND OBSERVATIONS

JPL

RWP

## REQUIREMENTS ISSUES (CONT'D)

- 222 OPEN ISSUES DISCUSSED AT DESIGN TEAM
  - APPROXIMATELY 40% WERE INTERFACE (ICD) ISSUES
- RESOLUTION RESULTED IN:
  - 52 ECRs TO ICDs, SRS AND SSDD
  - 34 RDWs TO SYSTEM SPECIFICATION
- APPROXIMATELY TWO-THIRDS OF OPEN ISSUES CAME FROM SIT STAFF DOING STT DESCRIPTIONS AND PROCEDURES (SITD/P)
- MOST ECRs AND RDWs RESULTED IN SOFTWARE, DOCUMENT AND TEST PROCEDURE AND DATA REWORK

WRITE INITIAL VERSION OF SYSTEM TEST PROCEDURES AS EARLY AS POSSIBLE

# RWP METRICS AND OBSERVATIONS

JPL

RWP

## SPFR COUNT
## (BY PHASE)

- o TOTAL APPROXIMATELY 2,100 SPFRs TO DATE
  - 1,452 RWP CSCI RELATED
  - 1,266 PRIORITY 1 - 3

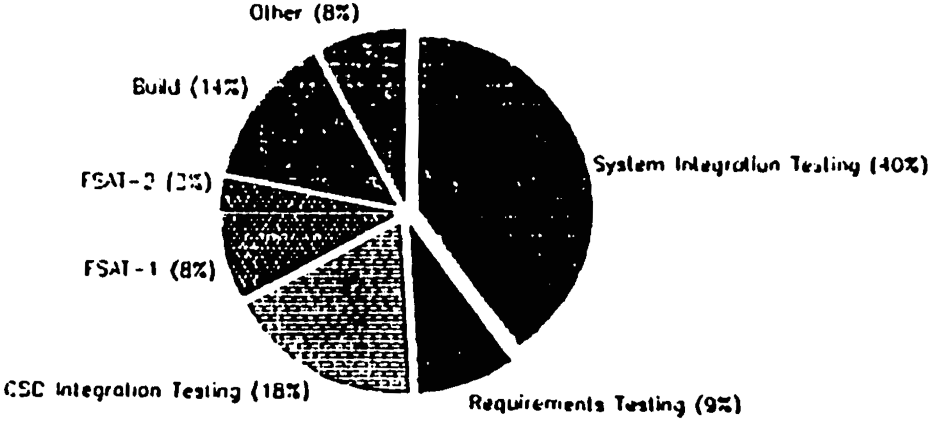
| PHASE | SPFR COUNT | PERCENT |
|---|---|---|
| SYSTEM INTEGRATION TESTING | 508 | 40% |
| CSCI REQUIREMENTS TESTING | 117 | 9% |
| CSC INTEGRATION TESTING | 231 | 18% |
| CODE AND UNIT | 6 | 0% |
| FSAT-1 | 98 | 8% |
| FSAT-2 | 35 | 3% |
| BUILD | 172 | 14% |
| OTHER | 98 | 8% |
|  | 1,266 | 100% |

JPL

# RWP METRICS AND OBSERVATIONS

RWP (FSR)

## RWP SPFR DISTRIBUTION

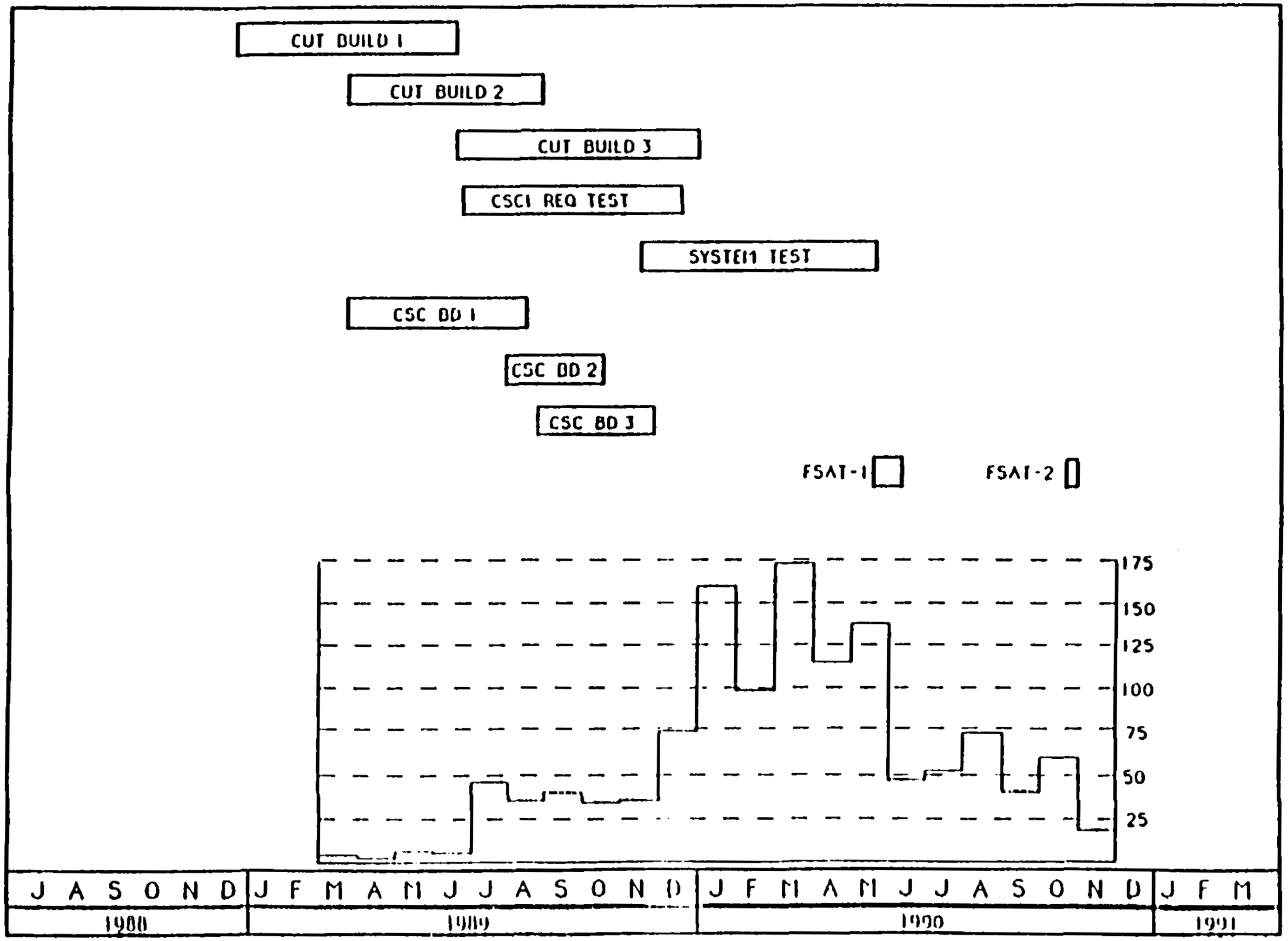


# RWP METRICS AND OBSERVATIONS

JPL

RWP

## RWP SPFR DISTRIBUTION (BY PHASE)



R. Lorch
NASA/JPL
Page 16 of 25

# RWP METRICS AND OBSERVATIONS

## SPFR DENSITY

| BUILD | CARRIAGE RETURNS | SEMI-COLONS | Δ FROM PREVIOUS BUILD (CARRIAGE RETURNS) | Δ FROM PREVIOUS BUILD (SEMI-COLONS) | SPFRS FIXED |
|---|---|---|---|---|---|
| 3.8 | 401,544 | 94,886 | --- | --- | 128 |
| 3.9 | 405,537 | 95,613 | 3,993 | 727 | 55 |
| 3.10 | 408,895 | 96,214 | 3,358 | 601 | 80 |
| 3.11 | 409,362 | 96,308 | 467 | 89 | 18 |
| 3.12 | 414,546 | 97,687 | 5,164 | 1,379 | 78 |
| 3.13 | 420,804 | 99,288 | 6,258 | 1,601 | 135 |
| 3.14 | 418,433 | 98,748 | -2,371* | -504* | 95 |
| TOTAL 6 BUILDS (3.9 - 3.14) | | | 16,889 | 3,857 | 461 |

| | |
|---|---|
| SEMI-COLONS PER SPFR | 8.4 |
| CARRIAGE RETURNS PER SPFR | 36.6 |

# RWP METRICS AND OBSERVATIONS

## ERROR DENSITY AND DISCOVERY RATES

- CSCI REQUIREMENTS TESTING
  - STOPPED FOR BUILD-3: SCHEDULE AND RETURN ON INVESTMENT
  - MOVED TO SDO FOR BUILD-4
    - PRODUCTIVITY: LESS OVERHEAD TO ERROR PROCESSING
    - EMPHASIZE REQUIREMENTS RESPONSIBILITY OF SDO STAFF

- SYSTEM INTEGRATION AND TESTING (SIT)
  - NINE MONTHS TEST EXECUTION PERIOD
  - RESET AFTER THREE MONTHS TO ACCOMMODATE LATE SOFTWARE DELIVERY
  - 40% OF ERRORS FOUND DURING SIT

# RWP METRICS AND OBSERVATIONS

JPL

## ERROR DENSITY AND DISCOVERY RATES (CONT'D)

- FORMAL ACCEPTANCE SYSTEM (FSAT)
  - FAA TEST WITNESS
  - 11% OF SPFRs FOUND DURING FSAT
  - TWO FSATs
    - FSAT-1: APPROXIMATELY THREE WEEKS: 98 SPFRs (8% SPFRs)
    - FSAT-2: APPROXIMATELY ONE WEEK: 35 SPFRs (3% SPFRs)

- 95+% SYSTEM FUNCTION AND PERFORMANCE REQUIREMENTS FULLY VALIDATED

- SIT METRICS AND OBSERVATIONS
  - APPROXIMATELY 5 ERRORS PER 1000 ADA STATEMENTS
  - APPROXIMATELY 2.3 ERRORS PER 1000 CARRIAGE RETURNS

APPROXIMATELY 1/2 LESS THAN TYPICAL FORTRAN

# RWP METRICS AND OBSERVATIONS

RWP

JPL

## ERROR DENSITY AND DISCOVERY RATES (CONT'D)

- SIT VERSUS CSCI REQUIREMENT TESTING (NOTE-1)
  - CSCI REQUIREMENTS TESTING APPROXIMATELY 19.5 ERRORS/TEST WORK YEAR
  - SYSTEM INTEGRATION AND TESTING APPROXIMATELY 29.5 ERRORS/TEST WORK YEAR

SIT APPROXIMATELY 51% MORE PRODUCTIVE THAN CSCI REQUIREMENTS TESTING

<u>NOTE-1</u>: SYSTEM = 1 CSCI

# RWP METRICS AND OBSERVATIONS

## ADA ERROR CORRECTION RATES

| NUMBER OF SPFRs IN BUILD | WORK DAYS PER/SPFR | |
|---|---|---|
| 108 | 1.5 | |
| 128 | 2.2 | - 2 - 3 WEEKS/BUILD |
| 55 | 2.1 | |
| 80 | 1.9 | - 8 BUILDS OVER 6 MONTHS |
| 98 | 2.6 | |
| 78 | 2.3 | - EXPERIENCED RWP/ADA STAFF |
| 135 | 2.3 | |
| 95 | 1.9 | |

16.9 = 2.1 AVERAGE WORK DAYS/SPFR CORRECTION

- o TYPICAL WORK DAYS PER SPFR APPROXIMATELY 1.9 TO 2.3

JPL

# RWP METRICS AND OBSERVATIONS

RWP

## ADA ERROR CORRECTION RATES (CONT'D)

- 96+% OF CORRECTIONS WERE VALID:
  - ∴ WE WERE ABLE TO MAKE ONE ADDITIONAL BUILD PRIOR TO FSATs TO INCREASE RELIABILITY

# RWP METRICS AND OBSERVATIONS

JPL

## ADA PORTABILITY METRICS AND OBSERVATIONS

- PROJECT ESTABLISHED PORTABILITY AS DESIGN OBJECTIVE EARLY
- PERFORMED ANALYSIS USING THREE TOOLS AND LIMITED HUMAN ANALYSIS
  - ADA COMPILER
  - JPL DEVELOPED TOOL:
    - * SEE PAPER BY BORIS SHENKER AND HERNAN GUARDA

      AN AUTOMATED TOOL FOR PORTABILITY ANALYSIS OF ADA CODE OF THE REAL-TIME WEATHER PROCESSOR PROJECT

      PRESENTED AT MINNOWBROOK WORKSHOP, JULY 1990
  - ADA-MAT: FOR VALIDATION

- PORTABILITY HAS THREE LEVELS OF RISK:

| | EFFORT TO CONVERT |
|---|---|
| - LOW | 0 - 2 WORK HOURS |
| - MEDIUM | 2 - 8 WORK HOURS |
| - HIGH | OVER 8 WORK HOURS |

# RWP METRICS AND OBSERVATIONS

JPL

RWP

## ADA PORTABILITY METRICS*

| | X-WINDOWS | NON-PORTABLE |
|---|---|---|
| TOTAL UNITS | 704 | 100% |
| PORTABLE UNITS | 455 | 65% |
| NON-PORTABLE UNITS | 249 | 35% |
| UNITS WITH HIGH RISK CONSTRUCTS | 145 | 21% |
| UNITS ONLY WITH LOW RISK CONSTRUCTS | 41 | 6% |
| TOTAL ADA STATEMENTS (;) | 97,687 | |
| TOTAL NON-PORTABLE CONSTRUCTS: | 12,267 | 100% |
| - HARDWARE | 1,192 | 10% |
| - OPERATING SYSTEM** | 2,290 | 19% |
| - ADA COMPILER | 5,220 | 42% |
| - COMMERCIAL OFF-THE-SHELF (COTS) | 3,565 | 29% |

* DOES NOT INCLUDE DATA ISSUES (E.G. WORD SIZE, STORAGE ISSUES)

** DOES NOT INCLUDE PARAMETER SETTINGS

JPL

# RWP METRICS AND OBSERVATIONS

RWP

## ADA PORTABILITY METRICS

| | X-WINDOWS PORTABLE | % DIFFERENCE |
|---|---|---|
| TOTAL NON-PORTABLE CONSTRUCTS | 11,444 | 7% |
| - HARDWARE | 1,192 | 0 |
| - OPERATING SYSTEM** | 2,290 | 0 |
| - ADA COMPILER | 5,220 | 0 |
| - COTS | 2,742 | 5% |

N92
19429
UNCLAS

N92-19429

# EARLY EXPERIENCES
# BUILDING A SOFTWARE QUALITY PREDICTION MODEL

W. W. Agresti, W. M. Evanco, M. C. Smith

The MITRE Corporation

## Abstract

Early experiences building a software quality prediction model are discussed. The overall research objective is to establish a capability to project a software system's quality from an analysis of its design. The technical approach is to build multivariate models for estimating reliability and maintainability. Data from twenty-one Ada subsystems have been analyzed thus far to test hypotheses about various design structures leading to failure-prone or unmaintainable systems. Current design variables highlight the interconnectivity and visibility of compilation units. Other model variables provide for the effects of reusability and software changes. Reported results are preliminary because additional project data is being obtained and new hypotheses are being developed and tested. Current multivariate regression models are encouraging, explaining 60-80% of the variation in error density of the subsystems.

## Introduction

A typical shortcoming of large-scale software development is the uncertainty concerning the consequences of design decisions until much later in the development process. Greater capability is needed during the design activity to assess the design itself for indications that, when implemented, the resulting system will have particular quality characteristics. This paper discusses the early experiences in a research project to evaluate the quality of Ada designs.

The research objective is to test the hypothesis that Ada software quality factors can be predicted during design. The technical approach is build

- This research was sponsored by The MITRE Corporation under the Mission Oriented Investigation and Experimentation (MOIE) program

- Authors' Address: The MITRE Corporation, 7525 Colshire Drive, McLean, Virginia 22102

Proceedings of the Fifteenth Annual Software Engineering Workshop,
National Aeronautics and Space Administration, Goddard Space Flight Center,
Greenbelt, Maryland, November 1990

multivariate models to estimate reliability and maintainability using characteristics of the design. The orientation to Ada is due to its prevalence in mission-critical systems under development and its ability to serve as a notation for software design. This role for Ada as a design language has been recognized as American National Standards Institute (ANSI)/Institute of Electrical and Electronics Engineers (IEEE) Standard 990-1987.

Previous research has established relationships between design or code characteristics and quality factors [1, 2]. A recent system-level design measure, incorporating both control flow and data flow in FORTRAN systems, shows a strong correlation with reliability [3]. The COnstructive QUAlity MOdel (COQUAMO) is being developed to estimate software quality, basing its estimate on the observed quality of previous projects [4].

## Quality Estimation Models

Building the estimation models depends on having access to three classes of project data:

- Design, expressed in Ada, from which design characteristics can be extracted
- Environmental factors that influence the quality of the software but cannot be deduced from the design artifact itself -- for example, level of reuse or volatility of changes to the software
- Data characterizing what resulted when the design was implemented, tested, and fielded -- for example, reported errors and effort to maintain the software

The basic form of the estimation models is shown in Figure 3. Independent, explanatory variables in the models represent architectural design characteristics. Additional explanatory variables account for the effects of the organization and its development process. By the error term in the model, we will learn if we have been successful in explaining the variation in quality factors by using design characteristics and environmental factors.

## Ada Design Representation

One of the first issues we faced was developing an effective representation from which we could extract design characteristics. Our interest was in the static architecture of units in subsystems, not in the arrangement of statements within a unit. We viewed the subsystem as being composed of design units and relations as illustrated in Figure 4. Our analysis of Ada identified several candidates to serve as design units in our structure: program units, compilation units, and library units. All three units have participated in our model building, but compilation units have been particularly useful as a structural unit because they also serve as the unit of observation for reporting errors and changes.

Our Ada analysis identified fifteen kinds of Ada compilation units: generic package specification, generic package body, and so on as shown in Figure 5. The compilation units are further divided into library units and secondary units (see Figure 5) and serve as the design unit nodes in the graphical Ada design structures in Figure 4. The nodes are related to one another by the design relations of context coupling, specification/body, parent/subunit, and generic template/instantiation. These design units and design relations comprise our representation of static Ada architecture. This Ada design representation is discussed further in [5].

Software Project Data

Project data used in the analysis is summarized in Figure 6. The twenty-one subsystems included 2,143 compilation units. Declarations are listed in Figure 6 because they play a key role in the hypotheses we are examining. One of our underlying themes is that a developer does not declare objects, types, subprograms, etc. unless they are needed. Thus, the number and distribution of these declarations is of interest to us in characterizing designs.

Our models attempt to explain variation in quality, and Figure 6 shows our project data exhibits significant variation. The data was obtained from the National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center (GSFC) Software Engineering Laboratory (SEL). Reliability is measured as error density and varies in the range 1.4 - 17.0 errors per thousand source lines of code for the twenty-one subsystems. Maintainability varied across the subsystems as 26 - 89% error corrections requiring less than or equal to one hour to complete.

Hypotheses About Design Structure

We are pursuing simple hypotheses about design decision making, the resulting design artifact, and the influences of design on reliability and maintainability . Figure 7 outlines an example of a general hypothesis that excessive context coupling contributes to errors. The rationale is that greater arc density in the directed graph in Figure 7 increases the likelihood of introducing an error, because a greater number of relationships must be understood.

Figure 8 expands on library unit B of Figure 7. We have found that a library unit aggregation – a library unit and its declarative scope – to be a very effective unit of granularity for our analysis of Ada designs. Figure 8 shows a second level of design decision making that occurs inside a library unit aggregation. We are interested in whether the designer has made any effort to manage the visibility of the 103 declarations that have been imported into unit B. By having 100 of the declarations brought in (via a "with" clause) to the specification, they are visible throughout the other units in the library unit aggregation, cascading through the structure. We don't know which of these declarations are used by each unit, but we want to record their visibility to the other units in the library unit aggregation. The measure of cascaded imports in Figure 8 takes visibility into account: 100 of the imports are visible to five units ( => 500 cascaded imports) and three of the imports are visible to two units ( => 6 cascaded imports), for a total of 506 cascaded imports.

## Preliminary Results of Statistical Analyses

Figure 9 summarizes the variables that have been introduced into our models thus far. Context coupling and visibility follow the example in Figure 8, while import origin records the fraction of declarations imported from within the subsystem. Two environmental factors have been analyzed to date: volatility captures the relative number of changes that have been made in the subsystem, and custom code is the percentage of new and extensively modified code used in the subsystem. Custom code is essentially the opposite of reuse.

The preliminary model explaining variation in total error density (Figure 10) includes the explanatory variables of context coupling, visibility, and volatility. In this model and other similar regression analyses we have conducted, the coefficients have the expected signs: error densities increase as context coupling, visibility, and volatility increase.

Because of our interest in architectural design decisions, we conducted additional regression analyses which concentrated on errors occurring during system and acceptance testing. Our rationale was that, by eliminating errors reported during unit testing (and, therefore, more likely to be errors in implementing a single unit), we were reflecting more strongly the architectural (inter-unit) relationships. Figure 11 summarizes a model to estimate errors reported during system and acceptance testing. Again, context coupling and visibility are included as explanatory variables. Now, however, *custom code is a significant factor in explaining the variation in error density.* The explanatory power (as indicated by the coefficient of determination) is stronger for the model in Figure 11.

## Summary

Early results in building estimation models for reliability and maintainability are encouraging. We have developed representations for the static structure of Ada systems using compilation units and library unit aggregations, allowing us to test hypotheses about the effects of different structures on reliability and maintainability . Context coupling measures consistently figure strongly in the multivariate regression analyses we have conducted. Visibility and import origin measures provide further refinement. The models show strong effects of volatility and custom code on reliability .

We stress the preliminary nature of the quantitative results, based as they are on twenty-one Ada subsystems. We look forward to continuing to explore hypotheses with additional data, leading to the development of more robust models that can be subjected to validation.

## Acknowledgement

We acknowledge the cooperation of Mr. Frank McGarry and Mr. Jon Valett of the NASA/GSFC SEL in allowing us to use SEL data for this research.

## References

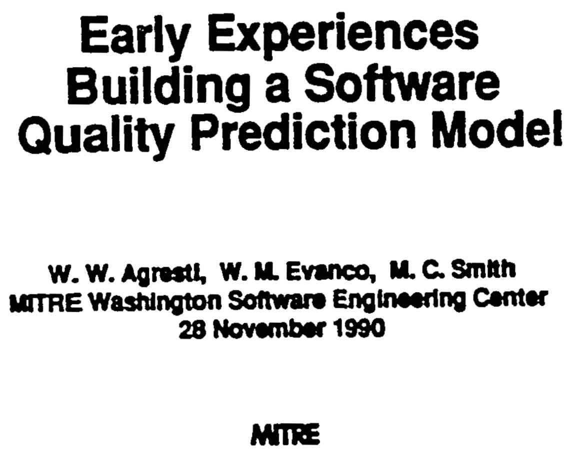
1. T. J. McCabe and C. W. Butler, "Design Complexity Measurement and Testing," Communications of the ACM, December 1989, pp. 1415-1425.

2. S. M. Henry and C. A Selig, "Predicting Source-Code Complexity at the Design Stage," IEEE Software, March 1990, pp. 36-44.

3. D. N. Card and W. W. Agresti, "Measuring Software Design Complexity," Journal of Systems and Software, March 1988, pp. 185-197.

4. B. Kitchenham, "Measuring Software Quality," Proceedings of the First Annual Software Quality Workshop, Rochester, New York, 1989.

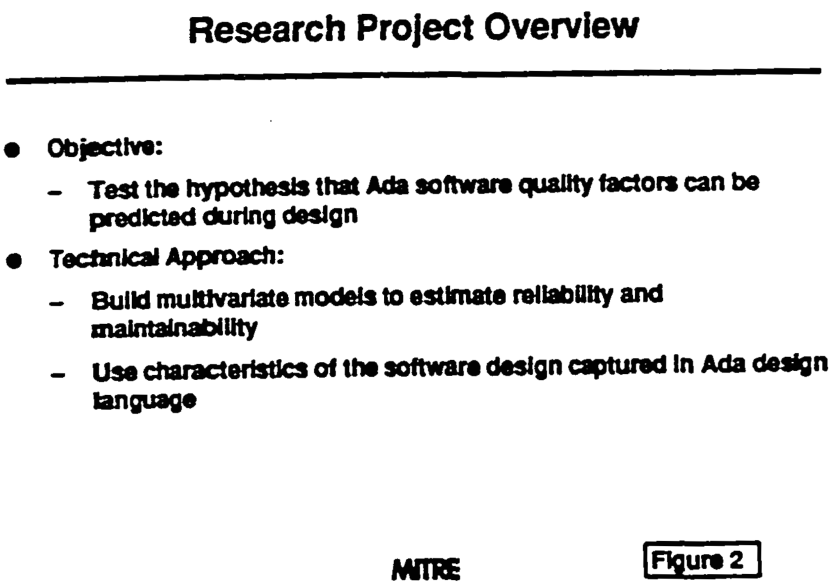
5. W. W. Agresti, W. M. Evanco, and M. C. Smith, "An Approach to Software Quality Prediction from Ada Designs," Proceedings of the Second Annual Software Quality Workshop, Rochester, New York, 1990.



# Early Experiences Building a Software Quality Prediction Model

W. W. Agresti, W. M. Evanco, M. C. Smith
MITRE Washington Software Engineering Center
28 November 1990

MITRE



## Research Project Overview

- Objective:
  - Test the hypothesis that Ada software quality factors can be predicted during design
- Technical Approach:
  - Build multivariate models to estimate reliability and maintainability
  - Use characteristics of the software design captured in Ada design language

MITRE

Figure 2

## Basic Form of the Estimation Models

Reliability $= f_1(DC_1, DC_2, \ldots, EF_1, EF_2, \ldots \mid a_1, a_2, \ldots) + e_1$

Maintainability $= f_2(DC_1, DC_2, \ldots, EF_1, EF_2, \ldots \mid b_1, b_2, \ldots) + e_2$

where -

$DC_i$ : design characteristic variable

$EF_i$ : environmental factor variable

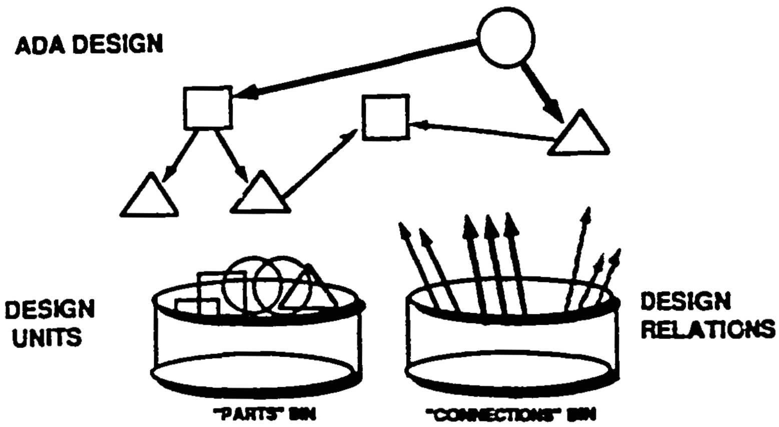
$a_i, b_i$ : model parameters

$e_i$ : error term (unexplained variation)

MITRE

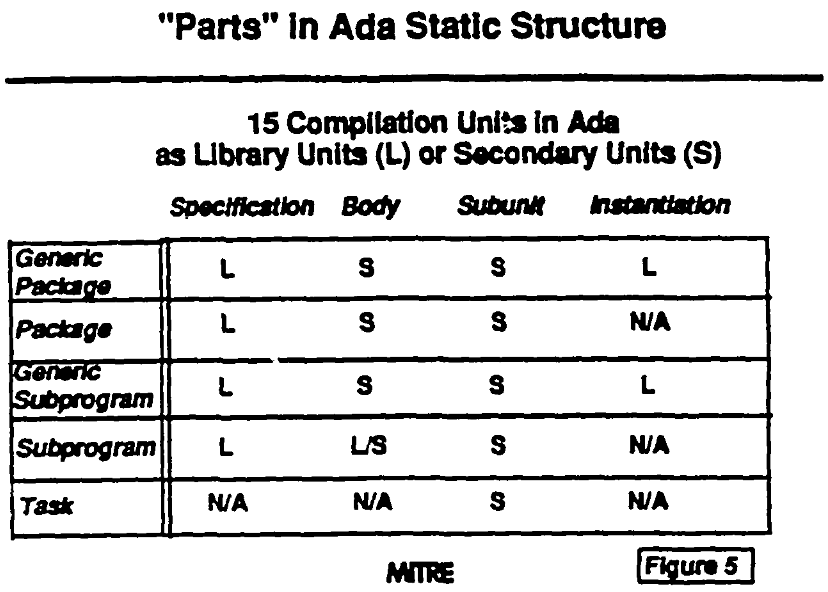
Figure 3

## Representing Ada Design Structure



MITRE

Figure 4



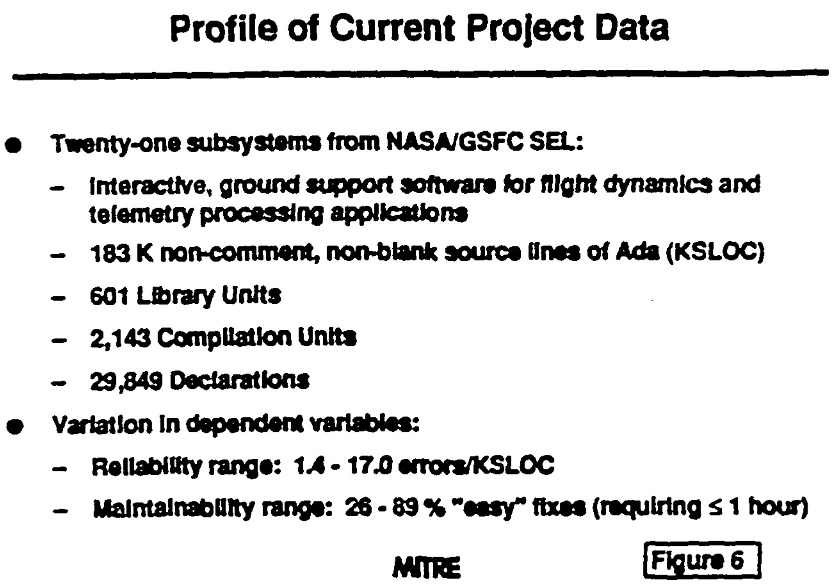
## "Parts" in Ada Static Structure

**15 Compilation Units in Ada as Library Units (L) or Secondary Units (S)**

| | *Specification* | *Body* | *Subunit* | *Instantiation* |
|---|---|---|---|---|
| *Generic Package* | L | S | S | L |
| *Package* | L | S | S | N/A |
| *Generic Subprogram* | L | S | S | L |
| *Subprogram* | L | L/S | S | N/A |
| *Task* | N/A | N/A | S | N/A |

MITRE

Figure 5



## Profile of Current Project Data

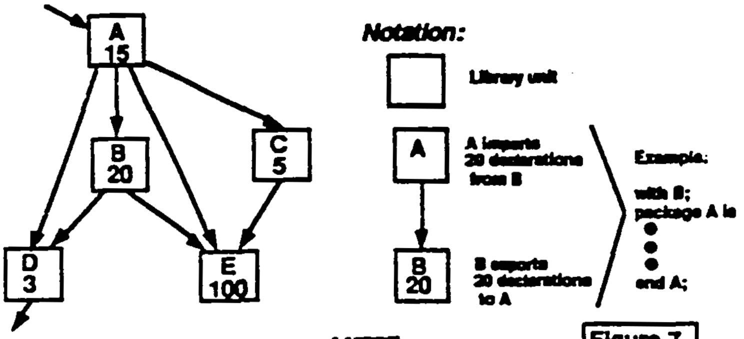
- Twenty-one subsystems from NASA/GSFC SEL:
  - Interactive, ground support software for flight dynamics and telemetry processing applications
  - 183 K non-comment, non-blank source lines of Ada (KSLOC)
  - 601 Library Units
  - 2,143 Compilation Units
  - 29,849 Declarations
- Variation in dependent variables:
  - Reliability range: 1.4 - 17.0 errors/KSLOC
  - Maintainability range: 26 - 89 % "easy" fixes (requiring ≤ 1 hour)

MITRE

Figure 6

## Exploring Simple Hypotheses About Design Structure

- Example of a general hypothesis: Excessive context coupling contributes to complexity which, in turn, contributes to errors
- Example of context coupling to access the resources of library units:

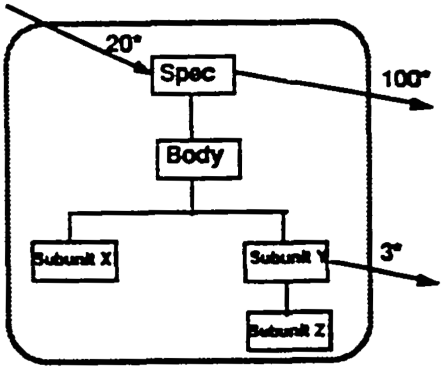


MITRE

Figure 7

## Inside a Library Unit Aggregation to Show Imported and Exported Declarations

A Library Unit Aggregation



Static Measures:

# imports = 103
# exports = 20
# cascaded imports = 506

* number of declarations

MITRE

Figure 8

## Model Variables

- Design Characteristics:
  - Context Coupling: # imports / # exports
  - Visibility: # cascaded imports / # imports
  - Import Origin: # internal imports / # imports
- Environmental Factors:
  - Volatility: # changes / # library units
  - Custom Code: % new and extensively modified code

MITRE

Figure 9

## Preliminary Model for Reliability: Total Errors (errtot)

- Dependent variable: TOTERRSL = errtot / KSLOC

$$\begin{array}{lcccc} \ln(\text{TOTERRSL}) = & .65 & +\ .27 \ln(X_1) & +\ .05 \ln(X_2) & +\ .27 \ln(X_3) \\ & (.36)^{*} & (.11) & (.16) & (.08) \end{array}$$

$X_1$ = context coupling

$X_2$ = visibility

$X_3$ = volatility

adjusted $R^2 = .72$

[*] Standard deviation of the parameter estimate

MITRE

Figure 10

## Preliminary Model for Reliability: System and Acceptance Testing Errors (errsa)

- Dependent variable: SYACERRSL = errsa / KSLOC

$$\begin{array}{lcccc} \ln(\text{SYACERRSL}) = & .77 & +\ .19\ \ln(X_1) & +\ .07\ \ln(X_2) & +\ .97\ \ln(X_3) \\ & (.65)^{*} & (.18) & (.21) & (.24) \end{array}$$

$X_1$ = context coupling
$X_2$ = visibility
$X_3$ = custom code

adjusted $R^2 = .78$

* Standard deviation of the parameter estimate

MITRE

Figure 11

## Current Research Activity

- Continue to develop process models and hypotheses about design decision-making and design structures -- and their relationships to reliability and maintainability
- Explore classification trees and other alternative analytical methods
- "Call for Ada Project Data" -- to test hypotheses and calibrate multivariate models

MITRE

Figure 12

# VIEWGRAPH MATERIALS

# FOR THE

# W. AGRESTI PRESENTATION

# Early Experiences Building a Software Quality Prediction Model

W. W. Agresti, W. M. Evanco, M. C. Smith
MITRE Washington Software Engineering Center

28 November 1990

MITRE

# Research Project Overview

- Objective:
  - Test the hypothesis that Ada software quality factors can be predicted during design
- Technical Approach:
  - Build multivariate models to estimate reliability and maintainability
  - Use characteristics of the software design captured in Ada design language

MITRE

Figure 2

# Basic Form of the Estimation Models

$$\text{Reliability} = f_1(DC_1, DC_2, \ldots, EF_1, EF_2, \ldots \mid a_1, a_2, \ldots) + e_1$$

$$\text{Maintainability} = f_2(DC_1, DC_2, \ldots, EF_1, EF_2, \ldots \mid b_1, b_2, \ldots) + e_2$$

where -

$DC_i$ : design characteristic variable

$EF_i$ : environmental factor variable

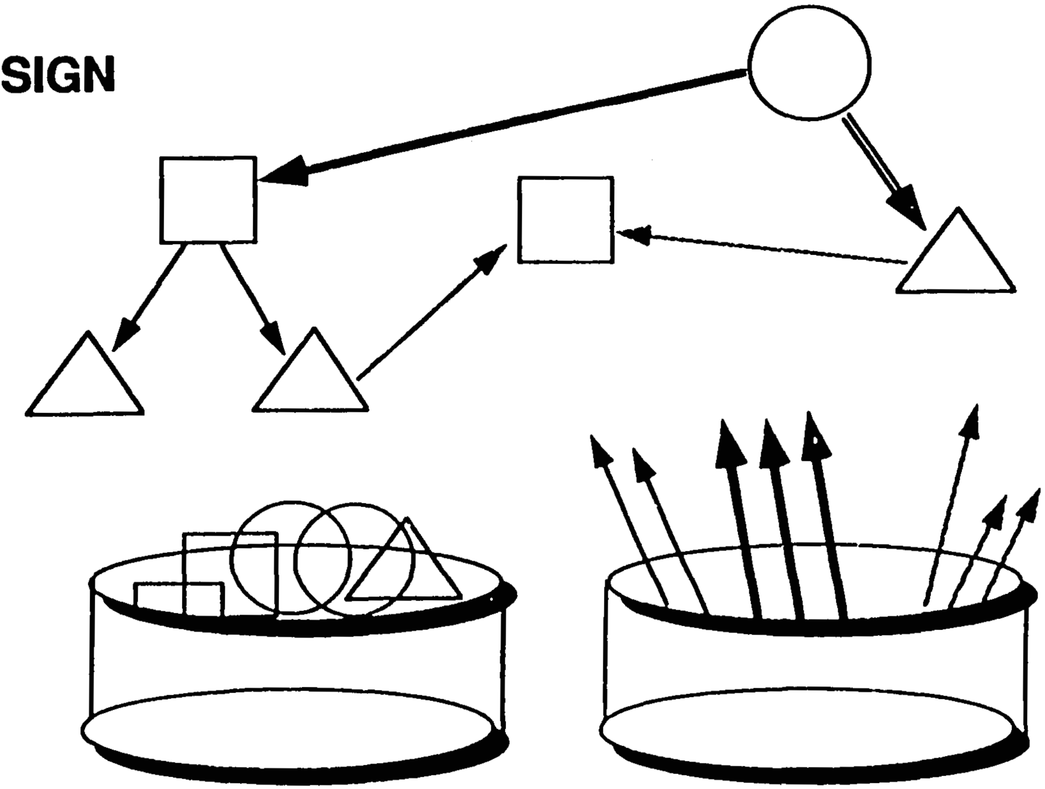
$a_i$, $b_i$ : model parameters

$e_i$ : error term (unexplained variation)

Figure 3

# Representing Ada Design Structure

ADA DESIGN



DESIGN UNITS

DESIGN RELATIONS

"PARTS" BIN

"CONNECTIONS" BIN

W. Agresti
MITRE
Page 15 of 23

Figure 4

# "Parts" in Ada Static Structure

## 15 Compilation Units in Ada as Library Units (L) or Secondary Units (S)

| | *Specification* | *Body* | *Subunit* | *Instantiation* |
|---|---|---|---|---|
| *Generic Package* | L | S | S | L |
| *Package* | L | S | S | N/A |
| *Generic Subprogram* | L | S | S | L |
| *Subprogram* | L | L/S | S | N/A |
| *Task* | N/A | N/A | S | N/A |

MITRE

Figure 5

# Profile of Current Project Data

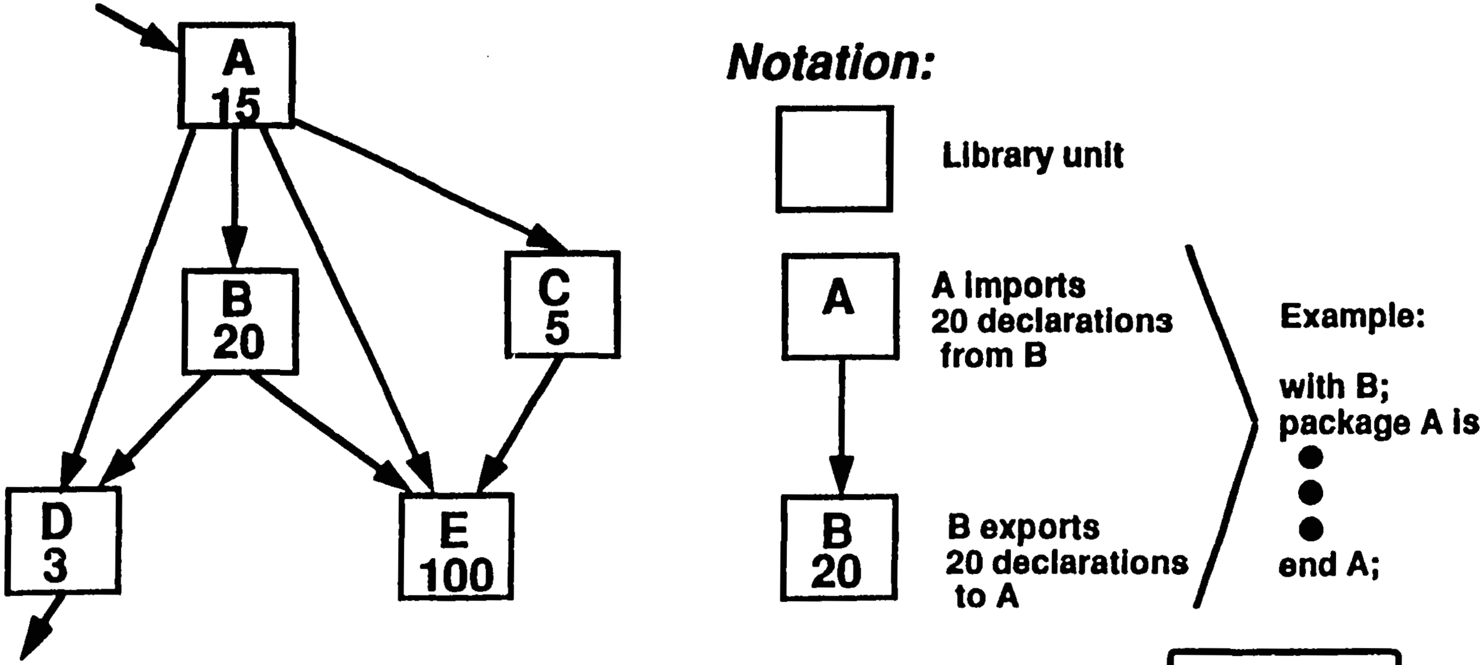
- Twenty-one subsystems from NASA/GSFC SEL:
  - Interactive, ground support software for flight dynamics and telemetry processing applications
  - 183 K non-comment, non-blank source lines of Ada (KSLOC)
  - 601 Library Units
  - 2,143 Compilation Units
  - 29,849 Declarations
- Variation in dependent variables:
  - Reliability range: 1.4 - 17.0 errors/KSLOC
  - Maintainability range: 26 - 89 % "easy" fixes (requiring ≤ 1 hour)

MITRE

Figure 6

# Exploring Simple Hypotheses About Design Structure

- Example of a general hypothesis: Excessive context coupling contributes to complexity which, in turn, contributes to errors
- Example of context coupling to access the resources of library units:



*Notation:*

Library unit

A

A imports 20 declarations from B

B
20

B exports 20 declarations to A

Example:

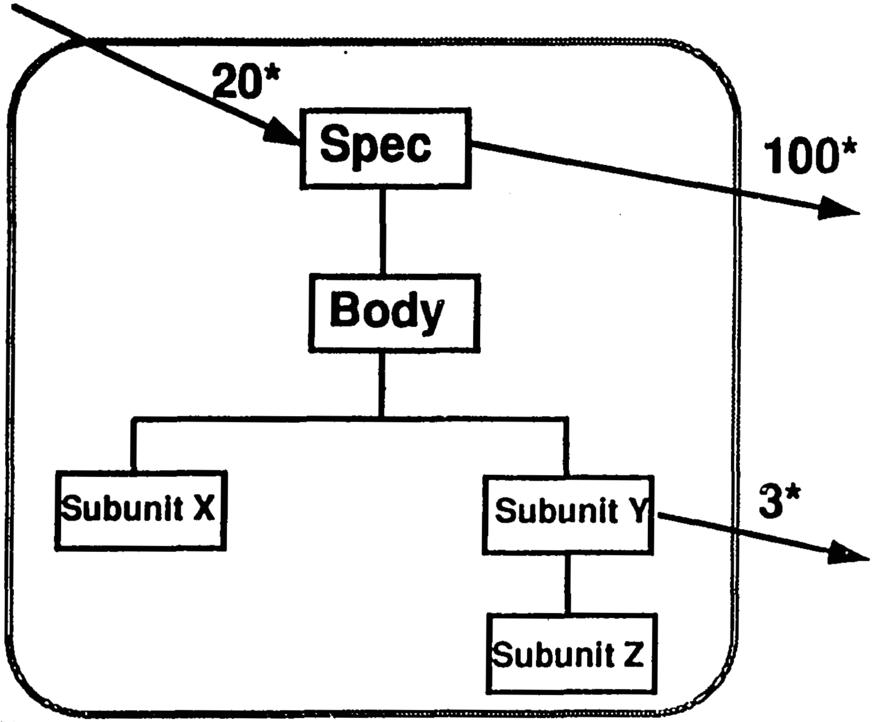
with B;
package A is
●
●
●
end A;

MITRE

Figure 7

# Inside a Library Unit Aggregation to Show Imported and Exported Declarations

## A Library Unit Aggregation



* number of declarations

**Static Measures:**

# imports = 103
# exports = 20
# cascaded imports = 506

MITRE

Figure 8

# Model Variables

- **Design Characteristics:**
  - **Context Coupling:** # imports / # exports
  - **Visibility:** # cascaded imports / # imports
  - **Import Origin:** # internal imports / # imports
- **Environmental Factors:**
  - **Volatility:** # changes / # library units
  - **Custom Code:** % new and extensively modified code

Figure 9

# Preliminary Model for Reliability: Total Errors (errtot)

- Dependent variable: TOTERRSL = errtot / KSLOC

$$\ln(\text{TOTERRSL}) = \underset{(.36)^*}{.65} + \underset{(.11)}{.27} \ln(X_1) + \underset{(.16)}{.05} \ln(X_2) + \underset{(.08)}{.27} \ln(X_3)$$

$X_1$ = context coupling

$X_2$ = visibility

$X_3$ = volatility

adjusted $R^2 = .72$

* Standard deviation of the parameter estimate

MITRE

Figure 10

# Preliminary Model for Reliability: System and Acceptance Testing Errors (errsa)

- Dependent variable: SYACERRSL = errsa / KSLOC

$$\ln(\text{SYACERRSL}) = \underset{(.65)^*}{.77} + \underset{(.18)}{.19} \ln(X_1) + \underset{(.21)}{.07} \ln(X_2) + \underset{(.24)}{.97} \ln(X_3)$$

$X_1$ = context coupling

$X_2$ = visibility

$X_3$ = custom code

adjusted $R^2 = .78$

* Standard deviation of the parameter estimate

MITRE

Figure 11

# Current Research Activity

- Continue to develop process models and hypotheses about design decision-making and design structures -- and their relationships to reliability and maintainability
- Explore classification trees and other alternative analytical methods
- "Call for Ada Project Data" -- to test hypotheses and calibrate multivariate models

Figure 12

N92
19430
UNCLAS

# Bias and Design in Software Specifications[*]

Pablo A. Straub[†]
Computer Science Department

Marvin V. Zelkowitz
Computer Science Department and
Institute for Advanced Computer Studies

University of Maryland
College Park, Maryland 20742

December 19, 1990

**Abstract**

Implementation bias in a specification is an arbitrary constraint in the solution space. While bias is a recognized problem, it has not been studied in its own right. This has resulted in two effects: Either (1) specifications are biased, or (2) they are incomplete, for fear of bias. In fact, what has been called "bias" in the literature is sometimes the desirable record of design constraints and design decisions.

This paper presents a model of bias in software specifications. Bias is defined in terms of the specification process and a classification of the attributes of the software product. Our definition of bias provides insight into both the origin and the consequences of bias: it also shows that bias is relative and essentially unavoidable. Finally we describe current work on defining a measure of bias, formalizing our model, and relating bias to software defects.

**Keywords.** Implementation bias, software design, software defects, requirements, formal specifications.

## 1 Introduction

Most informal software specifications are ambiguous, imprecise, unclear, incomplete, etc. Moreover, this is usually not evident by looking at a particular specification.

[*] This research is supported in part by NASA Goddard Space Flight Center, grant NSG-5123.
[†] Additional support from ODEPLAN Chile and the Catholic University of Chile.

The need to produce better specifications has prompted research in several features of specifications. Guttag and Horning [2] define *sufficient-completeness* and *consistency* of an algebraic specification in terms of existence and uniqueness of axioms in the specification. Jones [3] defines *bias* for model-based specifications as the property of nonuniqueness of representation within the model. Yue [8] gives a definition for *completeness* of a specification, in terms of the satisfaction of a set of explicitly stated goals. He also defines *pertinence*, a property related to bias. Nicholl [5] defines the concept of *reachability* for model-based specifications as the ability to reach every consistent state by some sequence of operations, and plans to study other features of specifications, including bias.

This current research work grew out of studies within the Software Engineering Laboratory (SEL) of NASA Goddard Space Flight Center, which has been monitoring the development of ground support software for unmanned spacecraft since 1976. Our goal is to improve the quality of software specifications within the SEL. On the realization that existing specification languages were inappropriate for use by programmers at NASA, we developed the executable specification language PUC (pronounced POOK), designed to be used with Ada in this environment [7].

The executability of specification languages like PUC had the disadvantage that much detail had to be included in specifications, limiting the creativity of the implementor and ruling out some possibly good designs. Hence, instead of looking at the language problem, now we are looking at this problem itself, the so-called 'implementation bias' in specifications. The area of bias in specifications is largely unexplored but is important. In fact. the problem of bias is mentioned in several works, including both description and critiques of specification methods.

## 1.1 Definitions

Some related concepts are defined.

**Attribute** An *attribute* of a product is a required or desired feature of the product, its environment, or its development process.

We use 'attribute' instead of the more customary 'requirement', because the latter is associated with mandatory features that are described in the initial phases of development.

**Specification** A *specification* of a product is a description of a set of its attributes.

Under this definition, both the *requirements document* and the *preliminary design document* of the waterfall model are specifications.

**Solution set** The *solution set* of a problem is the set of all products that solve the problem, regardless of the specification.

## 1.2 The Problem of Bias

An ideal specification is general and precise enough so that a software system satisfies the specification if and only if it solves the problem at hand. This view is too optimistic, because there can be many solutions to the real world problem that do not even involve software. In practice, we only need that software systems satisfying the specification be solutions, and that no substantial class of solutions does not satisfy the specification.

A specification is biased if it arbitrarily favors some implementations over others. Biased specifications can overly constrain the solution set, precluding some valid implementations as solutions to the problem at hand. Hence, the amount of bias is a common yardstick to judge software specification methods: those that are considered biased are usually rejected.

One of the main problems of not having a good definition of *bias* is that it is sometimes confused with intended constraints in the solution set. For example, a designer may want to favor some realizations over others for compliance with some programming techniques that are customary at that site. In fact, we argue that much of what has been called bias is simply a manifestation of design decisions, that purposely constrain the solution set. Of course, we also have many specifications that are indeed biased.

# 2 A Model of Bias

We present a framework to discuss bias, based on a classification of the attributes of the product being specified and the process of creation of attributes.

## 2.1 Classes of Attributes

We classify the attributes of a product with respect to their inclusion in the specification. The main criteria we consider are explicitness and origin.

### 2.1.1 Explicitness

An attribute is *explicit* if it is present in the specification; otherwise, it is *nonexplicit*.

Nonexplicit attributes are further classified in four classes.

**Implicit** attributes are those that are understood to be part of every product in the application domain, and so they are unstated.

**Implied** attributes are logical consequences of other attributes.

**Absent** attributes are requirements unintentionally omitted in the specification. These are not part of every product in the application domain.

**Fictitious** attributes [4] are not attributes at all, but assumptions made by the reader of the specification: the reader believes that they are either implicit, implied or absent attributes.

### 2.1.2 Origin

An explicit attribute is *new* with respect to a certain specification stage if it is first made explicit at that stage; otherwise, the attribute is *inherited* from previous stages.

In an ideal setting all attributes new in a specification stage are the consequence of design decisions taken at that stage. However, nonexplicit attributes in the previous specification usually induce the specifier of the current stage to introduce extra attributes. Besides, some attributes may be imposed by the limitations of the specification method and language used. This motivates the following classification of new attributes with respect to their origin.

**Designed** attributes are the consequence of design decisions taken at the current specification stage. They are purposely set to guide the implementation process and constrain the solution set.

**Explicatory** attributes are created by making explicit attributes that are implicit in, implied by, or absent from previous stages.

**Imposed** attributes are those imposed by the limitations of the specification method and language used.

For example, a method may accept only "complete" specifications (as defined by the language), which leads to introduce attributes to satisfy the rules of the language.

**Extraneous** attributes are created by making explicit fictitious attributes.

For example, a fictitious attribute seen by the designer in a requirements document may introduce explicit constraints in the design document.

## 2.2 The Nature of Bias

The process of refining successive specifications makes explicit attributes that were previously implicit, implied, or absent. This process also makes explicit design decisions taken at the current stage. Unfortunately, it also makes explicit fictitious attributes (i.e., creates extraneous attributes[1]) and creates imposed attributes (Figure 1). This leads to the definition of bias in terms of the origin of the attributes described in a specification.

[1] Extraneous attributes lead to errors and constraints in the solution set; here we are studying only the constraints.

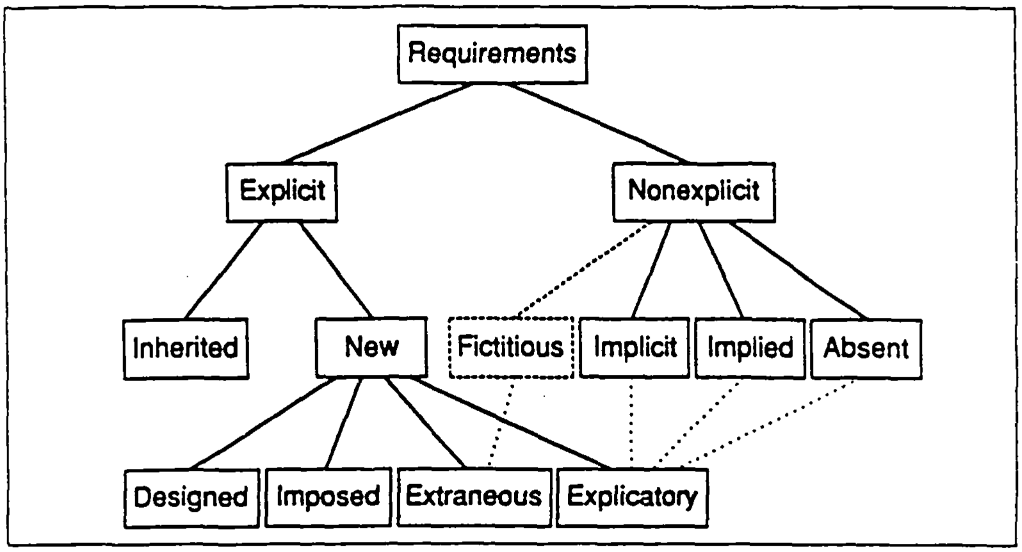


Figure 1: Classification of attributes. Fictitious attributes are shown with segmented line, because they are not real attributes but misconceptions. Dotted lines show the origin of new explicatory and extraneous attributes.

Definition. A specification containing extraneous or imposed attributes is *biased.*

This definition provides insight into the problem of bias, including both its origins and consequences. The origin of bias is either wrongful interpretation of nonexplicit attributes or the limitations imposed by the specification method. The consequences are that the set of possible solutions can be overly constrained or that the solution adopted can be suboptimal. That is, a biased specification will lead the design towards particular implementations that are not necessarily the best possible.

Bias content in a specification cannot be measured directly, because bias is defined in terms of the origin of attributes which is usually uncertain. Furthermore, bias is relative to the application domain and the software engineering environment, because the domain and environment define what is implicit.

The relative nature of bias is an essential characteristic. It stems from the existence of nonexplicit attributes and the inherent uncertainty with respect to those attributes. As long as there are nonexplicit attributes, there will be doubt about these attributes and hence possibility of bias. Furthermore, making explicit all implicit attributes of a certain domain and environment still leaves two sources of bias: restrictions on the method and languages, and absent attributes.

## 2.3 Example

Assume an environment in which all programs are written in a particular programming language. In this environment the presence of idioms of this language in a specification is not necessarily bias, unless another implementation language is introduced to the environment.

This is what happened at the SEL where software specifications for satellite dynamic simulators were “heavily biased toward FORTRAN. In fact the high level design for the simulators is actually in the specifications document” [1]. This was not a problem—on the contrary, it facilitated both development and reuse of specification and code—until the first development in Ada: the specifications had to be rewritten first.

Given our definition of bias these FORTRAN-oriented specifications were not necessarily biased; they contained many designed attributes. Before Ada was introduced, the use of FORTRAN was implicit. After that, the language used had to be decided: assuming a FORTRAN implementation was a fictitious attribute.

# 3 Current Research

We are improving the model presented in this paper in several aspects.

**Formalization** One weakness of our model as presented here is that we do not formalize the concept of ‘attribute’. Moreover, we define ‘specification’ as a set of attributes, disregarding dependencies among attributes. At least two kinds of dependencies are relevant: attributes defined in terms of other attributes, and origin relationships among attributes.

To address this problem, we have develped a formalism to write specifications that is flexible and extensible enough to include information about the specification itself (e.g., origin information). Within the system, called Extensible Description Formalism (EDF), attributes are defined as mappings from objects to values; objects are represented by extensible polymorphic records whose fields are the attribute names. EDF can represent both functional dependencies of attributes and also attributes defined as aggregations of several attributes. Origin information is stored by representing all attribute values as objects that have an *origin* attribute and a *content* attribute.

We developed a prototype of EDF and used it in the context of classification of reusable software components. We are currently developing a complete version based on a formal specification of the language [6].

**Measuring Bias** In this work we have not provided a characterization of biased specifications. Because of the relative nature of bias we cannot develop a precise metric of bias, but we can define approximate metrics, based on origin information explicitly recorded in a specification.

An important feature of EDF is that it is possible to compare two specifications defining some *distance* from one specification to another. There is a predefined comparator function to estimate the adaptation effort in the context of reuse of software components, and it is possible to define other comparator functions.

We can measure bias comparing the distance between two succesive specification stages. If we use the predefined comparator function we get a gross upper bound on bias (as if all attributes new to the second stage were bias). On the other hand, by defining a comparator function that uses origin information, we expect to provide a reasonable estimate of bias introduced in the second stage.

**Bias Propagation** Our model does not explain how bias propagates, because we have defined bias in terms of new attributes. Strictly speaking, within our model no inherited attribute is bias. Since we want to measure bias content in a specification, we have to consider those attributes whose origin include extraneous or imposed attributes. For example, if a design decision is taken consistently with some inherited attribute that was extraneous when created, then this decision has some form of bias too.

**Bias and Software Defects** Our model describes the origin for software attributes, and defines bias as the existence of some attributes with 'illegitimate' origin. The reader can realize that these illegitimate origins are also the cause of software defects.

Software defects are classified in three groups: *errors* are conceptual misunderstandings, *faults* are concrete (explicit) manifestations of errors in documents, and *failures* are manifestations of faults during execution.

There is an intimate relashionship between errors and fictitious attributes, and between software faults and bias. In a sense, bias is like a very minor fault that instead of leading to failures, leads to inefficiencies. The consequence of this is that methods to avoid bias (e.g., making explicit implicit requirements) will also avoid software defects.

## 4 Conclusion

Even though bias is widely recognized as an undesirable property of specifications, it has not been adequately studied. This has caused confusion with the related concepts of design constraint and design decision, so that the presence of designed attributes in specifications has been considered undesirable. This is in contrast with the use of specifications in other engineering disciplines, where a specification may include many designed attributes (e.g., materials, manufacturing methods).

In this paper we presented a model to describe the nature of bias and distinguish bias from designed attributes and other attributes in a specification. This model is based on a classification of all the attributes described in a specification and also those that are not described (i.e., nonexplicit); it explains the nature of bias, but since it uses nonexplicit attributes it does not lead to any definite method to detect bias. However, the model does explain both the relative and unavoidable nature of bias. Moreover, because the model explains the origin of bias, it provides insight into bias avoidance.

Our goal is to improve the quality of the specifications by removing bias and including all relevant implementation-oriented information. To achieve this goal we need to tell bias from designed attributes. This requires information on the origin of the attributes, which is usually unknown. Hence, we have developed a formalism in which origin information can be recorded, as a generalization of the common practice of tracing design documents and actual code back to the original statement of requirements.

## Acknowledgements

Thanks to Sergio Cárdenas-García and Eduardo Ostertag for their helpful comments. The Extensible Description Formalism (EDF) was defined jointly with Eduardo Ostertag, who is also the implementer.

## References

[1] Carolyn E. Brophy, W.W. Agresti, and Victor R. Basili. Lessons learned in use of Ada-oriented design methods. In *Proceedings of the Joint Ada Conference*, March 1987.

[2] John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[3] Cliff B. Jones. Systematic program development. In N. Guehani and A.D. McGettrick, editors, *Software Specification Techniques*. Addison Wesley, Reading, Massachusetts, 1986.

[4] Edward V. Krick. *An Introduction to Engineering and Engineering Design*. John Wiley and Sons, New York, N.Y., second edition, 1969.

[5] Robin A. Nicholl. Unreachable states in object-oriented specifications. *IEEE Transactions on Software Engineering*, 16(4):472–477, April 1990.

[6] Pablo A. Straub and Eduardo J. Ostertag. Semantics of the Extensible Description Formalism. Technical Report CS-TR-2561, UMIACS-TR-90-137, University of Maryland, Department of Computer Science, November 1990.

[7] Pablo A. Straub and Marvin V. Zelkowitz. PUC: A functional specification language for Ada. In *X International Conference of the Chilean Computer Science Society*, pages 111–122, Santiago, Chile, July 1990.

[8] Kaizhi Yue. What does it mean to say a specification is complete? In *Fourth Int'l Workshop on Software Specification and Design*, pages 42–49, Los Alamitos, California, 1987. CS Press.

# VIEWGRAPH MATERIALS

# FOR THE

# P. STRAUB PRESENTATION

6269-0

# Bias and Design in Software Specifications

Pablo A. Straub    Marvin V. Zelkowitz
Computer Science Department
Institute for Advanced Computer Studies
University of Maryland at College Park

## Contents

- Introduction
- Classification of requirements
- The nature of bias
- Conclusions

# Introduction
## Importance of specifications

Life-cycle models consist of refinement of succesive specification stages.

**Specification**: description of a set of requirements.

'Requirement' is used in all stages, not just the first.

Staged specifications imply

- errors in previous stages are costly
- product quality depends on specification

We need high quality specifications.

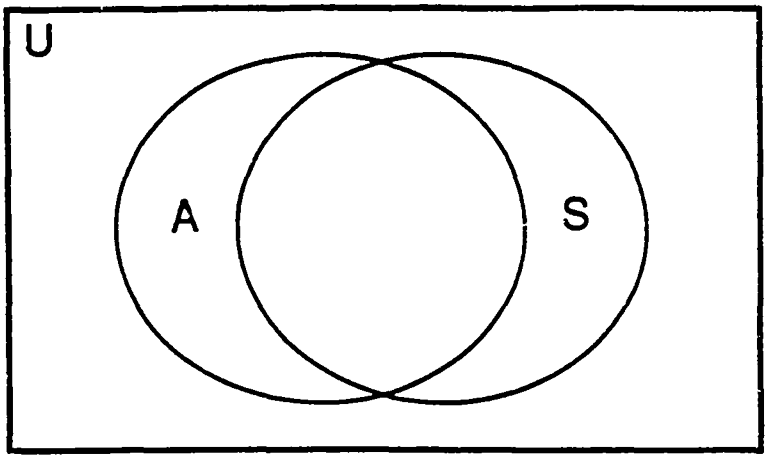
## Introduction
## We want specifications that are...

- abstract
- complete
- consistent
- correct
- reusable
- traceable

our focus

- concise
- executable
- feasible
- formal
- modifiable
- realizable
- structured
- verifiable

## Introduction
## Solutions vs. specified products



$U$ = product universe

$A$ = acceptable products (solutions)

$S$ = specified products

$S - A$ = specified unacceptable products

$A - S$ = solutions not specified

Ideally: $S = A$

Needed: $S - A = \emptyset$

Desired: $A - S$ is *small*

# Introduction
## The *what* and *how* dilemma

Typical rule to avoid overspecification

> Specify *what* the system should do,
> not *how* to do it.

But *what*'s and *how*'s depend on viewpoint.

**What:** something already fixed

**How:** an option

*How*'s become *what*'s.

Confusion creates underspecification.

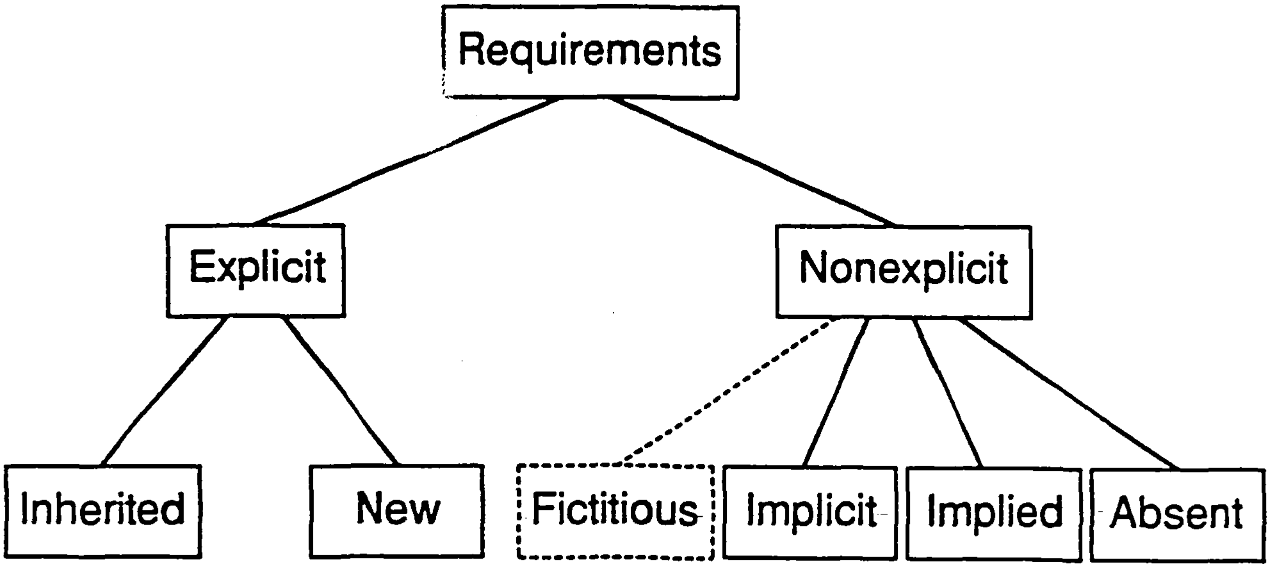
# Classification of requirements
## Explicitness

Requirements of a product are classified as

- Explicit: written in the specification
  - Inherited: comes from previous stages
  - New: created at this stage

- Nonexplicit: not written
  - Fictitious: not a requirement, but a misconception
  - Implicit: belongs to all products
  - Implied: consequence of other requirements
  - Absent: unintentionally omitted

# Classification of requirements
## Explicitness

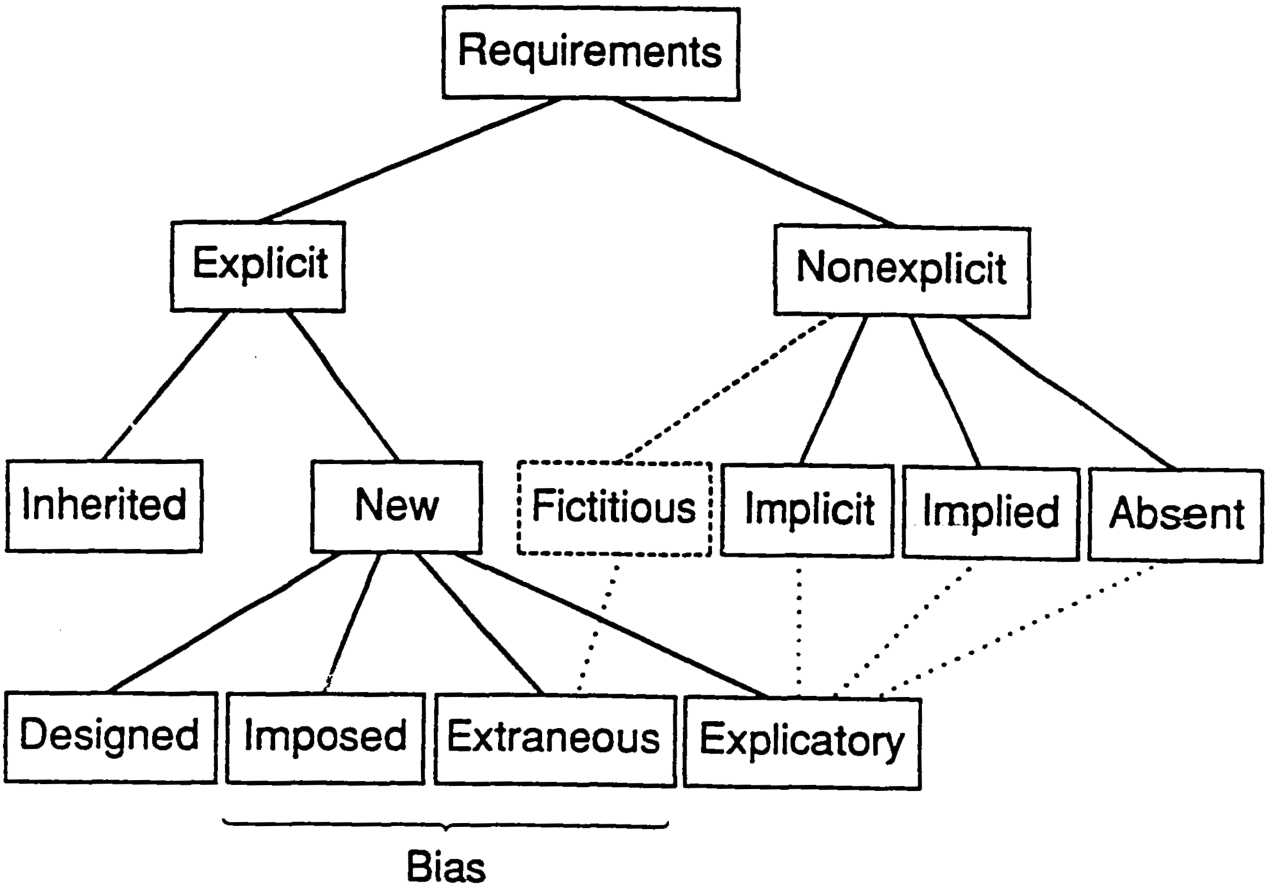


# Classification of requirements
## Origin of new requirements

New requirements are classified as

- Designed: restriction on purpose
- Imposed: restriction of method or language
- Extraneous: makes explicit a fictitious requirement
- Explicatory: makes explicit a nonexplicit requirement

## Classification of requirements
## Creation of new requirements



Requirements

Explicit

Nonexplicit

Inherited

New

Fictitious

Implicit

Implied

Absent

Designed

Imposed

Extraneous

Explicatory

Bias

# The nature of bias
# Definition

## Definition

A specification containing extraneous or imposed requirements is *biased*.

## Origin of bias

- wrongful interpretation of nonexplicit requirements
- limitation imposed by method or language

## Consequences of bias

- solutions not specified
- adoption of a nonoptimal solution

# The nature of bias
## Essential limitations

Bias is not an absolute property.

Bias depends on origin.

Bias depends on application domain and environment, because of different implicit requirements.

Bias cannot be completely eliminated.

## The nature of bias
## Example (at NASA/GSFC)

Before introduction of Ada, FORTRAN was implicit.

Specifications had many FORTRAN-oriented requirements.

During first Ada project, the specifications had to be rewritten.

After introducing Ada, assuming FORTRAN was a fictitious requirement.

# Conclusions
## Other Considerations

- Formal definition of 'requirement'.
- Method to find bias.
- Formalism to write specifications with attributes (e.g., origin of requirements).

# Conclusions
# Contributions

A theory of bias

- classification of requirements
- origin of requirements
- precise definition of bias
- bias is inherent to specifications

The Extensible Description Formalism (EDF) a language to

- describe requirements and their attributes
- compare specifications
- measure bias

Bias in relationship with software defects:

- errors ↔ fictitious requirements
- faults ↔ bias

# Conclusions
## Next Steps

Try these ideas measuring bias in a specific project.

Extend this theory to explain creation of software defects.

## SESSION 4—REUSE

P. A. Straub, University of Maryland

R. W. Kester, CSC

D. J. Reifer, RCI

6269-0

N92
19431
UNCLAS

# SEL ADA REUSE ANALYSIS AND REPRESENTATIONS

Rush Kester

Computer Sciences Corporation
GreenTec II
10110 Aerospace Road
Lanham-Seabrook, MD 20706
(301) 794–1714

## INTRODUCTION

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC). It was created in 1977 to investigate the effectiveness of software engineering technologies applied to the development of applications software. The SEL has three primary organizational members: NASA/GSFC, Systems Development Branch; University of Maryland, Computer Science Department; and Computer Sciences Corporation, Flight Dynamics Technology Group.

Applications developed in the SEL environment are used primarily to determine and predict the orbit and orientation of earth orbiting satellites. There are many similarities among systems developed for different satellites, and those similarities create a climate in which software reuse enhances efficiency and cost effectiveness in the development process. Consequently, reuse has always been an important SEL priority. Over the last several years, with the introduction of Ada and object-oriented design (OOD) techniques, the SEL has been able to achieve a significant increase in the amount of software reuse. Figure 1 represents graphically the increase in software reuse on recent Ada projects as compared with reuse on FORTRAN projects in a similar time period (Reference 1).

Incorporated into all SEL development is the Process Improvement Paradigm (Reference 2), which includes the following four steps:

1. Understand and characterize the current environment.
2. Try a candidate improvement.
3. Measure any change and provide feedback on experiences.
4. Adopt candidate improvements with favorable results; reject those with unfavorable results.

This Ada reuse study has as a primary objective the first process improvement step.

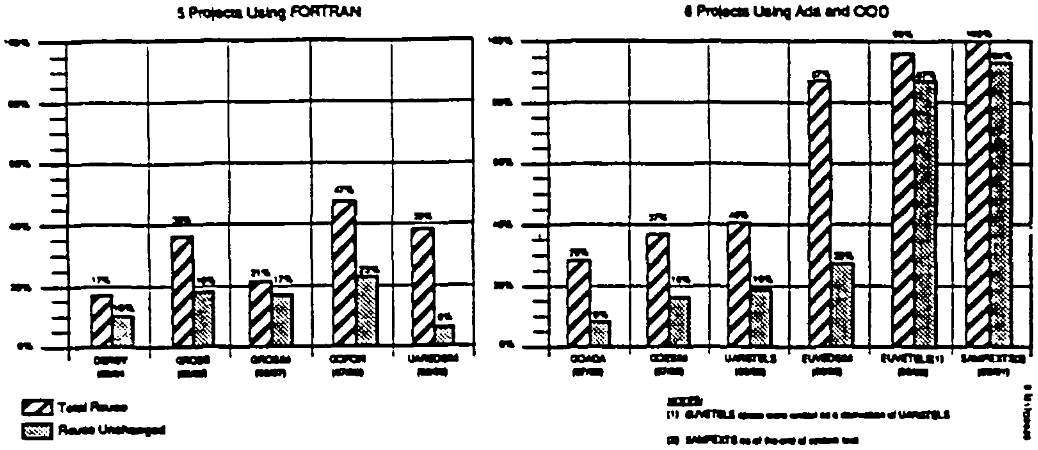


Figure 1. Reuse of Simulator Components

This paper describes the analysis performed and some preliminary findings of a study of software reuse on Ada projects at the SEL.

It describes the representations used to make reuse patterns and trends apparent. The paper focuses on those aspects of the analysis that are applicable to other environments and demonstrates graphically some of the specific patterns of reuse studied by the SEL.

## THE STUDY

The study examined software components (i.e., source code files) of three general types: those that were developed on an earlier project within the SEL environment; those that were acquired from an external source, such as a public domain repository or commercial vendor; and those that were adapted from a similar component on the same project. The reuse information in the SEL data base (Reference 3), combined with the source code files and design documents for each executable program, enabled the study team to trace the evolution of software components over a 5-year period that included several generations of similar applications.

As part of the SEL's standard data collection process (Reference 4), the projects involved in the study had recorded each component of the programs comprising each system. Each component was classified according to the percentages of new and reused code it contained.

For those components considered reused, the parent project (or library) was identified and recorded.

## Reuse Defined by SEL Study

This SEL Ada study focuses on the reuse of source code files obtained from existing projects or libraries. Although projects in the SEL apply other forms of reuse, such as specification or design products, those other forms were not the subject of this study.

During development, project developers classified (Reference 4) the origin of each component according to the amount of new versus reused code it contained. The highest degree of reuse was for components that were reused verbatim (i.e., unchanged). The next degree was for components that were slightly modified (25 percent or less of their source code changed). Finally, the lowest degree was for components that were extensively modified (more than 25 percent changed).

The following types of components were excluded from the definition of reuse for this study:

- All components developed from scratch, including any such component that may have contained fragments of code from one or more source files or design concepts borrowed from existing components.
- Common components developed for a given project, whether used unchanged multiple times in a given executable program or in multiple programs within the project.

## Ada Projects Studied

The study included nine Ada projects. Three projects are dynamics simulators that model the spacecraft's orbit and attitude to evaluate attitude control algorithms. Three projects are telemetry simulators used to generate test data for attitude determination software. One project is an embedded orbit determination system. Another project studied developed or collected components for a reuse library. Finally, one project developed a tool for assembling systems from reusable components. Figure 2 shows a timeline of the nine Ada projects studied.

## Representations of Reuse

To identify patterns of reuse over time or within the software's architecture, the SEL study created a series of graphical, textual, and combination graphical/textual representations of component origination and reuse information.

### Reuse Across Projects Over Time

One group of four reports graphically represents software reuse by multiple projects over time: PROJECT REUSE SUMMARY, PROJECT REUSE NETWORK, REUSE FROM LINEAGE, and REUSE BY LINEAGE.

The PROJECT REUSE SUMMARY report shows, at the project level, the number of components reused from each project or library. The report is constructed as a matrix

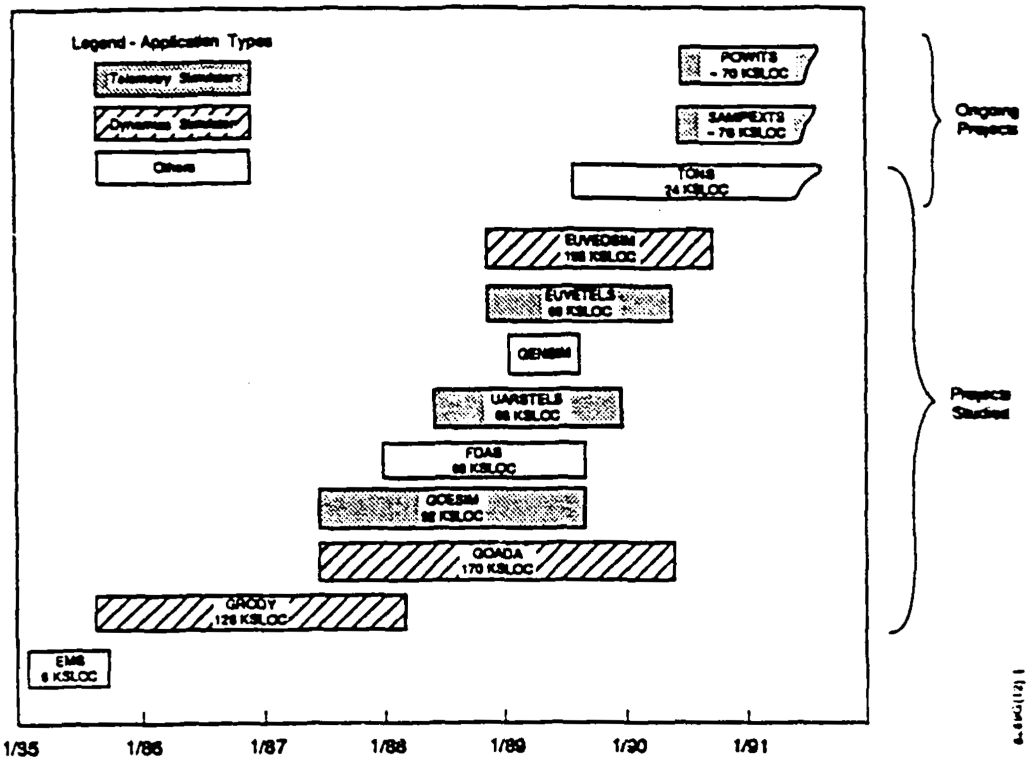


Figure 2. Ada Projects in the Flight Dynamics Division

with projects producing reusable components along the horizontal axis and projects consuming reusable components along the vertical axis. Each cell gives the number of components obtained from a producing project and reused by a consuming project. Two versions of this report could be produced: a detailed report (not shown), which gives subtotals for each of the three degrees of reuse defined for the study, and a total reuse report (shown in Table 1), which gives only the total for all three degrees of reuse.

The PROJECT REUSE NETWORK report (Figure 3) illustrates reuse as a directed graph. The nodes in the graph represent each project, while the fill pattern indicates the type of application. The thickness of each arrow indicates the rough order of magnitude for the number of components reused, and the direction of the arrow indicates the producing and consuming projects. In this report, an increase was evident in the amount of reuse for successive generations of both telemetry simulator and dynamics simulator applications.

A REUSE FROM LINEAGE report focuses on the origin of each reused component. For each instance of a component's use, the project name, the subsystem name, the

Table 1. Number of Reused Components by Project

| Reusing Project | Parent Project | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Beech | GRODY | GOADA | GOESIM | GENSIM | UARSTELS | Other Ada | Other Non-Ada | Total Reused | Total Components |
| GRODY | | 5 | | | | | 2 | 11 | 18 | 551 |
| GOADA | 2 | 187 | 17 | 1 | | | 6 | 6 | 219 | 785 |
| GOESIM | 16 | 4 | 166 | 4 | | 13 | | | 203 | 541 |
| FDAS | 4 | 1 | | | | | | | 5 | 870 |
| GENSIM | 8 | 2 | 10 | | | | | 1 | 21 | 79 |
| UARSTELS | 10 | 45 | 97 | 8 | | 4 | | 6 | 170 | 425 |
| EUVETELS | 2 | | 1 | | | 404 | 2 | | 409 | 427 |
| EUVEDSIM | | | 533 | | | 34 | 173 | | 740 | 861 |
| TONSVAX | 2 | | | | 4 | | 2 | 127 | 135 | 483 |
| TOTALS | | | | | | | | | 1920 | 4860 |

component name, and degree of change required are given. The lineage history of each component is shown by indentation, with each level of indentation indicating a prior reuse generation. The following example shows the lineage of the project EUVEDSIM, subsystem SHEM's component EARTH_ATMOSPHERE.

```
EUVEDSIM  SHEM  EARTH_ATMOSPHERE         Reused (Unchanged) from
     GOADA  SHEM  EARTH_ATMOSPHERE       Reused (Extensively modified) from
          GRODY  TM  ATMOSB              New
```

Note that the names of the original component, ATMOSB, and subsystem, TM, of GRODY were changed by GOADA.

A REUSE BY LINEAGE report was used to show the project(s) that have reused each component. For each instance of a component's use, the project name, the subsystem name, the component name, and degree of change are given. The family tree of each component is shown by indentation, with each level of indentation indicating a subsequent generation of reuse. Using this report and focusing on the degree of change

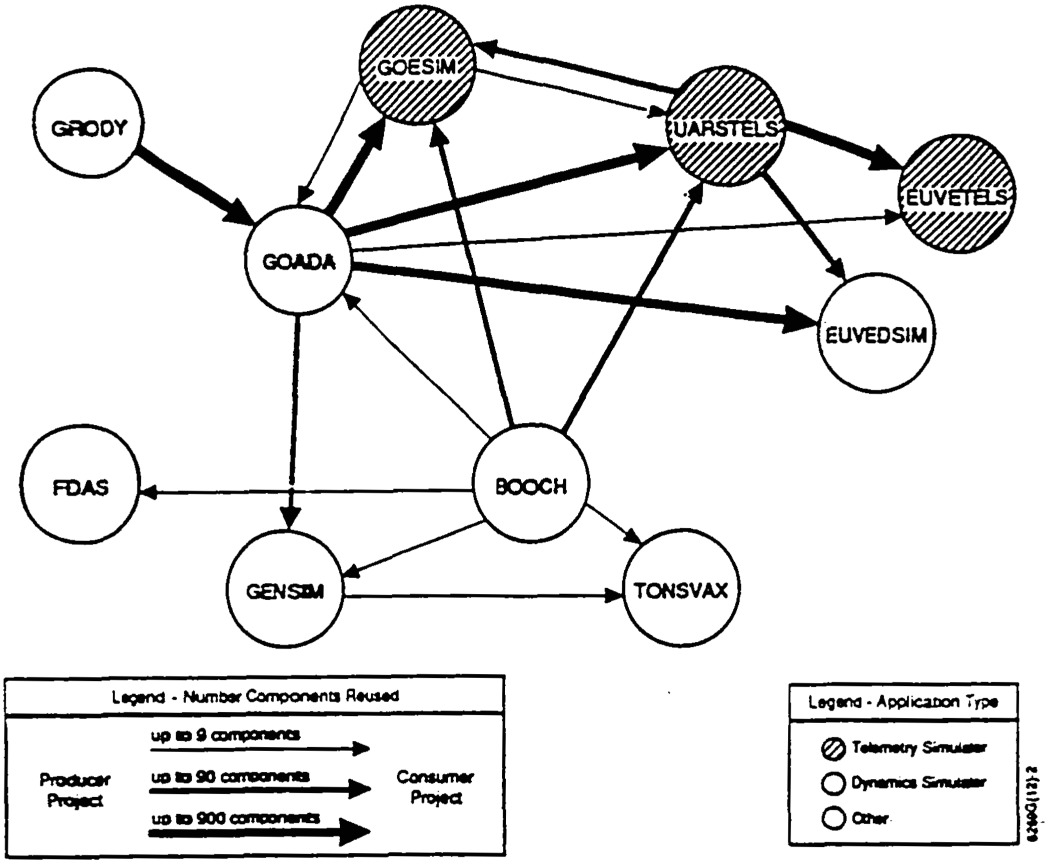


**Figure 3. Project Level Reuse Network**

required, the study staff noticed two common patterns. The first pattern, a sample of which follows, is for components that implement a general solution.

```
    ...component_name   (NEW) Reused by
       ...                  (UNCHANGED)
       ...                  (UNCHANGED)
       ...                  (UNCHANGED)
```

The second pattern, shown in the following example, was seen in components that have incompletely or incorrectly implemented a general solution.

```
    ...component_name   (NEW) Reused by
       ...                  (...MODIFIED)
       ...                  (...MODIFIED)
       ...                  (...MODIFIED)
```

Another component attribute that can be determined by using the REUSE BY LINEAGE report is the component's domain of reuse. Assuming, as is the case in this study, that the researcher knows the application type (or domain) of each project, the domain of reuse can be derived by examining the type of projects reusing a given component.

Both the REUSE FROM LINEAGE and the REUSE BY LINEAGE reports can also be useful for configuration management purposes, by identifying the projects using a given component. If one of the projects suggests an enhancement or correction, the other reusing projects could also be notified.

## Reuse Within a Project's Architecture

In addition to examining reuse over time, this study also examined reuse within each project's architecture. To illustrate the findings on a system-wide scale, component reuse was superimposed on graphical/textual representations of each system's static structure, its calling hierarchy, and its compilation order. Architectural representations were derived from the source code and/or design documents.

Each representation of a project's architecture requires between 3 and 10 pages of graphical/textual hard copy, which made it difficult to observe the overall reuse patterns. To overcome this difficulty, the degree of change was color-coded on all representations of reuse within a project's architecture. These representations could then be posted on a wall and the color-coded reuse patterns observed from a distance.

### Reuse on a Project's Static Structure

Components were organized according to the Ada package to which they belong. These packages were further organized according to the logical subsystems defined by the original developers. The degree of component reuse was then overlaid on the subsystem and the resulting representation analyzed for patterns. The following is a sample from the REUSE ON STATIC STRUCTURE report for the GOADA project. This sample shows the degree of reuse in one package, EPHEM_FILE_MANAGER, which is part of the SHEM subsystem.

| Ada Name | | Component Type | Degree of Reuse |
|---|---|---|---|
| EPHEM_FILE_MANAGER | . | Package Spec | Unchanged |
| | . | Package Body | Unchanged |
| | .CLOSE | Procedure Body | Unchanged |
| | .COORDINATE_SYSTEM_OF | Function Body | New |
| | .DATA_INTERVAL_OF | Function Body | Unchanged |
| | .END_TIME_OF | Function Body | Unchanged |
| | .INITIALIZE | Function Body | Unchanged |
| | .READ_EPHEM_POINT | Procedure Body | Unchanged |
| | .REAL_TO_AC_TIME | Function Body | Unchanged |
| | .START_TIME_OF | Function Body | Unchanged |

Examination of this representation confirmed our intuition that in most cases the granularity of reuse was library units (such as packages or standalone procedures or functions). The granularity of reuse for telemetry simulators changed dramatically to the entire system architecture in the EUVETELS project, in which structure and almost all components from the UARSTELS project were reused unchanged.

The study examined the hypothesis that a component's interface (i.e., its Ada specification) is reusable with fewer changes than its implementation (i.e., its body). In general, the hypothesis was confirmed by the preceding representation. The notable exception was the second generation of dynamics simulators (i.e., the GOADA project), in which previously large packages were divided into smaller packages. In that case, new package specifications were created for unchanged or slightly modified package bodies.

Another hypothesis examined was that for some groups of components (i.e., a package or a subsystem), there are some parts that must be consistently tailored to each use. Although parts of some packages or subsystems were modified in each successive generation, it was not possible to distinguish mission-tailored parts from those modified for other reasons.

Another hypothesis examined was that some parts of an application's architecture lend themselves to reuse more than do others. The study confirmed the hypothesis by revealing that the highest concentration of consistently reused components was among those implementing basic data structures and mathematical functions or operating on standardized data files.

### Reuse on Project's Call Tree

The static analysis of subprogram calls in each component, starting with the main subprogram, is used to create a call tree. The call tree is represented textually as an indented outline, in which each level of indentation denotes a level of nested calls. The order of subprograms in the call tree reflects the order in which each subprogram call appears in the text and, for sequential code, reflects the execution order of the calls. The call analysis demonstrates whether reuse occurs predominantly at the branch or leaf-node level of the call tree.

### Reuse on Project's Compilation Order

To study reuse on compilation order, the study group generated a textual representation combined with color-coded graphical elements. Each project's library units were ordered in a roughly bottom-up fashion according to Ada WITH dependency (i.e., the units with fewer WITH dependencies were listed first). Component reuse was then overlaid on the library units. Examination of this representation revealed that the ability in Ada to separate specification from implementation was effective in isolating higher level reused components from extensive changes in lower level components. Also evident using this representation was the ability to reuse Ada generics without change, even in cases where their functionality included entirely new capabilities implemented by generic subprogram parameters. A sample from a REUSE ON PROJECT'S COMPILATION ORDER report is shown in Figure 4.

```
GENERIC_HARDWARE              (Spec = OLDUC, Body = OLDUC, Subunits OLDUC = 8)
WHEEL                         (Spec = OLDUC)
TORQUER                       (Spec = OLDUC)
GYRO_ANALOG_RATES             (Spec = OLDUC)
TONS_DOPPLER                  (Spec =  NEW )
GENERIC_MODEL_GYRO_ANGLES     (Spec = OLDUC, Body = OLDUC)
GYRO_ANGLES                   (Spec = OLDUC)
SCHEDULER                     (Spec = OLDUC, Body = SLMOD, Subunits SLMOD = 2, OLDUC = 4)
SIMULATOR_DATABASE            (Spec = SLMOD, Body = SLMOD, Subunits SLMOD = 2)
INITIAL_PARAMETERS_REPORT     (Spec = OLDUC, Body = OLDUC, Subunits SLMOD = 1)
MINOR_FRAME                   (Spec = SLMOD)
TELEMETRY_RECORD              (Spec = OLDUC)
TELEMETRY_SIMULATOR           (              Body = OLDUC)

                              Legend: OLDUC is unchanged
                                      SLMOD is slightly modified
                                      NEW is developed from scratch
```

Figure 4. Reuse on Compilation Order

## CONCLUSIONS

Overall, the study revealed that the pattern of reuse has evolved from initial reuse of utility components into reuse of generalized application architectures. Utility components were both domain-independent utilities, such as queues and stacks, and domain-specific utilities, such as those that implement spacecraft orbit and attitude mathematical functions and physics or astronomical models. The level of reuse was significantly increased with the development of a generalized telemetry simulator architecture.

The use of Ada generics significantly increased the level of verbatim reuse, which is due to the ability, using Ada generics, to parameterize the aspects of design that are configurable during reuse. A key factor in implementing generalized architectures was the ability to use generic subprogram parameters to tailor parts of the algorithm embedded within the architecture.

The use of object-oriented design (in which objects model real-world entities) significantly improved the modularity for reuse. Encapsulating into packages the data and operations associated with common real-world entities creates natural building blocks for reuse.

## REFERENCES

1. Software Engineering Laboratory, SEL-89-007, "Experiences in the SEL—Applying Software Measurement," *Proceedings of the Fourteenth Annual Software Engineering Workshop*, F. E. McGarry (GSFC), S. R. Waligora (CSC), and T. P. McDermott (CSC), November 1989

2. Software Engineering Laboratory, SEL-89-007, "The Experience Factory: Packaging Software Experience," *Proceedings of the Fourteenth Annual Software Engineering Workshop*, V. R. Basili (University of Maryland), November 1989

3. Software Engineering Laboratory, SEL-89-101, “*Software Engineering Laboratory (SEL) Database Organization and User's Guide (Revision 1)*, M. So, G. Heller, S. Steinberg, K. Pumphrey, and D. Speigel, February 1980

4. Software Engineering Laboratory, SEL-87-008, *Data Collection Procedures for the Rehosted SEL Database*, G. Heller, October 1987

# VIEWGRAPH MATERIALS

# FOR THE

# R. KESTER PRESENTATION

# Ada Reuse Analysis and Representation at the Software Engineering Laboratory (SEL)

Rush Kester
Rhea White
Robert Kazden

# Agenda

- Background
- Representations of reuse
- Preliminary observations

Computer Sciences Corporation
System Sciences Division

# Definition of Reuse

- Ada source code obtained from existing projects or libraries
- Each source file (a.k.a. component) classified according to percent of lines reused without change
- Definition does not include other forms of reuse

# Potential Benefits of Reuse

***With high-level reuse, delivered systems can be***

- **Delivered sooner and at lower cost**
- **Incrementally improved**
- **More rellable**

CSC Computer Sciences Corporation
System Sciences Division

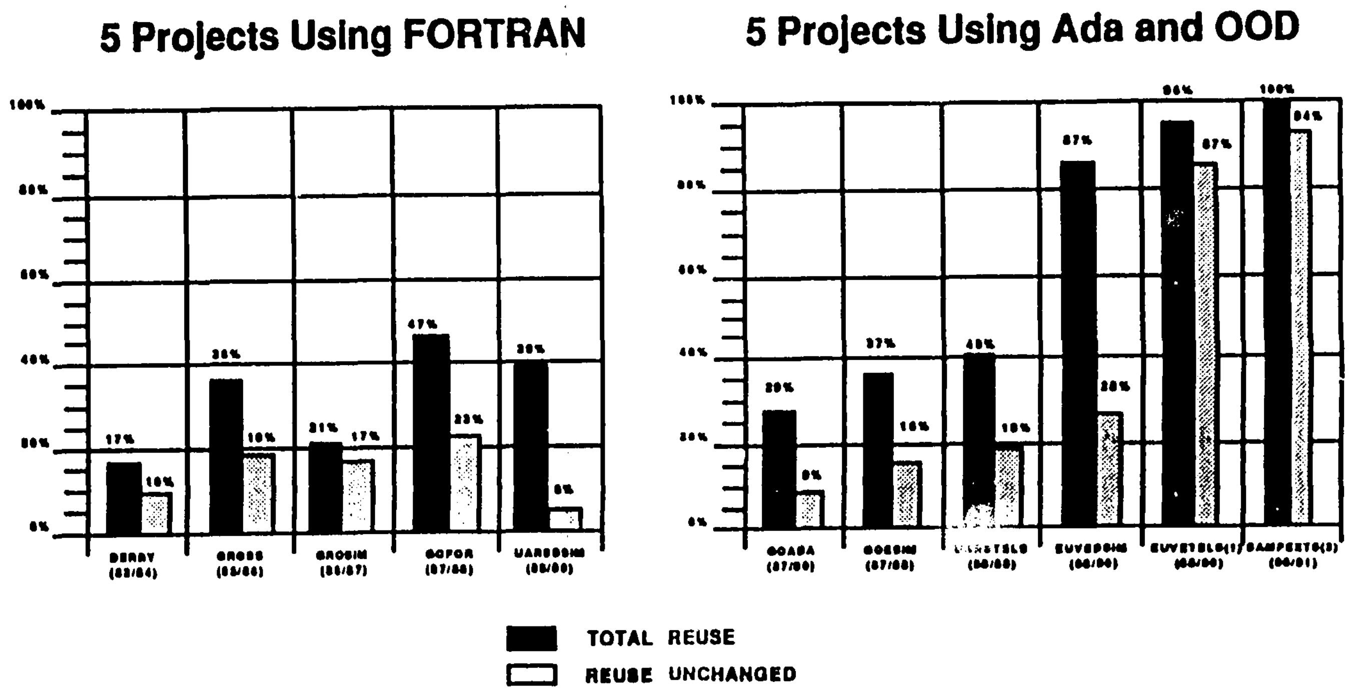
# Flight Dynamics Environment

- Develop similar systems for different satellites
- Knowledge carried between missions
- Reuse an important part of culture
- Economic benefits directly related to amount of code reused without change
- Introduction of Ada and OOD significantly increased reuse of code

Computer Sciences Corporation
System Sciences Division

6151G(12)

# Reuse of Simulator Components



Computer Sciences Corporation
System Sciences Division

8181Q(12)

R. Kester
CSC
Page 16 of 38

# Steps in Process Improvement

1. Understand and characterize current environment
2. Try candidate improvement
3. Measure change – feedback experience
4. Adopt candidates with favorable results; Reject candidates with unfavorable results

CSC Computer Sciences Corporation
System Sciences Division

# Goals of Current Phase of Study

- Understand and characterize reuse
  - Determine patterns and trends of reuse
  - Determine characteristics distinguishing reused from non-reused components
- Identify candidates for reuse library
- Identify domain of component's reusability within the environment
- Address some CM issues related to reuse

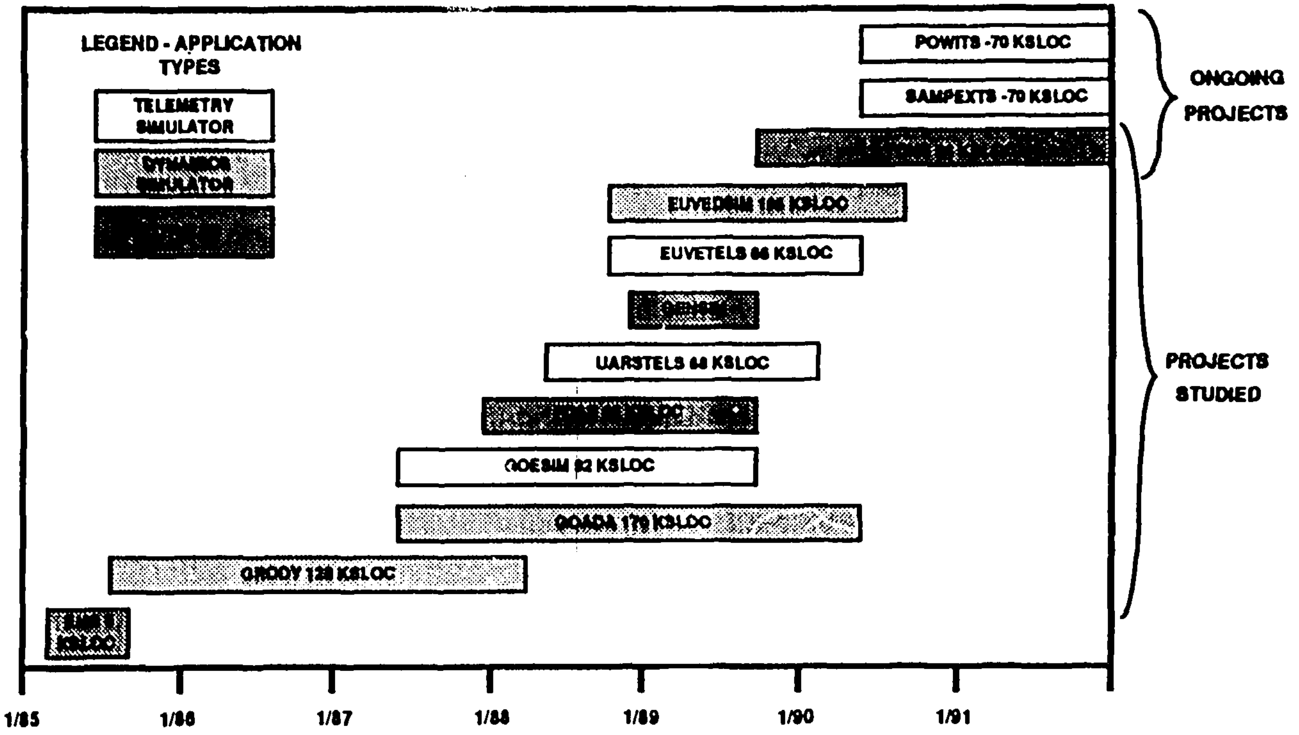
Computer Sciences Corporation
System Sciences Division

# Questions Addressed by Study

- Does separation of interface specification and implementation affect degree of change required?
- Do Ada generics improve the level of reuse without change?
- Does the extent of intercomponent dependencies affect reuse?
- What is the granularity of reuse?
- Where in software architecture does reuse occur?
- Do patterns of component evolution suggest guidelines for more effective reuse?

CSC Computer Sciences Corporation
System Sciences Division

# Ada Projects in the Flight Dynamics Division



R. Kester
CSC
Page 29 of 30

Computer Sciences Corporation
System Sciences Division

6151G(12)

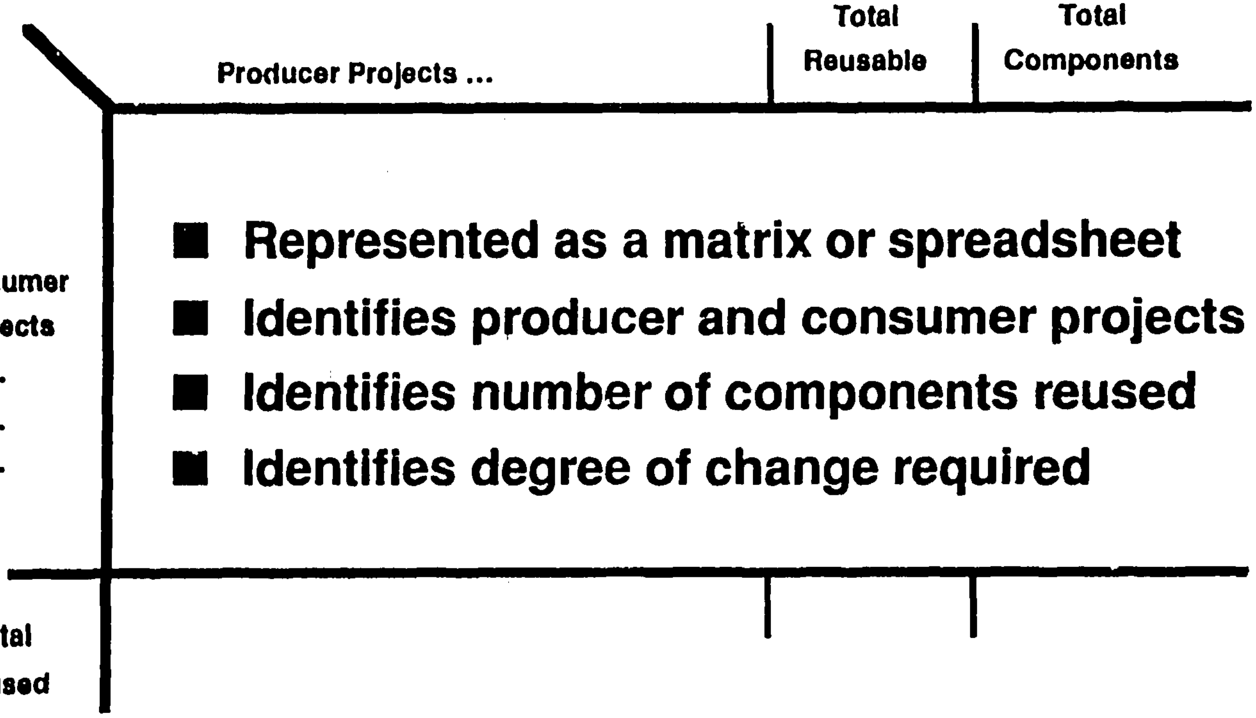
# Data Used

- Size of Study
  - 9 Ada projects, over 5 years
  - Over 1900 reused components
- Input Used
  - SEL Component Origination Forms
  - Source code files
  - Representations of software design

# Representations of Reuse

1. Multi-Project Reuse Summary
2. Project Level Reuse Network
3. Component Lineage Reports
4. Reuse on Software Static Structure
5. Reuse on Software Call Tree
6. Reuse on Ada Compilation Order

# 1. Multi-Project Reuse Summary Report



Producer Projects ...

Total Reusable

Total Components

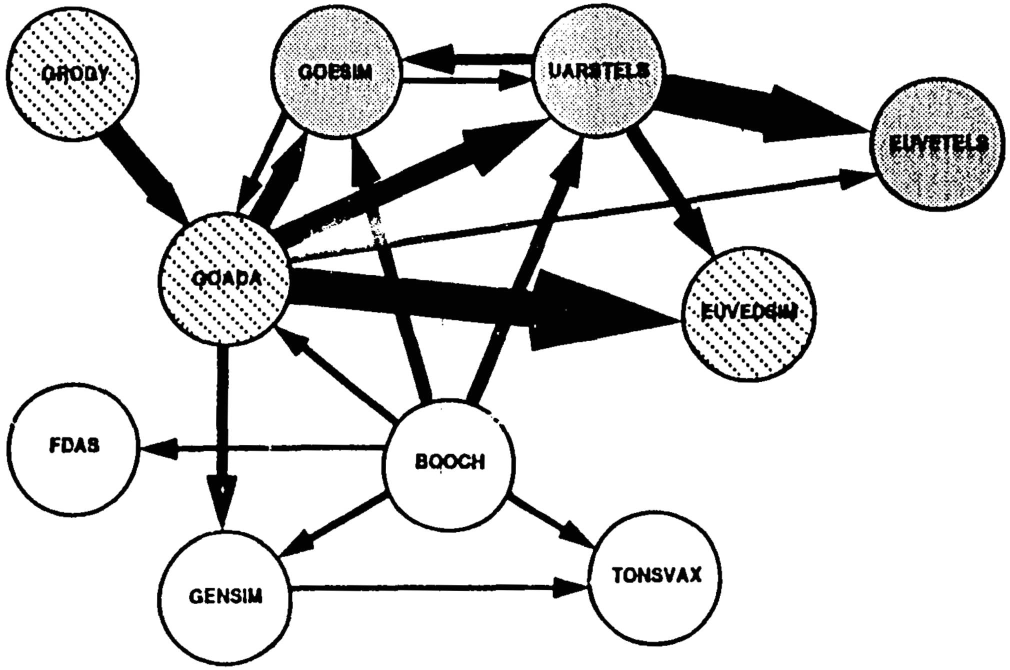
Consumer Projects
.
.
.

- Represented as a matrix or spreadsheet
- Identifies producer and consumer projects
- Identifies number of components reused
- Identifies degree of change required

Total Reused

R. Kester
CSC
Page 23 of 30

# 2. Project Level Reuse Network



Computer Sciences Corporation
System Sciences Division

6151C(12)

# 3. Component Lineage Reports

- Represents reuse over time
  - From originating project forward
  - From reusing project back to origin
- Identifies parent - child relationship
- Identifies components:
  - Implementing general solution
  - Generalized incorrectly or incompletely
  - Domain of applicability
- Useful for CM purposes

Computer Sciences Corporation
System Sciences Division

6151G(12)

# 4. Reuse on Software Static Structure

- Represents reuse at project level
- Reflects developer's logical view
- Makes visible:
  - Granularity of reuse
  - Project dependent parts

CSC Computer Sciences Corporation
System Sciences Division

61510(12)

# 5. Reuse on Software Call Tree

- Represents reuse at project level
- Reflects actual calling hierarchy
- Makes visible:
  - Level of functionality reused
  - Location of reuse within architecture

CSC Computer Sciences Corporation
*System Sciences Division*

# 6. Reuse on Ada Compilation Order

- Represents reuse at project level
- Reflects coupling between "Library Units"
- Makes visible:
  - Scope of change required to reuse
  - Location of reuse within architecture

CSC Computer Sciences Corporation
System Sciences Division

# Reuse Patterns and Trends Observed

- Initially application independent components reused, now majority reflect organization's problem domain
- Ada generics significantly increased the level of verbatim reuse
- OOD (where objects model real world entities) significantly improved modularity

CSC Computer Sciences Corporation
System Sciences Division

6151G(12)

# Work Remaining

- Develop guidance for improving verbatim reuse
- Investigate rationale behind characteristics that distinguish reusable components
- Confirm hypothesis -- Achieved highly reusable solution for Telemetry Simulator applications

CSC Computer Sciences Corporation
System Sciences Division

N92
19432
UNCLAS

P-20

R0235972

# REUSE METRICS AND MEASUREMENT - A FRAMEWORK

Donald J. Reifer, President
Reifer Consultants, Inc.
Torrance, CA 90505

Abstract: This presentation will describe the lessons learned and experience gleaned by those firms which have started to implement the reuse metrics and measurement framework prepared by the Joint Integrated Avionics Working Group (JIAWG) for use in controlling the development of common avionics and software for its affiliated aircraft programs (e.g., the Air Force's Advanced Tactical Fighter (ATF), the Army's LH helicopter and the Navy's A-12 fighter). The framework was developed to permit the JIAWG and Service System Program Offices (SPOs) to measure the long-term cost/benefits resulting from the creation and use of Reusable Software Objects (RSOs). The framework also monitors the efficiency and effectiveness of the JIAWG's Software Reuse Library (SRL).

The presentation will begin by defining the metrics and measurement framework which was established to allow the following six determinations and findings to be made relative to software reuse:

1. Impact of RSO creation on software cost and productivity.
2. Impact of RSO reuse on software cost and productivity.
3. Impact of RSO mining on software cost and productivity.
4. Minimum standards of quality for RSOs as they enter the SRL.
5. Efficiency and effectiveness of SRL usage.
6. Long-term cost/benefits of SRL usage.

The presentation will discuss how the following seven criteria were used to guide the establishment of the proposed reuse framework:

1. Compatible - The framework should be compatible with the software processes used by JIAWG contractors to develop avionics software products in Ada under DoD-STD-2167A.
2. Ease of Data Collection - The data needed to quantify the metric should be easy to collect and normalize.
3. Ease of Understanding - The metrics employed should be easy to understand, analyze and interpret.
4. Minimum Cost - The measurement costs (i.e., data collection, analysis and reporting) should be kept to a minimum.
5. Nonobtrusive - Collection of metrics data must not adversely impact the processes or products being measured.
6. Objective - It should be difficult to bias or distort the value of the metric.
7. Predictive - The metric should facilitate generation of accurate estimates of software cost, productivity and quality.

Next, object recapture and creation metrics will be explained along with their normalized use in effort, productivity and quality determination. A single and multiple reuse instance version of the popular COCOMO cost model will be presented which employs these metrics and the measurement scheme proposed by the Software Productivity Consortium (SPC) to predict the software effort and duration under various reuse assumptions. Investigations in using this model to predict actuals taken from the RCI database of over one thousand completed projects will be discussed along with statistical findings.

User experience with this metrics and measurement framework as part of the Air Force's Reusable Ada Avionics Software Package (RAASP) and Avionics Fault-Tolerant Software/Ada Technology Insertion Program (AFTS/ATIP) projects will be discussed next. The lessons learned with these metrics by these projects will be summarized. These two projects are conducting controlled experiments to capture measurement data that provides insight into those factors which impact software cost, quality, productivity and system performance. The RAASP effort is focusing on determining the relative impact of object-oriented methods, reuse paradigms and SRL operational policies software productivity, cost and quality. AFTS/ATIP is assessing the impact of a large number of process and product factors on overall cost and system performance.

The presentation will conclude with a summary of key points. Recommendations will be presented to help those embarking on a reuse program to improve their measurement and prediction capabilities.

# VIEWGRAPH MATERIALS

# FOR THE

# D. REIFER PRESENTATION

RCI-TN-470

# REUSE METRICS AND MEASUREMENT - A FRAMEWORK

28 November 1990

Prepared For:

NASA/Goddard Fifteenth Annual
Software Engineering Workshop

Reifer Consultants, Inc.

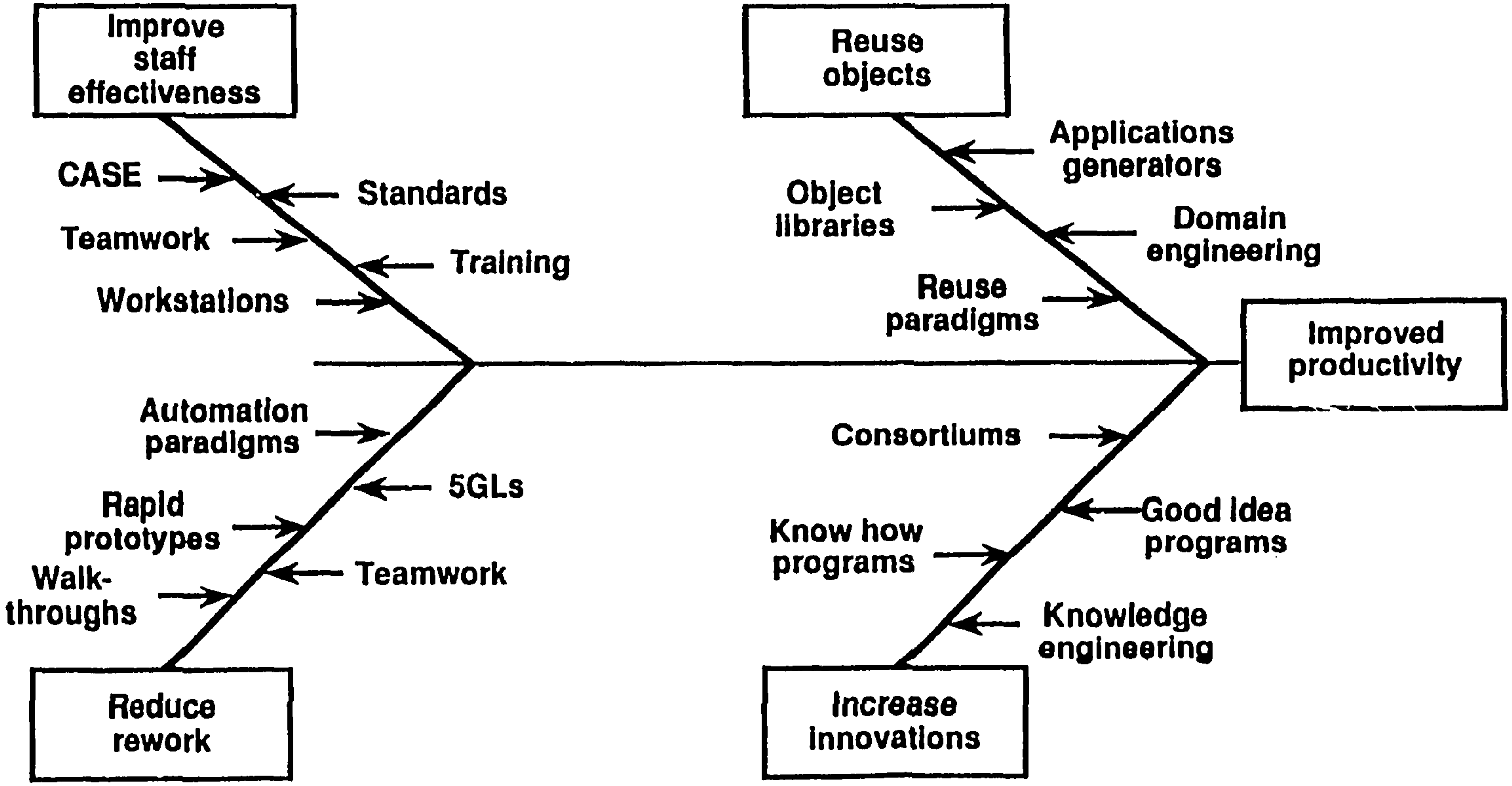
25550 Hawthorne Boulevard, Suite 112, Torrance, California 90505 / Phone: (213) 373-8728 / Fax: (213) 375-9845

# PURPOSE

- Describe the reuse metrics and measurement framework created by JIAWG to make the following determinations:
  - Impact of RSO acquisition on software cost and productivity
  - Impact of RSO reuse on software cost and productivity
  - Minimum standards of quality for RSOs entering the Software Reuse Library (SRL)
  - Efficiency and effectiveness of SRL usage
  - Long-term cost/benefits of SRL usage
- Discuss implementation of the framework on the OSS and RAASP projects

Reusable Software Object (RSO) - life cycle products developed to be reused (designs, algorithms, code, tests/test cases, etc.)

# PRODUCTIVITY IMPROVEMENT STRATEGIES



Improve staff effectiveness

CASE

Standards

Teamwork

Training

Workstations

Reuse objects

Applications generators

Object libraries

Domain engineering

Reuse paradigms

Improved productivity

Automation paradigms

5GLs

Rapid prototypes

Teamwork

Walk-throughs

Reduce rework

Consortiums

Good idea programs

Know how programs

Knowledge engineering

Increase innovations

* Productivity must be measured from a quality viewpoint

# BARRIERS TO REUSE

- Lack of incentives
- Few standards
- Limited tool support
- Champion needed
- Multiple quality levels
- NIH bias
- Needed infrastructure changes
- Few quantitative metrics

<u>Source</u>: RCI Reuse Survey, 8/89

# METRICS SELECTION CRITERIA

- Compatible with DOD processes
- Ease of data collection
- Ease of understanding
- Minimum measurement cost
- Objective and unbiased
- Predictive of the future
- Unobtrusive as possible

# REUSE METRICS

## OBJECT ACQUISITION RATIO

$$OAR = \sum_{i=1}^{n} (w_i)\ (a_i/A_i)$$

where: $a_i$ = no. of RSOs acquired per collection

$A_i$ = no. of objects in that collection

$n$ = no. of collections

$w_i$ = weighting factor for each collection

and $\sum_{i=1}^{n} w_i = 1;\ a_i/A_i \geq 0$

## OBJECT REUSE RATIO

$$ORR = \sum_{i=1}^{n} (w_i)\ (r_i/R_i)$$

where: $r_i$ = no. of reused objects in a collection

$R_i$ = no. of objects in that collection

$n$ = no. of collections

$w_i$ = weighting factor for each collection

and $\sum_{i=1}^{n} w_i = 1;\ r_i/R_i \geq 0$

# REUSE MECHANIZATION

| Collection | $W_i$ |
|---|---|
| • Requirements | 0.20 |
| • Design | 0.30 |
| • Source code | 0.20 |
| • Tests/test cases | 0.30 |

- $OAR_e = (0.2)(a_{1n} + a_{1r})/A_1 + (0.3)(a_{2n} + a_{2p} + a_{2r})/A_2 + (0.2)(a_{3n} + a_{3p} + a_{3r})/A_3 + (0.3)(a_{4n} + a_{4p} + a_{4r})/A_4$

  where: $a_{xn}$ = newly created objects
  $a_{xp}$ = purchased objects
  $a_{xr}$ = recovered objects

- $ORR_e = (0.2)\ r_l/R_l + (0.3)\ r_2/R_2 + (0.2)\ r_3/R_3 + (0.3)\ r_4/R_4$

  where: $r_x/R_x$ = reuse ratio for a collection

## REUSE VERSION COCOMO (SINGLE INSTANCE)

- $\text{Effort}_r = c\ (l + (\text{OAR}_x)(b_{18}) - (\text{ORR}_e)(b_{19}))\ \text{Effort}$

where:

$c$ = adjustment factor for domain

$b_{18}$ = RSO cost factor ($0.10 < b_{18} < 0.36$)

$b_{19}$ = RSO benefits factor ($0.20 < b_{19} < 0.60$)

$\text{OAR}_x$ = expanded form of OAR

$\text{ORR}_e$ = effective form of ORR

$\text{Effort}_r$ = cost in staff-months with reuse

Effort = cost in staff-months (COCOMO)

- $\text{OARx} = (0.2)(a_{ln} + (0.5)\ a_{lr})/A_1 + (0.3)(a_{2n} + (0.2)\ a_{2p} + (0.4)\ a_{2r})/A_2 + (0.2)(a_{3n} + (0.2)\ a_{3p} + (0.5)\ a_{3r})/A_3 + (0.3)(a_{4n} + (0.3)\ a_{4p} + (0.6)\ a_{4r})/A_4$

# FACTOR RATINGS

| | LOW | NOMINAL | HIGH | VERY HIGH | EXTRA HIGH |
|---|---|---|---|---|---|
| Reuse Cost Factor ($b_{18}$) | 0.10 | 0.17 | 0.26 | 0.31 | 0.36 |
| | Limited reuse packaging | Design and code RSO reuse packaging | Full RSO reuse packaging | Domain specific RSO reuse packaging | Extensive reuse packaging (synthesis) |

| | LOW | NOMINAL | HIGH | VERY HIGH | EXTRA HIGH |
|---|---|---|---|---|---|
| Reuse Benefits Factor ($b_{19}$) | 0.20 | 0.25 | 0.34 | 0.48 | 0.60 |
| | Planned reuse | Systematic reuse | Managed reuse | Institutionalized reuse (within and across jobs) | Optimized reuse (domain specific) |

# METRICS USAGE

## SPC MODEL

$$\text{Cost} = (1-R) + R(B + E/N)$$

where: $B$ = relative cost to reuse RSO
$R$ = proportion of reused software
$E$ = cost to develop RSO
$N$ = number of reuses

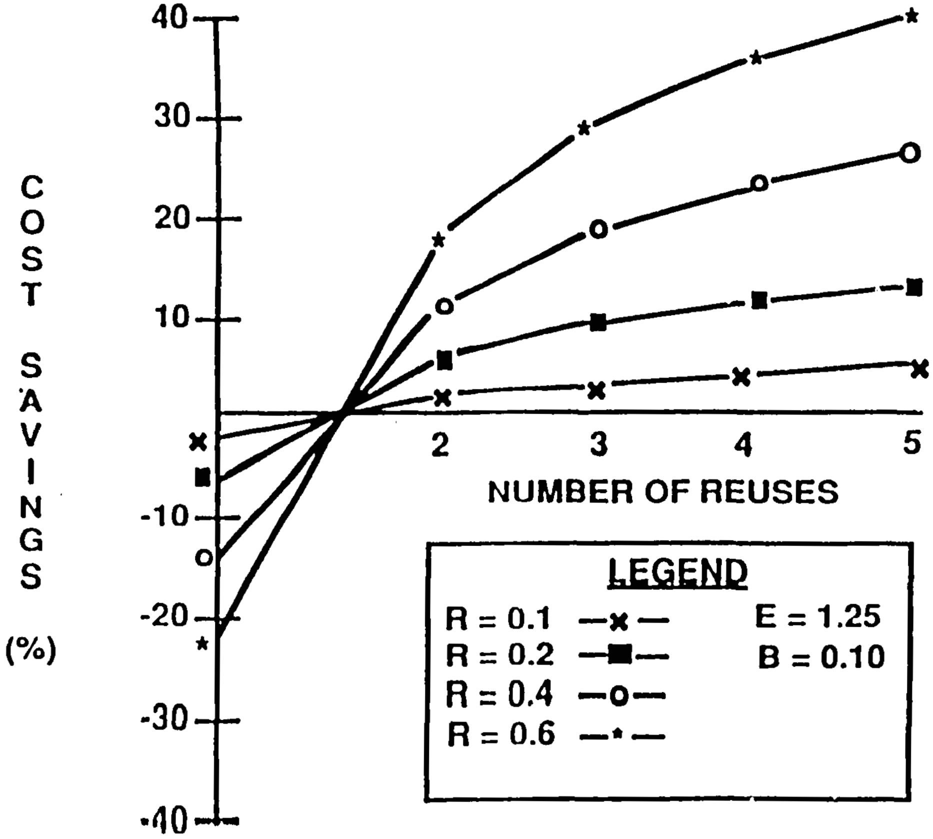
## REUSE COCOMO MODEL (MULTIPLE INSTANCES)

$$\text{Effort}_m = c\left(1 - \frac{(OAR_x)\,(b_{18})}{m} - (ORR_e)\,(b_{19})\right)\text{Effort}$$

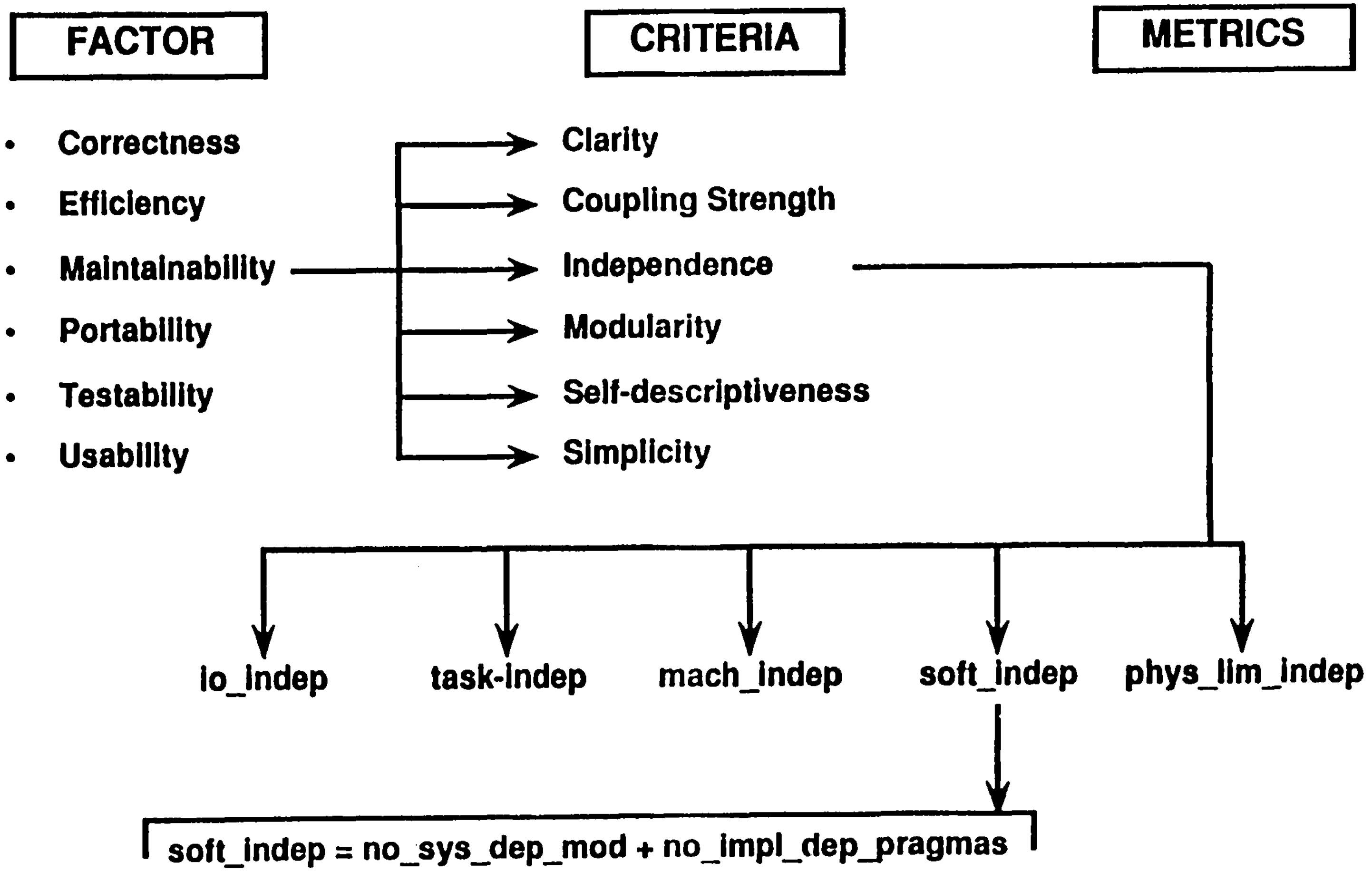
where: $c$ = calibration coefficient
$m$ = number of reuses ($m > 1$)
$b_{18}$ = cost factor ($0.10 < b_{18} < 0.36$)
$b_{19}$ = benefit factor ($0.20 < b_{19} < 0.60$)
$OAR_x$ = Object Acquisition Ratio (average)
$ORR_e$ = Object Reuse Ratio (average)

D. Reifer
RCI
Page 12 of 19

# THE OSS EXAMPLE



# QUALITY METRICS



FACTOR

- Correctness
- Efficiency
- Maintainability
- Portability
- Testability
- Usability

CRITERIA

Clarity

Coupling Strength

Independence

Modularity

Self-descriptiveness

Simplicity

METRICS

io_indep

task-indep

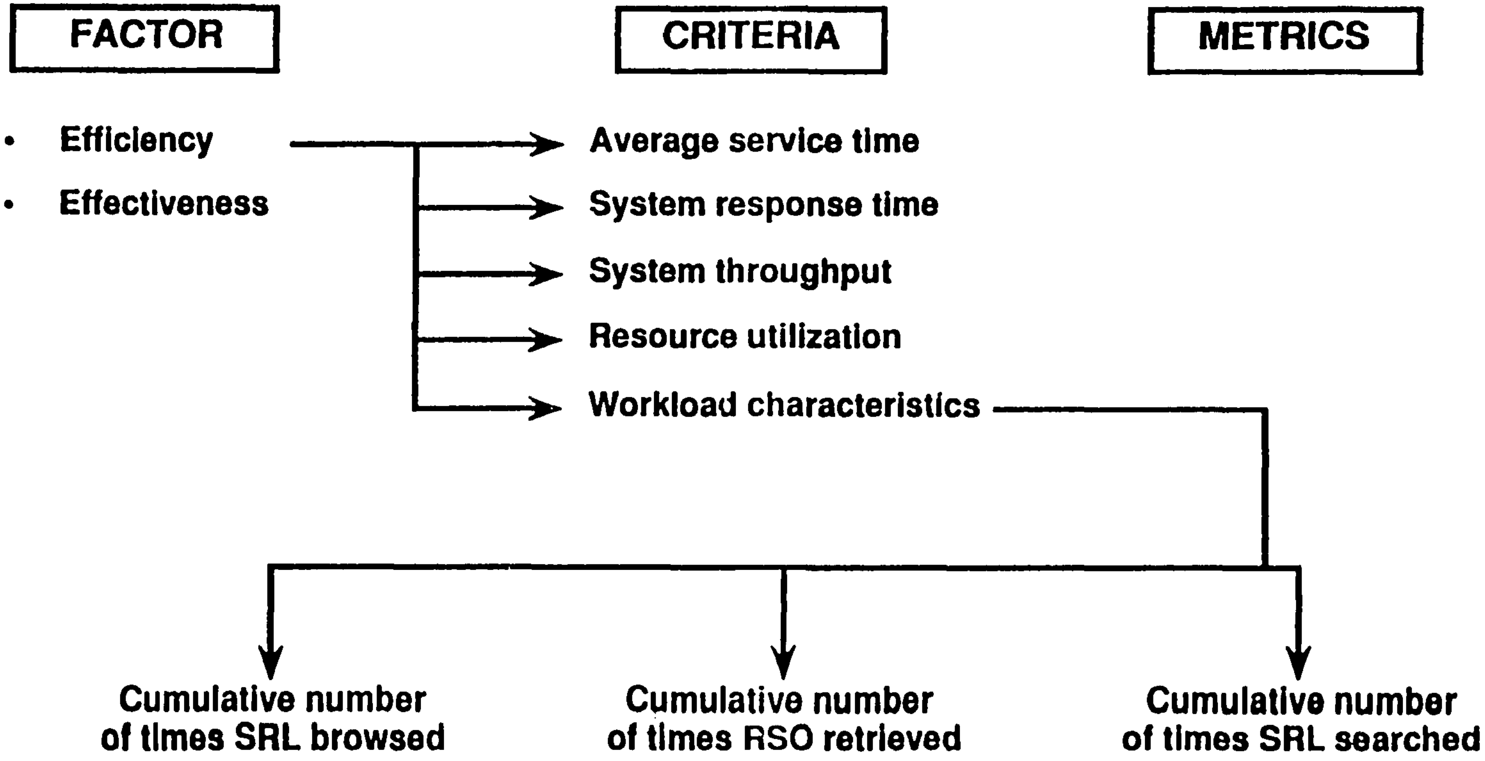
mach_indep

soft_indep

phys_lim_indep

soft_indep = no_sys_dep_mod + no_impl_dep_pragmas

# LIBRARY EFFICIENCY



FACTOR

- Efficiency
- Effectiveness

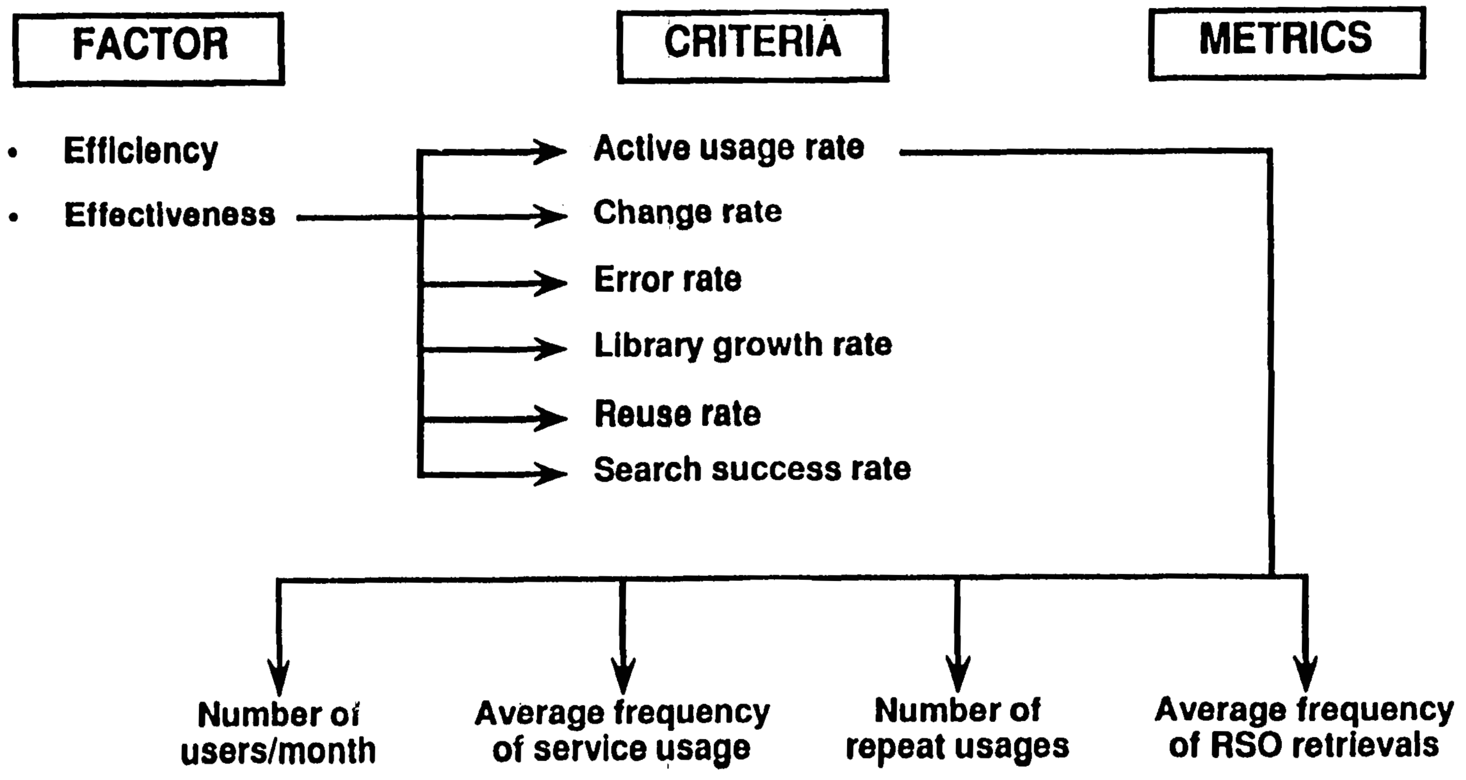
CRITERIA

- Average service time
- System response time
- System throughput
- Resource utilization
- Workload characteristics

METRICS

- Cumulative number of times SRL browsed
- Cumulative number of times RSO retrieved
- Cumulative number of times SRL searched

# LIBRARY EFFECTIVENESS



FACTOR

- Efficiency
- Effectiveness

CRITERIA

Active usage rate

Change rate

Error rate

Library growth rate

Reuse rate

Search success rate

METRICS

Number of users/month

Average frequency of service usage

Number of repeat usages

Average frequency of RSO retrievals

# LONG-TERM COST/BENEFITS

## NET PRESENT WORTH

$$NPW = \sum_{t=0}^{T} (B_t)\ (1/(1+i)^t)$$

### NON-RECURRING COSTS

- Acquisition $ ________
- Adaptation ________
- Documentation ________
- Infrastructure ________
- Training ________

COSTS $

### RECURRING COSTS

- Administration $ ________
- Maintenance ________
- Operations ________

COSTS $

### TANGIBLE BENEFITS

- Cost avoidance $ ________
- Added capability ________
- Reduced cost of quality ________
- Cost savings ________

BENEFITS $

### INTANGIBLE BENEFITS

- Better customer satisfaction $ ________
- Fitness for use ________

BENEFITS $ ________

# RAASP ADAPTATION

## HYPERTEXT LIBRARY EFFICIENCY AND EFFECTIVENESS

- No. of objects In library
- No. of links traversed/hit
- No. of items browsed/hit
- Amount of time for a hit
- No. of log in's per user
- Amount of time/user session
- No. of objects withdrawn/user session
- No. librarian actions/object
- No. of objects submitted/month
- No. of objects withdrawn/month
- No. of SPRs/object/month
- No. of SCRs/object/month

Usage Profiles

- By object
- By service
- System-wide

# SUMMARY AND CONCLUSIONS

- We've described the JIAWG software reuse metrics and measurement framework
- We've described the pilot implementation of the framework on OSS and RAASP
- We've discussed our multiple instance reuse version of the COCOMO model
  - Needed to explore the economics of reuse
- We've just touched the surface of the issues involved
- Your thoughts, feedback and help are solicited especially if you have "hard" data to share

# SESSION 5—PROCESS ASSESSMENT

K. Y. Rone, IBM

A. L. Goel, Syracuse University

J. C. Kelly, NASA/JPL

N92
19433
UNCLAS

[illegible]

G[illegible]2

P- 25

N92-19433

# Cost and Quality Planning for Large NASA Programs

IA639067

Kyle Y. Rone

FEDERAL SECTOR DIVISION
3700 BAY AREA BOULEVARD
HOUSTON, TEXAS 77058-1199

# Cost and Quality Planning for Large NASA Programs

The Software Cost and Quality Engineering methodology developed over the last two decades at IBM Federal Sector Division (FSD) in Houston is used to plan the NASA Space Station Data Management System (DMS). An ongoing project to capture this methodology, which is built on a foundation of experiences and "lessons learned," has resulted in the development of a PC-based tool that integrates cost and quality forecasting methodologies and data in a consistent manner. This tool, Software Cost and Quality Engineering Starter Set (SCQESS), is being employed to assist in the DMS costing exercises. At the same time, DMS planning serves as a forcing function and provides a platform for the continuing, iterative development, calibration, and validation and verification of SCQESS. The data that forms the cost and quality engineering database is derived from more than 17 years of development of NASA Space Shuttle software, ranging from low criticality, low complexity support tools to highly complex and highly critical onboard software.

## INTRODUCTION

Software cost and quality engineering is the systematic approach to the estimation, measurement, and control of software costs and quality on a project. This discipline provides the vital link between the concepts of economic analysis and the methodology of software engineering. The tasks involved in software cost and quality engineering are complex, and individuals with the knowledge and skill required are scarce. The accuracy and consistency of the results are often questionable. There is a definite need for tools to enable analysis by managers and planners who are not experts and to improve the results.

There are many instances in planning software development activities when a quick and easy to use cost and quality estimating tool would be of value. These include management consulting, proposal generation, and analysis of existing programs for problem correction or assurance. There is little time to learn a complex tool or to digest introductory user information. SCQESS requires no set-up, has a selectable demonstration, direct access to the tool functions, and contains examples of varied applications.

SCQESS resides on a tool sharing disk which is available to all company sites. The user has the option of viewing a demonstration program to illustrate the cost and quality estimation process. SCQESS includes a Lotus-based cost and quality estimation spreadsheet. The spreadsheet includes cost and quality models which are based on historical data and various criticality levels. If different models are required, the existing models can be easily modified. An example Lotus spreadsheet illustrates a completed estimation. This spreadsheet can be modified or a blank file is supplied if the work is entirely new. The tool can handle estimates involving many languages including Ada and can also be used for estimating reused elements.

Once the basic cost and quality estimates are completed, the user executes a Rayleigh Curve program to phase the estimates over time. A Rayleigh curve is a plot of a mathematical function which describes life cycle phenomena. A Rayleigh curve indicates whether the slope of the staffing curve is too steep or whether the error density of the project is too great at certain points in the process.

The cost and quality estimate can then be quickly modified to determine variance of results based on changes in assumptions. The cost and quality estimates support the strength of the plan being generated. The estimates can be used to analyze risk and mitigation plans. Examples of actual project cost and quality reports are available to the user. These examples include a project involving Ada, reuse and commercial elements; a project involving translation of code from one machine architecture to another; and a transaction oriented commercial system. The spreadsheets and the Rayleigh curves are included in the report.

## Software Life Cycle Costing and Quality Estimation Methodology

Past experience in managing large software projects such as Space Shuttle Onboard Software illustrates that accurate cost and quality estimates based on reliable historical data are essential to software planning. FSD Houston has collected extensive data from NASA and other software development projects over the last 17 years. This data includes source lines of code, productivity, error rates, computer usage, etc. for more than 250 projects. Historical data supports initial estimation of project size and estimation of effort from that size. The standard Rayleigh curve model converts estimated labor to a schedule and staffing profile - the elements of a cost plan. It also projects error estimates across the schedule to generate a quality plan.

The widely used Rayleigh curve models the typical build-up of staff and errors during the requirements and design phases, the peak for implementation, and the tail-off during the testing phase. The staffing schedule for an ideal project approximates a Rayleigh curve. During sustaining engineering, a minimum level of critical skills is required for effective maintenance. This steady-state staffing level forms the support line. It includes critical skills for requirements, design, implementation, testing, and management. The support line is a function of system size and productivity as well as unique skill requirements specific to the software being maintained.

The areas below the support line and above the maintenance tail of the Rayleigh curve is available for new development work. Sustaining engineering operational increment also corresponds to a Rayleigh curve. Each sustaining engineering effort can be modeled as the sum of a sequence of such curves. The sizing and scheduling of new development activities should be planned to provide a stable level of effort. Software maintenance which handles Discrepancy Reports can continue at a lower support level.

Historical project data supports software size, labor, and quality estimation. The Lotus-based Matrix Method function distributes the effort and errors over organizational elements. The Rayleigh Curve function generates a staffing and error discovery profile over time. Special models are available in other packages to adjust estimates involving expert systems, reusable software, reconfiguration, quality tracking, and maintenance.

## Planning the Space Station DMS Utilizing SCQESS

SCQESS has been used to assist in the costing of the Space Station Data Management System (DMS), a complex software system involving a distributed environment with multiple languages and applications. The DMS for Space Station is also affected by the requirements for long lifetime, permanent operations, remote integration, and phased technology insertion of productivity tools, applications, expert systems, etc. Major cost and quality drivers include the large size and diversity of the software, complexity, development support environment, off-the-shelf and reusable software, and criticality, which varies from one module to another. An example of the type of results — at the end of the intermediate step of development cost and quality estimation — obtained with SCQESS for the DMS planning is included in the presentation.

## Summary/Conclusions

The software cost and quality engineering methodology employed at IBM FSD Houston has been captured and integrated into a prototype tool, SCQESS. This PC-based tool integrates cost and quality planning methodology and data.

SCQESS has been employed to assist in the cost and quality planning of the Space Station DMS (Data Management System). It is providing a standardized approach for the DMS planning, which involves several individuals. It has made the process more efficient and has allowed a consistent approach to planning. The automation and captured methodology has established the foundation and mechanism enabling the continuing calibration and improvement in accuracy and consistency for Space Station DMS costing.

# VIEWGRAPH MATERIALS

# FOR THE

# K. RONE PRESENTATION

# Cost and Quality Planning for Large NASA Programs

Kyle Y. Rone

FEDERAL SECTOR DIVISION
3700 BAY AREA BOULEVARD
HOUSTON, TEXAS 77058 – 1199

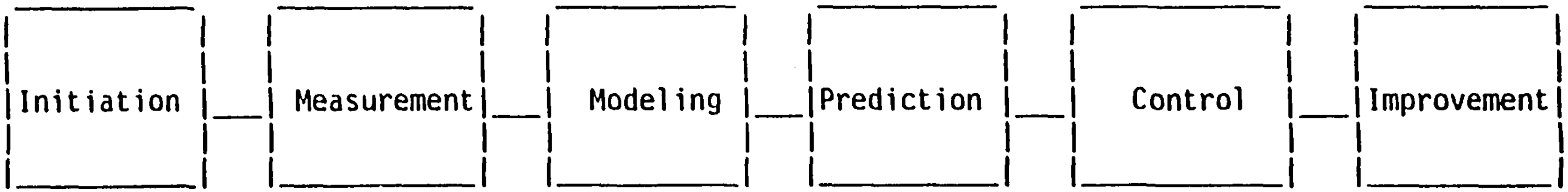
# Keys to Customer Satisfaction

- Compliant Product
- Within Budget
- On Time
- Appropriate Quality Level

Concurrently

Consistently

# Approach



Initiation

Measurement

Modeling

Prediction

Control

Improvement

| Keys \ Steps | Initiation | Measurement | Modeling | Prediction | Control | Improvement |
|---|---|---|---|---|---|---|
| Product | • Process<br>• Interim Product<br>• Procedure Order<br>• Tailoring Mechanism | • Size<br>• Process Proficiency | • Process Models | • Process Tailoring | • Control Points | • Modify Process<br>• Modify Ordering<br>• Automation |
| Cost | | • Function Driven Cost<br>• Schedule Driven Cost<br>• Complexity<br>• Criticality | • Factor Models<br>• % Models<br>• Phasing | • Calibrate To Process<br>• Function Driven | • Cost Management | • Modify Cost Models |
| Schedule | | • Process Elapsed Time<br>• Process Order | • Schedule Rules Of Thumb | • Phased Cost and Errors | • Schedule Management | • Modify Schedule Rules of Thumb |
| Quality | | • Inspection Errors<br>• Process Errors<br>• Product Errors<br>• Total Errors | • Life Cycle Errors<br>• % Models<br>• Phasing | • Calibrate To Process<br>• Function Driven<br>• Cost Driven | • Quality Management | • Modify Quality Models |

## MODEL RECONCILIATION

- ALT (EARLY MODEL)
  - PROJECTED = 10429 MM
  - DEVELOPMENT PART OF ALT = 9403 MM ACTUALS
  - ERROR IS 1026 MM OR 11% HIGH

- STS-1 (MIDDLE MODEL)
  - PROJECTED = 8905 MM
  - DEVELOPMENT PART OF STS-1 = 9521 MM ACTUALS
  - ERROR IS 616 MM OR 6% LOW

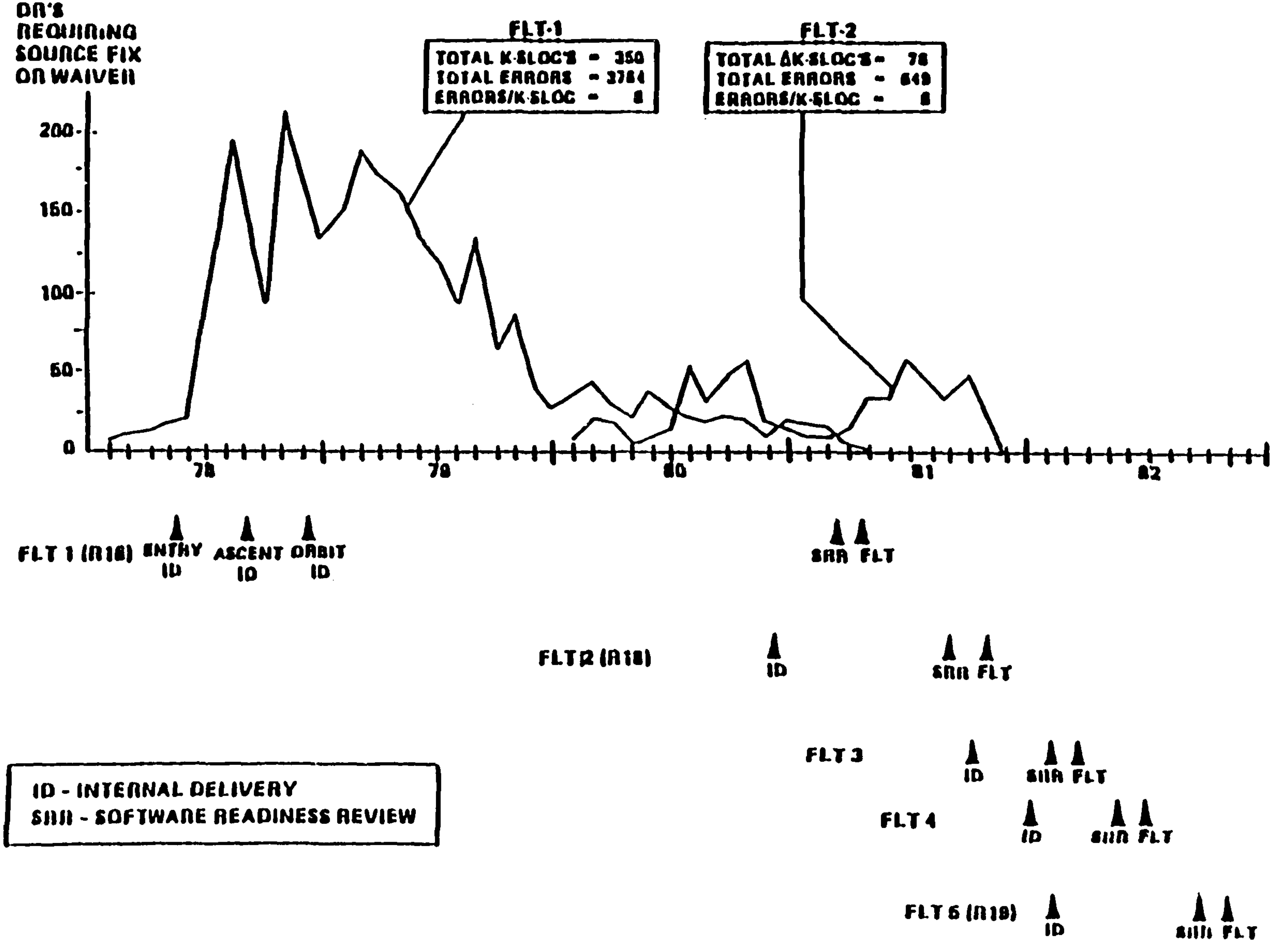
- STS-2 THRU STS-5 (MIDDLE MODEL)
  - PROJECTED = 5864 MM
  - DEVELOPMENT PART OF STS-2 THRU STS-5 = 5994 MM ACTUALS
  - ERROR IS 130 MM OR 2% LOW

- TOTAL
  - TOTAL PROJECTED = 10429 + 8905 + 5864 = 25198 MM
  - TOTAL ACTUALS = 9403 + 9521 + 5994 = 24918 MM
  - ERROR IS 280 MM OR 1% HIGH

- STS-1 THRU STS-5 (MIDDLE MODEL)
  - TOTAL STS-1 THRU STS-5 PROJECTED = 8905 + 5864 = 14769 MM
  - TOTAL ACTUALS = 9521 + 5994 = 15515 MM
  - ERROR IS 746 MM OR 5%

# MODEL RECONCILIATION

- SDL
  - (900,000/230) 1.4 = 5478 MM
  - SDL ACTUALS = 5730 MM
  - ERROR IS 252 MM OR 4% LOW
- SPF
  - ((250,000/230) + (315,000/250) + (385,000/475)) 1.4 = 4421 MM
  - SPF ACTUALS = 4033
  - ERROR IS 388 MM OR 10% HIGH
- TOTAL
  - TOTAL PROJECTED = 5478 + 4421 = 9899 MM
  - TOTAL ACTUALS = 5730 + 4033 = 9763 MM
  - ERROR IS 136 MM OR 1% HIGH



ID - INTERNAL DELIVERY
SRR - SOFTWARE READINESS REVIEW

```
              ERROR DISCOVERY PROFILE FOR PROJECT       04-13-1990

        SOFTWARE                                    ERRORS PER KSLOC
   DEVELOPMENT ACTIVITY                  USER  PROVIDED        PROGRAM  ESTI
 ----------------------------------------------------------------------------
  HIGH LEVEL DESIGN INSPECTION                   0.11             1.0062

  LOW LEVEL DESIGN INSPECTION                    4.20             3.1667

  CODE INSPECTION                                6.80             6.9371

  UNIT TEST                                      ---              3.721

  INTEGRATION TEST                               ---              2.011

  SYSTEM TEST                                    ---              0.854

  LATENT ERROR                                   ---              0.598
 ----------------------------------------------------------------------------
THE PATTERN PROJECTED IS PATTERN NUMBER:  14
THE ESTIMATED TOTAL LIFETIME ERROR CONTENT IS:   18.29

PRESS:ENTER FOR MENU SCREEN ;PrtSc & Shift TO PRINT SCREEN: Y TO GRAPH DAT
```

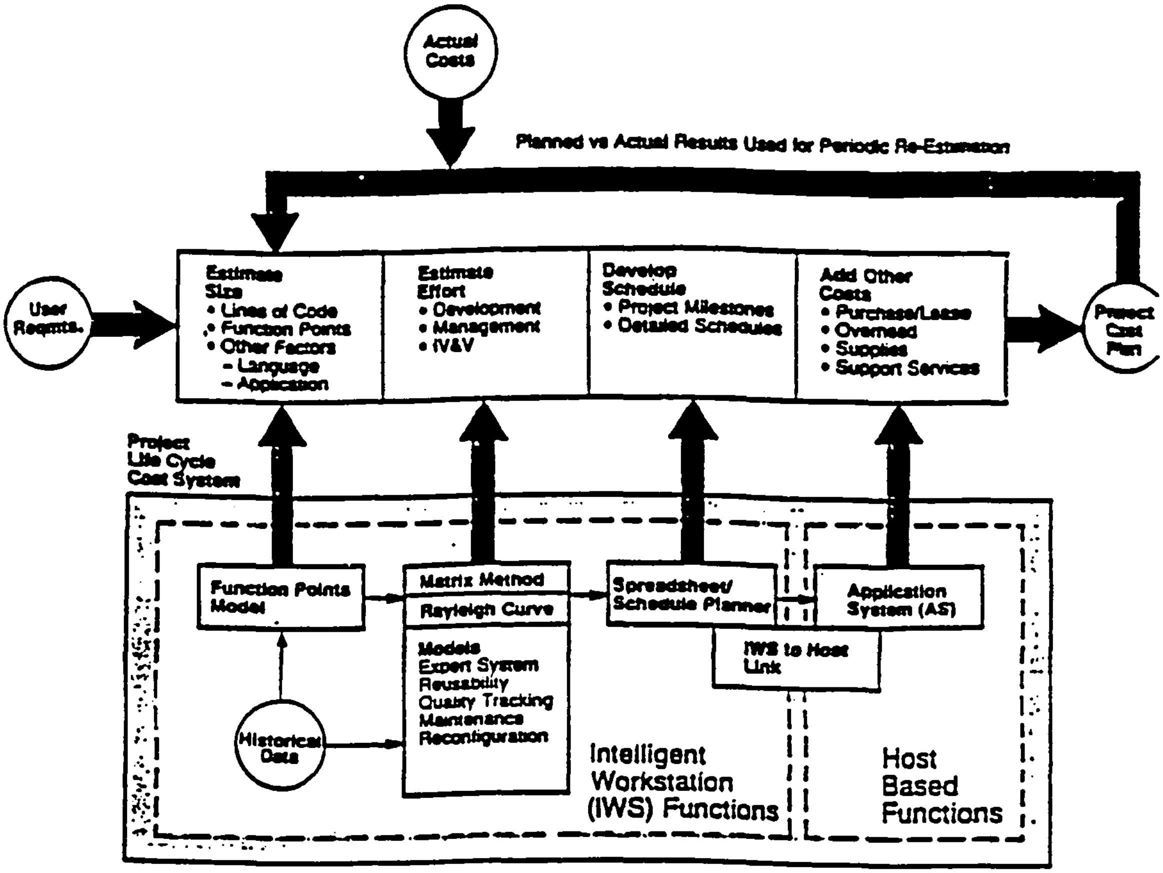
# PROJECTION COMPARISON

| | STEER | ACTUAL |
|---|---|---|
| • TOTAL RELEASE | | |
| - TOTAL INSERTED ERROR RATE | 18.3 | 13.3 |
| - PRODUCT ERROR RATE | .6 | 1.0 |
| • HOST | | |
| - TOTAL INSERTED ERROR RATE | 13.0 | 8.5 |
| - PRODUCT ERROR RATE | .4 | .3 |
| • W/S | | |
| - TOTAL INSERTED ERROR RATE | 35.1 | 28.8 |
| - PRODUCT ERROR RATE | 1.1 | 3.2 |

# PROJECTION COMPARISON
# (INCLUDING PROCESS DATA)

| | STEER | ACTUAL |
|---|---|---|
| • TOTAL RELEASE | | |
| - TOTAL INSERTED ERROR RATE | 13.54 | 13.28 |
| - PRODUCT ERROR RATE | 1.25 | 1.0 |
| • HOST | | |
| - TOTAL INSERTED ERROR RATE | 8.47 | 8.52 |
| - PRODUCT ERROR RATE | .28 | .33 |
| • W/S | | |
| - TOTAL INSERTED ERROR RATE | 28.28 | 28.82 |
| - PRODUCT ERROR RATE | 2.62 | 3.16 |

# Software Life Cycle Costing (LCC) Methodology



Actual Costs

Planned vs Actual Results Used for Periodic Re-Estimation

User Reqmts.

Estimate Size
- Lines of Code
- Function Points
- Other Factors
  - Language
  - Application

Estimate Effort
- Development
- Management
- IV&V

Develop Schedule
- Project Milestones
- Detailed Schedules

Add Other Costs
- Purchase/Lease
- Overhead
- Supplies
- Support Services

Project Cost Plan

Project Life Cycle Cost System

Function Points Model

Matrix Method

Rayleigh Curve

Models
Expert System
Reusability
Quality Tracking
Maintenance
Reconfiguration

Spreadsheet/ Schedule Planner

Application System (AS)

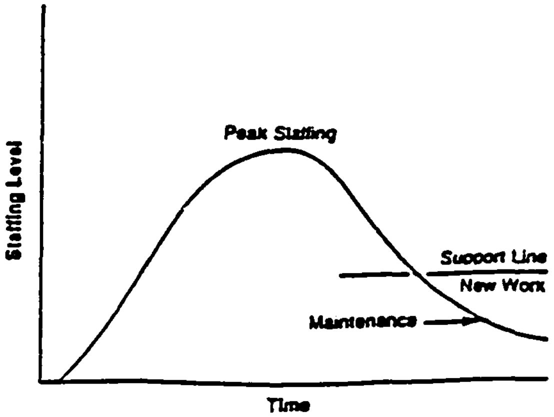
IWS to Host Link

Historical Data

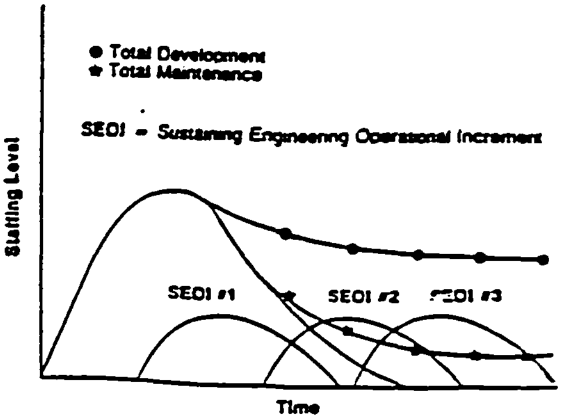
Intelligent Workstation (IWS) Functions

Host Based Functions

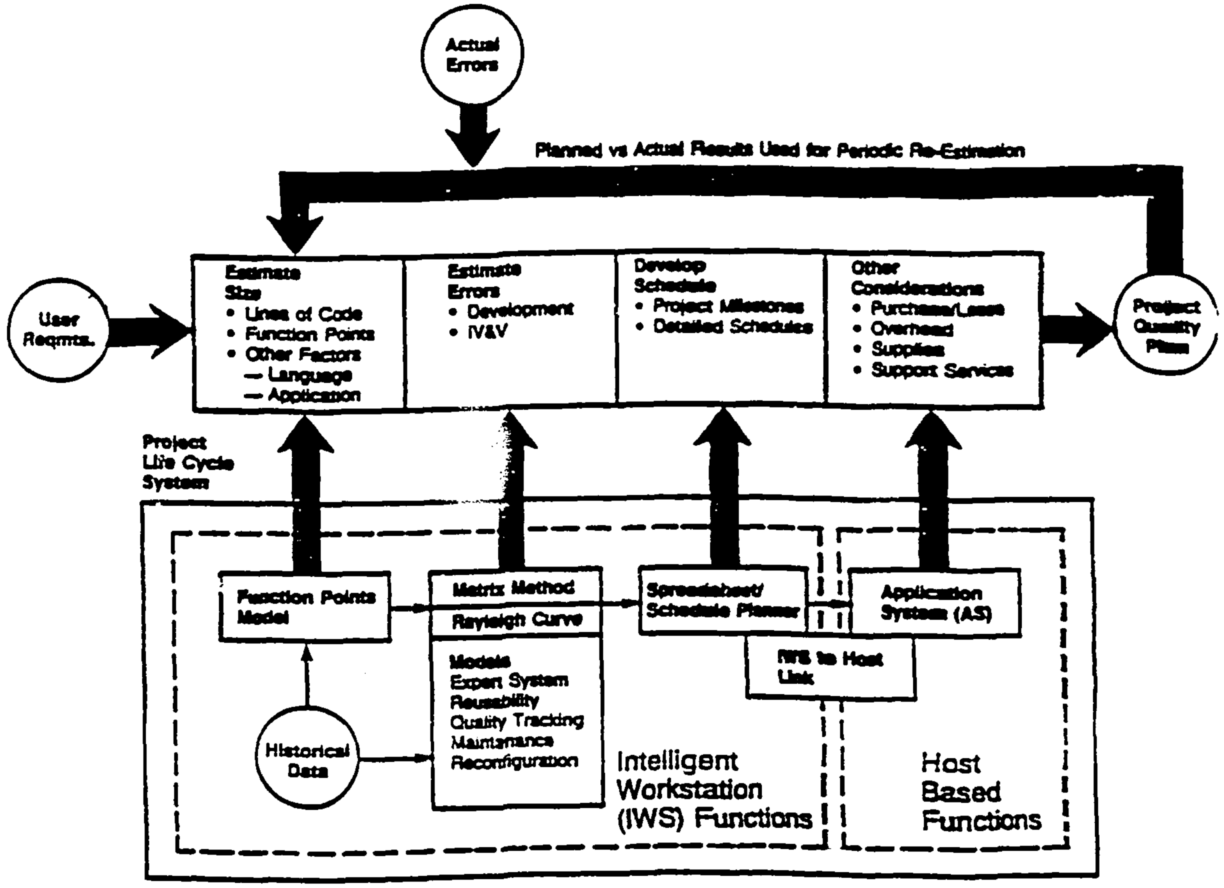
*Staffing over a software development project is modeled using a Rayleigh Curve.*



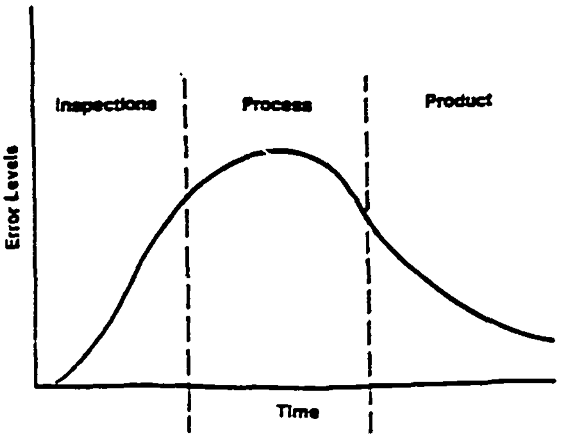
*Software sustaining engineering is modeled as a sequence of overlapping Rayleigh Curves.*



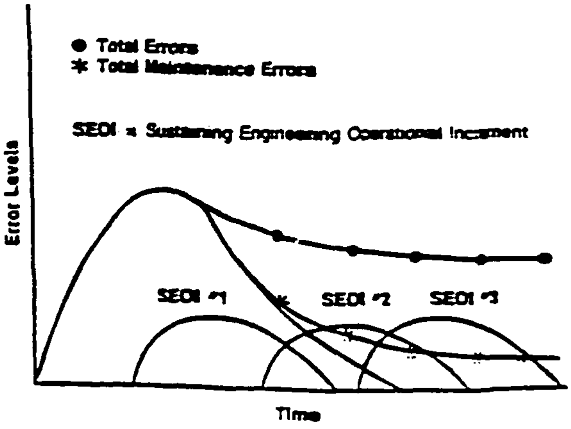
## Software Life Cycle Quality Management

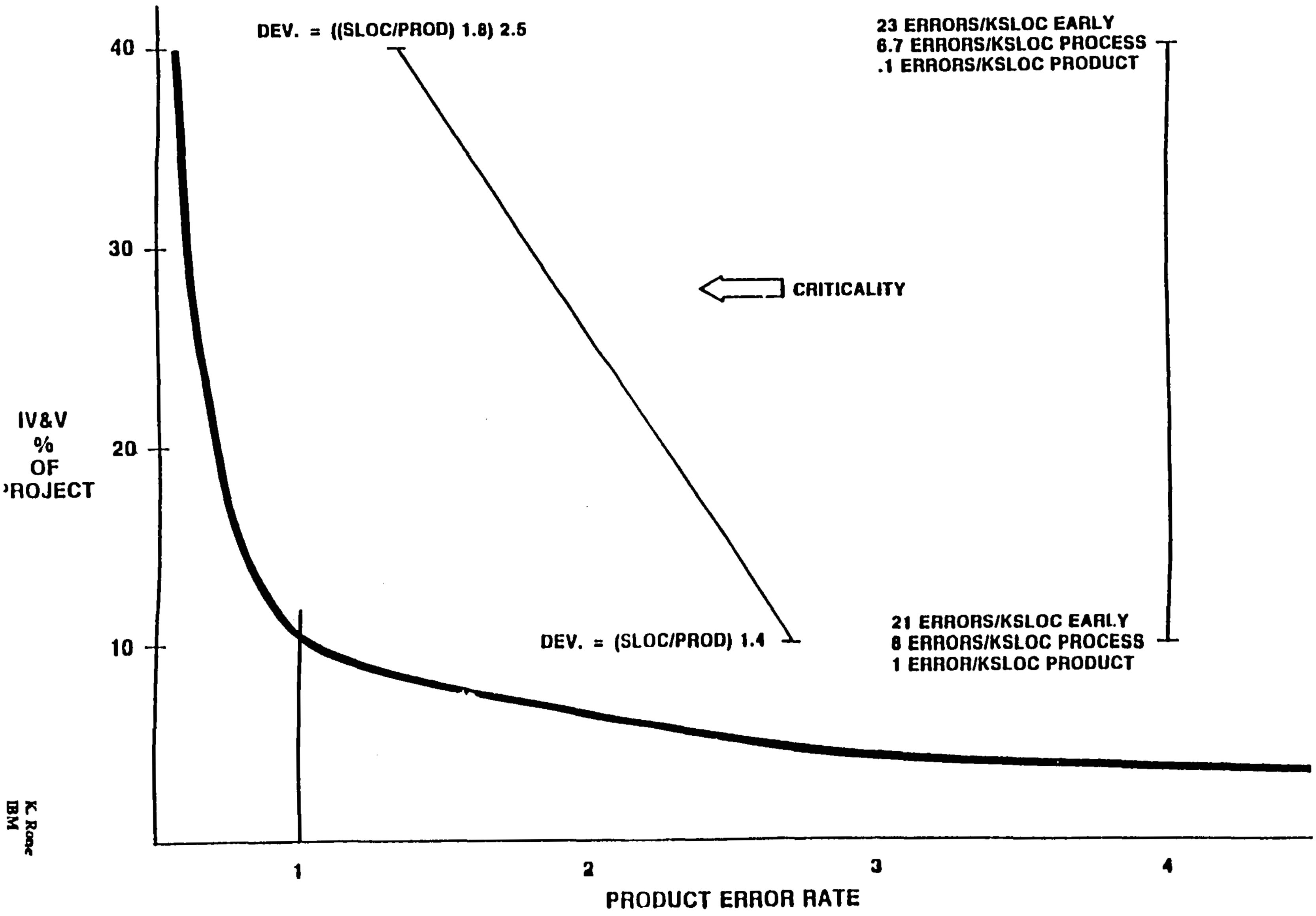


*Error discovery over a software development project is modeled using a Rayleigh Curve.*

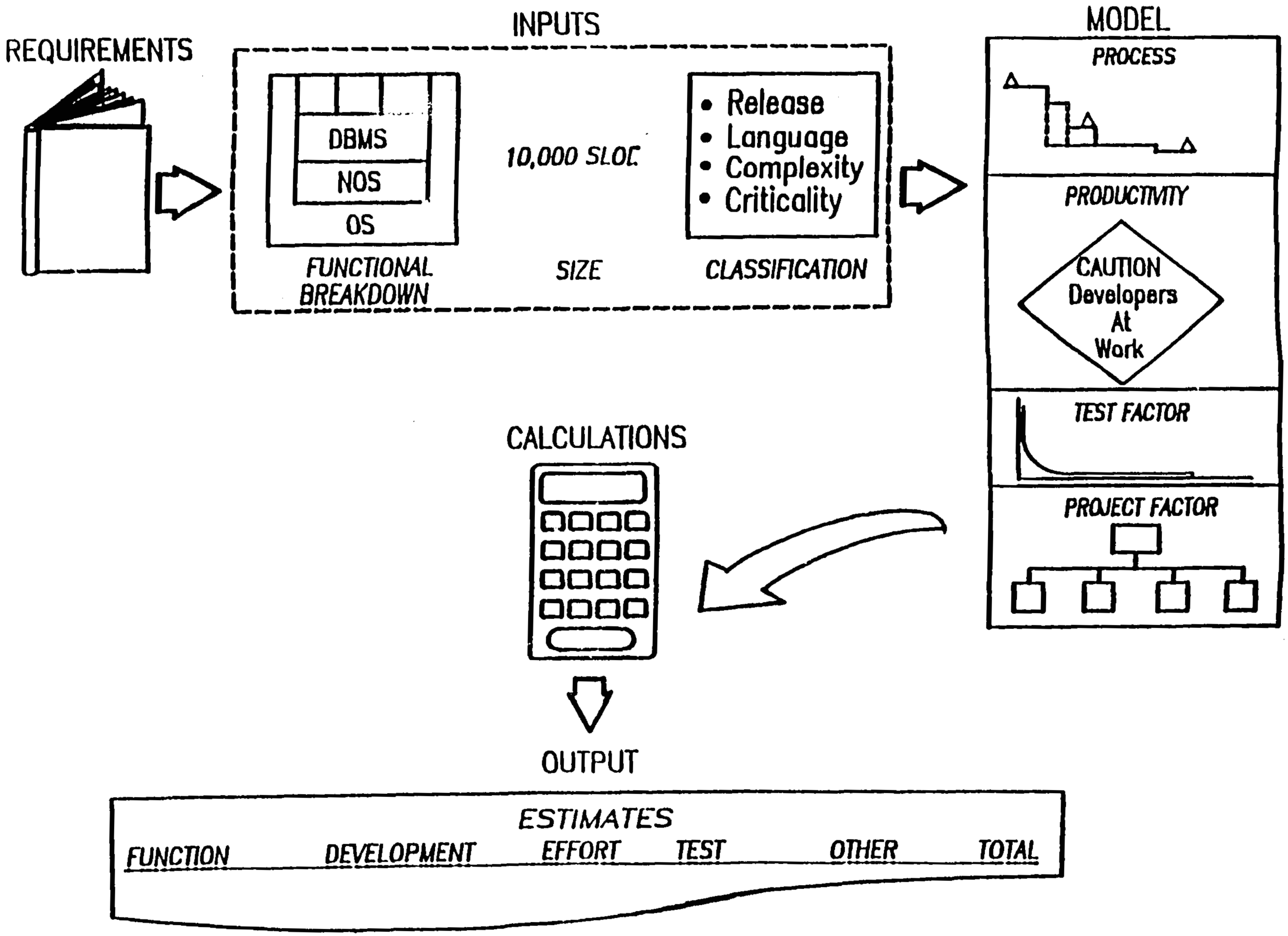


*Software sustaining engineering is modeled as a sequence of overlapping Raleigh Curves.*




DEV. = ((SLOC/PROD) 1.8) 2.5
23 ERRORS/KSLOC EARLY
6.7 ERRORS/KSLOC PROCESS
.1 ERRORS/KSLOC PRODUCT
40
30
CRITICALITY
IV&V
%
OF
ᐧROJECT
20
10
DEV. = (SLOC/PROD) 1.4
21 ERRORS/KSLOC EARLY
8 ERRORS/KSLOC PROCESS
1 ERROR/KSLOC PRODUCT
1
2
3
4
PRODUCT ERROR RATE

# SOFTWARE COSTING METHODOLOGY



INPUTS

REQUIREMENTS

DBMS

NOS

OS

FUNCTIONAL BREAKDOWN

10,000 SLOC

SIZE

- Release
- Language
- Complexity
- Criticality

CLASSIFICATION

MODEL

PROCESS

PRODUCTIVITY

CAUTION Developers At Work

TEST FACTOR

PROJECT FACTOR

CALCULATIONS

OUTPUT

ESTIMATES

FUNCTION DEVELOPMENT EFFORT TEST OTHER TOTAL

NOS

| AREA | NEW (SLOC) | COTS (SLOC) | REL. | CRIT. | LANG. | COMP. | DEV. | VERIF. | INDIRECT | TOTAL |
|---|---|---|---|---|---|---|---|---|---|---|
| NETWORK LAYERS | 3200 | | 2 | SC | ADA | C | 24.6 | 19.7 | 66.5 | 110.8 |
| TRANSPORT LAYERS | 3900 | | 2 | SC | ADA | C | 30 | 24 | 81 | 135 |
| SESSION LAYERS | 5700 | | 2 | SC | ADA | C | 43.8 | 35 | 118.2 | 197 |
| PRESENTATION LAYERS | | | | | | | | | | |
| APPLICATION LAYERS | | | | | | | | | | |
| CASE | 1800 | | 2 | SC | ADA | C | 13.8 | 11 | 37.2 | 62 |
| RJE | 5000 | | 2 | SC | ADA | C | 38.5 | 30.8 | 104 | 173.3 |
| DIRECT ACCESS | 2000 | | 3 | SC | ADA | C | 13.3 | 10.6 | 35.9 | 59.8 |
| NETWORK MGMT. | 16000 | | 3 | SC | ADA | C | 106.7 | 85.4 | 288.2 | 480.3 |
| APPLICATION SERV. | 2000 | | 2 | SC | ADA | C | 15.4 | 12.3 | 41.6 | 69.3 |
| TOTALS: | 39600 | | | | | | 286.1 | 228.8 | 772.6 | 1287.5 |

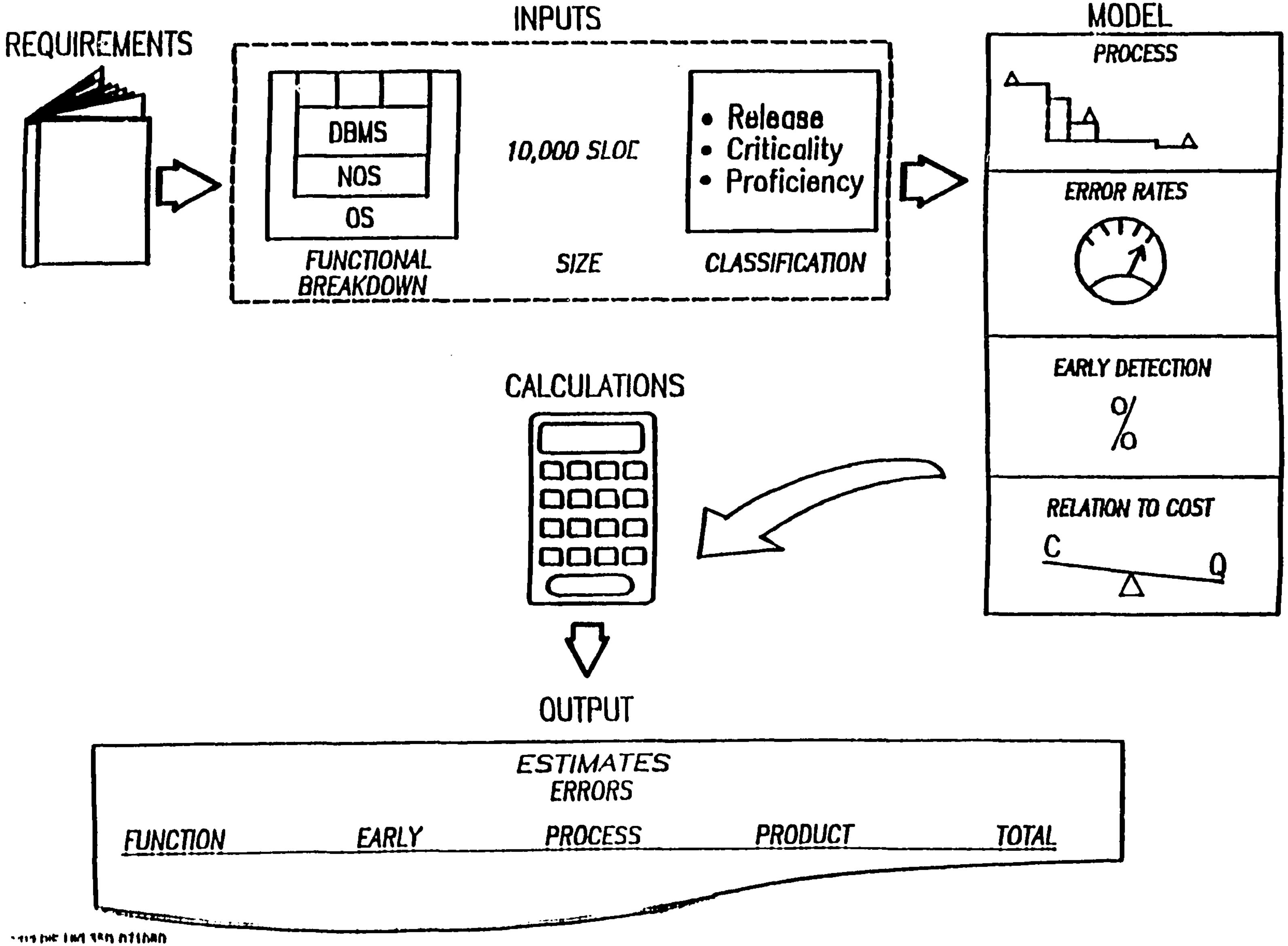
OS

| AREA | NEW (SLOC) | COTS (SLOC) | REL. | CRIT. | LANG. | COMP. | DEV. | VERIF. | INDIRECT | TOTAL |
|---|---|---|---|---|---|---|---|---|---|---|
| KERNAL | | | | | | | | | | |
| TASK | | 2300 | 1 | SC | ASSM | L | 1 | 0.8 | 2.7 | 4.5 |
| MEMORY MGMT. | | 5400 | 1 | SC | ASSM | L | 2.3 | 1.8 | 6.2 | 10.3 |
| INTERUPT | | 6500 | 1 | SC | ASSM | L | 2.8 | 2.2 | 7.5 | 12.5 |
| I/O MGMT. | | 5000 | 1 | SC | ASSM | L | 2.2 | 1.8 | 6 | 10 |
| RESOURCE MGMT. | | 3000 | 1 | SC | ASSM | L | 1.3 | 1 | 3.5 | 5.8 |
| INTER PROC. COMM. | | 1000 | 1 | SC | ASSM | L | 0.4 | 0.3 | 1.1 | 1.8 |
| FAULT MGMT. | | 1000 | 1 | SC | ASSM | L | 0.4 | 0.3 | 1.1 | 1.8 |
| MPAC UNIQUE | | 16700 | 1 | SC | ASSM | L | 7.3 | 5.8 | 19.7 | 32.8 |
| MPAC UNIQUE | 2200 | | 1 | SC | ADA | C | 22 | 17.6 | 59.4 | 99 |
| SDP UNIQUE | 1800 | | 1 | SC | ADA | C | 18 | 14.4 | 48.6 | 81 |
| NIU UNIQUE | 1800 | | 1 | SC | ADA | C | 18 | 14.4 | 48.6 | 81 |
| EDP UNIQUE | 1800 | | 1 | SC | ADA | C | 18 | 14.4 | 48.6 | 81 |
| TOTALS: | 7600 | 40900 | | | | | 93.7 | 74.8 | 253 | 421.5 |

## SUMMARY CHART

| AREA | NEW (SLOC) | COTS (SLOC) | REL. | CRIT. | LANG. | COMP. | DEV. | VERIF. | INDIRECT | TOTAL |
|---|---|---|---|---|---|---|---|---|---|---|
| NOS | 39600 | | | | | | 286.1 | 228.8 | 772.6 | 1287.5 |
| OS | 7600 | 40900 | | | | | 93.7 | 74.8 | 253 | 421.5 |
| ADA RTE | 41500 | | | | | | 383.2 | 306.6 | 1034.8 | 1724.6 |
| STANDARD SERVICES | 93100 | | | | | | 606.9 | 390.7 | 1248.2 | 2245.8 |
| DMS SYSTEM MANAGEMENT | 20700 | | | | | | 119.9 | 73.1 | 229.9 | 422.9 |
| DATA STORAGE & RETRIEVAL | 26000 | 50000 | | | | | 196.2 | 109.3 | 333.2 | 638.7 |
| USE | 223500 | | | | | | 1403.5 | 452.6 | 1449.1 | 3305.2 |
| USE CONTINUED | 144000 | 170000 | | | | | 521.5 | 52.3 | 172.2 | 746 |
| OMA | 42000 | | | | | | 280 | 28 | 92.4 | 400.4 |
| TOTALS: | 638000 | 260900 | | | | | 3891 | 1716.2 | 5605.4 | 11212.6 |

# SOFTWARE QUALITY FORECASTING



REQUIREMENTS

INPUTS

DBMS

NOS

OS

FUNCTIONAL BREAKDOWN

10,000 SLOC

SIZE

- Release
- Criticality
- Proficiency

CLASSIFICATION

MODEL

PROCESS

ERROR RATES

EARLY DETECTION

%

RELATION TO COST

C

Q

CALCULATIONS

OUTPUT

ESTIMATES

ERRORS

FUNCTION EARLY PROCESS PRODUCT TOTAL

NOS

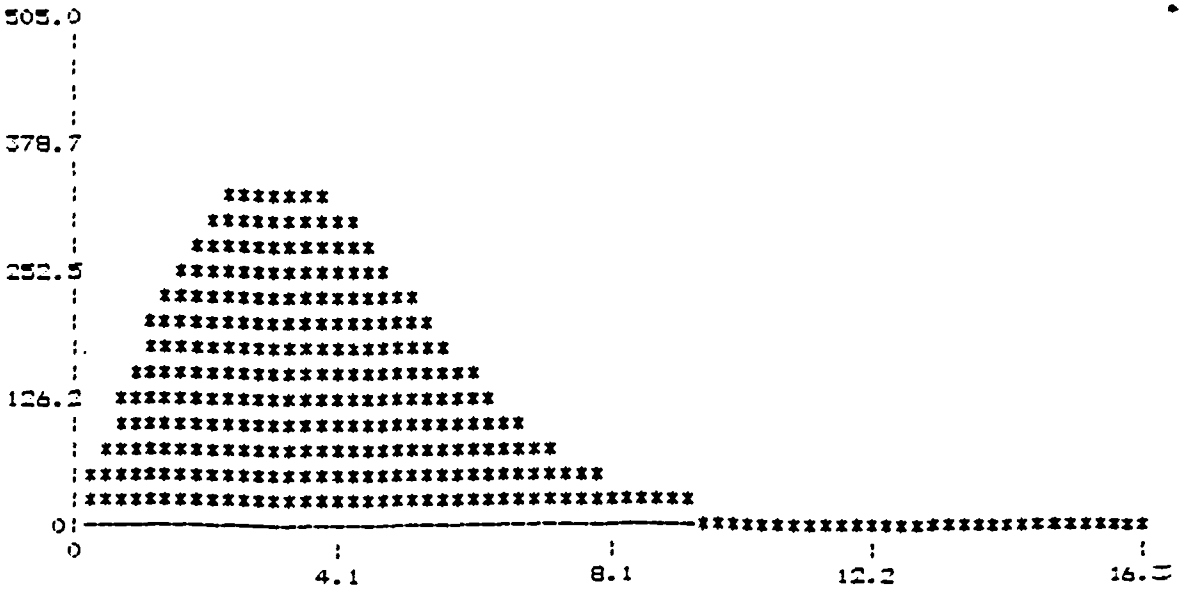
| AREA | NEW (SLOC) | COTS (SLOC) | REL. | CRIT. | PROJECT PROF. | DVPMENT PROF. | EARLY | PROCESS | PRODUCT | TOTAL |
|---|---|---|---|---|---|---|---|---|---|---|
| NETWORK LAYERS | 3200 | | 2 | SC | GD | AV | 62.4 | 33.3 | 0.3 | 96.0 |
| TRANSPORT LAYERS | 3900 | | 2 | SC | GD | AV | 76.1 | 40.6 | 0.4 | 117.0 |
| SESSION LAYERS | 5700 | | 2 | SC | GD | AV | 111.2 | 59.3 | 0.6 | 171.0 |
| PRESENTATION LAYERS | | | | | | | | | | |
| APPLICATION LAYERS | | | | | | | | | | |
| CASE | 1800 | | 2 | SC | GD | AV | 35.1 | 18.7 | 0.2 | 54.0 |
| RJE. | 5000 | | 2 | SC | GD | AV | 97.5 | 52.0 | 0.5 | 150.0 |
| DIRECT ACCESS | 2000 | | 3 | SC | GD | AV | 42.0 | 17.8 | 0.2 | 60.0 |
| NETWORK MGMT. | 16000 | | 3 | SC | GD | AV | 336.0 | 142.4 | 1.6 | 480.0 |
| APPLICATION SERV. | 2000 | | 2 | SC | GD | AV | 39.0 | 20.8 | 0.2 | 60.0 |
| TOTALS: | 39600 | | | | | | 799.2 | 384.8 | 4.0 | 1188.0 |

OS

| AREA | NEW (SLOC) | COTS (SLOC) | REL. | CRIT. | PROJECT PROF. | DVPMENT PROF. | EARLY | PROCESS | PRODUCT | TOTAL |
|---|---|---|---|---|---|---|---|---|---|---|
| KERNAL | | | | | | | | | | |
| TASK | | 2300 | 1 | SC | GD | AV | 4.1 | 2.7 | 0.0 | 6.9 |
| MEMORY MGMT. | | 5400 | 1 | SC | GD | AV | 9.7 | 6.4 | 0.1 | 16.2 |
| INTERUPT | | 6500 | 1 | SC | GD | AV | 11.7 | 7.7 | 0.1 | 19.5 |
| I/O MGMT. | | 5000 | 1 | SC | GD | AV | 9.0 | 6.0 | 0.1 | 15.0 |
| RESOURCE MGMT. | | 3000 | 1 | SC | GD | AV | 5.4 | 3.6 | 0.0 | 9.0 |
| INTER PROC. COMM. | | 1000 | 1 | SC | GD | AV | 1.8 | 1.2 | 0.0 | 3.0 |
| FAULT MGMT. | | 1000 | 1 | SC | GD | AV | 1.8 | 1.2 | 0.0 | 3.0 |
| MPAC UNIQUE | | 16700 | 1 | SC | GD | AV | 30.1 | 19.9 | 0.2 | 50.1 |
| MPAC UNIQUE | 2200 | | 1 | SC | GD | AV | 39.6 | 26.2 | 0.2 | 66.0 |
| SBP UNIQUE | 1800 | | 1 | SC | GD | AV | 32.4 | 21.4 | 0.2 | 54.0 |
| NIU UNIQUE | 1800 | | 1 | SC | GD | AV | 32.4 | 21.4 | 0.2 | 54.0 |
| ESP UNIQUE | 1800 | | 1 | SC | GD | AV | 32.4 | 21.4 | 0.2 | 54.0 |
| TOTALS: | 7600 | 40900 | | | | | 210.4 | 139.1 | 1.2 | 350.7 |

S U M M A R Y   C H A R T

| AREA | NEW (SLOC) | COTS (SLOC) | REL. | CRIT. | PROJECT PROF. | DVPMENT PROF. | EARLY | PROCESS | PRODUCT | TOTAL |
|---|---|---|---|---|---|---|---|---|---|---|
| NOS | 39600 | | | | | | 799.2 | 384.8 | 4.0 | 1188.0 |
| OS | 7600 | 40900 | | | | | 210.4 | 139.1 | 1.2 | 350.7 |
| ABA RTE | 41500 | | | | | | 771.0 | 469.9 | 4.2 | 1245.0 |
| STANDARD SERVICES | 93100 | | | | | | 1811.6 | 957.7 | 23.7 | 2793.0 |
| DMS SYSTEM MANAGEMENT | 20700 | | | | | | 433.4 | 181.2 | 6.4 | 621.0 |
| DATA STORAGE & RETRIEVAL | 26000 | 50000 | | | | | 629.6 | 289.0 | 11.5 | 930.0 |
| USE | 223500 | | | | | | 4395.0 | 2147.5 | 162.5 | 6705.0 |
| USE CONTINUED | 144000 | 170000 | | | | | 3295.5 | 1373.5 | 161.0 | 4830.0 |
| OMA | 42000 | | | | | | 882.0 | 336.0 | 42.0 | 1260.0 |
| TOTALS: | 638000 | 260900 | | | | | 13227.6 | 6278.7 | 416.4 | 19922.7 |


505.0
378.7
252.5
126.2
0
0
4.1
8.1
12.2
16.3

N92
19434
UNCLAS

# Effect Of Formal Specifications On Program Complexity And Reliability: An Experimental Study [*]

Amrit L. Goel[†]
Department of Electrical and Computer Engineering
Syracuse University
Syracuse, NY 13244
(315)443-4350
goel@suvm.acs.syr.edu

Swarupa N. Sahoo[‡]
School of Computer and Information Science

October 16, 1990

**Abstract**

In this paper we present the results of an experimental study undertaken to assess the improvement in program quality by using formal specifications. Specifications in the Z notation were developed for a simple but realistic anti-missile system. These specifications were then used to develop 2 versions in C by 2 programmers. Another set of 3 versions in Ada were independently developed from informal specifications in English. A comparison of the reliability and complexity of the resulting programs suggests the advantages of using formal specifications in terms of number of errors detected and fault avoidance.

[*] Funding during the course of this project was provided via US Army contract no. DAKF 11-89-C-0038 and NASA Grant no. NAG-1-806.

[†] Professor, ECE and CIS

[‡] Graduate Assistant, CIS

# EXTENDED ABSTRACT

Specification languages are widely accepted as a stepping stone for design and development of a complex software system [1, 2, 3, 4, 5]. The advantages of a specification language are often not immediately clear in terms of program quality and reliability. Proving an executable program correct for complex systems is computationally an intractable task [6]. Also "an effective testing strategy which is reliable for all programs cannot be constructed" [7]. In such a setting, formal specification languages coupled with structured design methodologies [8] provide a streamlined approach for software design and development.

In this experimental investigation, we study the effect of the specification language Z [11] on program reliability and complexity. For our experiment we chose the NASA Launch Interceptor Problem(LIP) since it has been used extensively for several other studies in software reliability and fault tolerance. It is a simple but realistic representation of an anti-missile system. The original specifications were taken from Knight and Leveson [12]. The LIP is a constraint satisfaction problem, a solution to which is a decision procedure which takes a set of input points and launch characteristics to evaluate a set of initial launch conditions, called the preliminary unlocking matrix. The procedure then evaluates a logical combination(the combination is decided by an input matrix) of the initial conditions called the final unlocking vector the components of which collectively decide if the launch signal should be true or otherwise.

The experiment consisted of usual phases of software design and development with minor differences. In the specification phase a set of specifications of the requirements was developed in the Z notation and was validated by other specifiers. Several versions were developed based on informal and formal requirements specifications separately, by independent groups of programmers. For testing, a hybrid approach [13] was developed based on functional and structural information about the LIP. For generating test cases, the hypothetical launch conditions were divided into 7 relatively independent groups. The truth values of one of the groups was fixed a priori, and an input data set was constructed to satisfy the prefixed truth values of this group and the truth values of the rest were computed against the input set. Such manually designed test cases were used to test each program. After debugging, when the computations of launch conditions for all the versions match, the cyclomatic complexity measure [9] is applied to compute internal complexity of each individual module.

Also computed are the external complexity due to the interconnections between various modules based on "information flow" concepts [10], and finally the total system complexity as a weighted sum of internal and external complexities.

The versions based on informal requirements are found to be afflicted with usual problems caused by the inherent ambiguities in the informal requirements. However, a significant reduction was observed in the number of errors detected in the testing phase in case of the versions based on formal requirements. Further, complexity measures strongly suggest that versions based on formal specifications are less complex and more reliable than those based on informal requirements. The study also suggests that the formal specifications developed through several successive stages of operations refinement lend themselves to an automatic modular program development(special case of a divide and conquer technique) in an optimal way, and thus reduce the error-proneness of the program and make it more reliable.

### Summary of Experimental Results

### I. Productivity:

*Table 1 - specification development time*

| Version number | Total Specification Development Time(hours) |
|---|---|
| Spec I | 47 |

*Table 2 - program development time*

| Version number | Total Program Development Time(hours) |
|---|---|
| Cver I | 18 |
| Cver II | 38 |
| Adaver I | 76 |
| Adaver II | 73 |
| Adaver III | 89 |

### II. Reliability*(in terms of number of errors detected)*

*Table 3 - Number of errors detected during development*

| Version number | Total Number of Errors |
|---|---|
| Cver I | 3 |
| Cver II | 8 |
| Adaver I | 8 |
| Adaver II | 7 |
| Adaver III | 4 |

*Table 4 - Number of errors detected during testing*

| Version number | Total Number of Errors |
|---|---|
| Cver I | 0 |
| Cver II | 7 |
| Adaver I | 13 |
| Adaver II | 11 |
| Adaver III | 8 |

# References

[1] D. L. Parnas, *The Use of Precise Specification in the Development of Software*, Proceedings of IFIP Congress 77, Toronto, 1977.

[2] I. Hayes, editor. *Specification Case Studies.* Prentice-Hall International, London, 1987.

[3] C. Morgan, B. Sufrin, *Specification of the Unix Filing System*, IEEE Trans. Software Eng., March 1984.

[4] D. Bjorner, C. B. Jones. *Formal Specification and Software Development.* Prentice-Hall International, London, 1982.

[5] C. B. Jones, *Systematic Software Development using VDM*, Prentice-Hall International, 1986.

[6] D. L. Parnas, *When can Software be Trustworthy ?*, COMPASS-86 Conference, Washington, D. C., July 1986.

[7] W. E. Howden, *Reliability of the Path Analysis Testing Strategy*, IEEE Trans. Software Eng., Sep 1976.

[8] E. Yourdon, L. L. Constantine, *Structured Design*. Prentice-Hall Inc., 1979.

[9] T. J. McCabe, *A Complexity Measure*, IEEE Trans. Software Eng., Sep 1981.

[10] K. S. Lew, T. S. Dillon, K. E. Forward, *Software Complexity and its Impact on Software Reliability*, IEEE Trans. on Software Eng., Nov 1988.

[11] J. M. Spivy, *The Z Notation: A Reference Manual*, Prentice Hall International, 1989.

[12] J. Knight, N. Leveson, *An Experimental Evaluation Of The Assumption Of Independence In Multi-version Programming*, IEEE Trans. on Software Eng., Jan 1986.

[13] A. L. Goel, *An Experimental Investigation Into Software Reliability*, RADC-TR-88-213, Oct 1988.

VIEWGRAPH MATERIALS

FOR THE

A. GOEL PRESENTATION

# EFFECT OF FORMAL SPECIFICATIONS ON PROGRAM COMPLEXITY AND RELIABILITY: AN EXPERIMENTAL STUDY

Amrit L. Goel[1]
Swarupa N. Sahoo[2]

Syracuse University
Syracuse, NY 13244

Presented at the Fifteenth Annual Software Engineering Workshop (SEL) held at NASA/G SFC, Greenbelt, MD, November 28-29, 1990.

[1] Professor, Electrical and Computer Engineering and School of Computer and Information Science, (315) 443-4350, goel@suvm.acs.syr.edu.

[2] Research Assistant

# OUTLINE

- Objectives of Study
- Experimental Appraoch
- Results of Experiment
- Comparison with Versions from Informal Specifications
- Fault Aviodance by Using Z
- Concluding Remarks

# OUTLINE

- Objectives of Study

- Experimental Appraoch

  - LIP Problem
  - Z-Specifications
  - Experiment Description

- Results of Experiment

  - Development Effort
  - Size and Complexity Metrics
  - Errors During Development
  - Errors During "Operational Testing"

- Comparison with Versions from Informal Specifications

- Fault Aviodance by Using Z

- Concluding Remarks

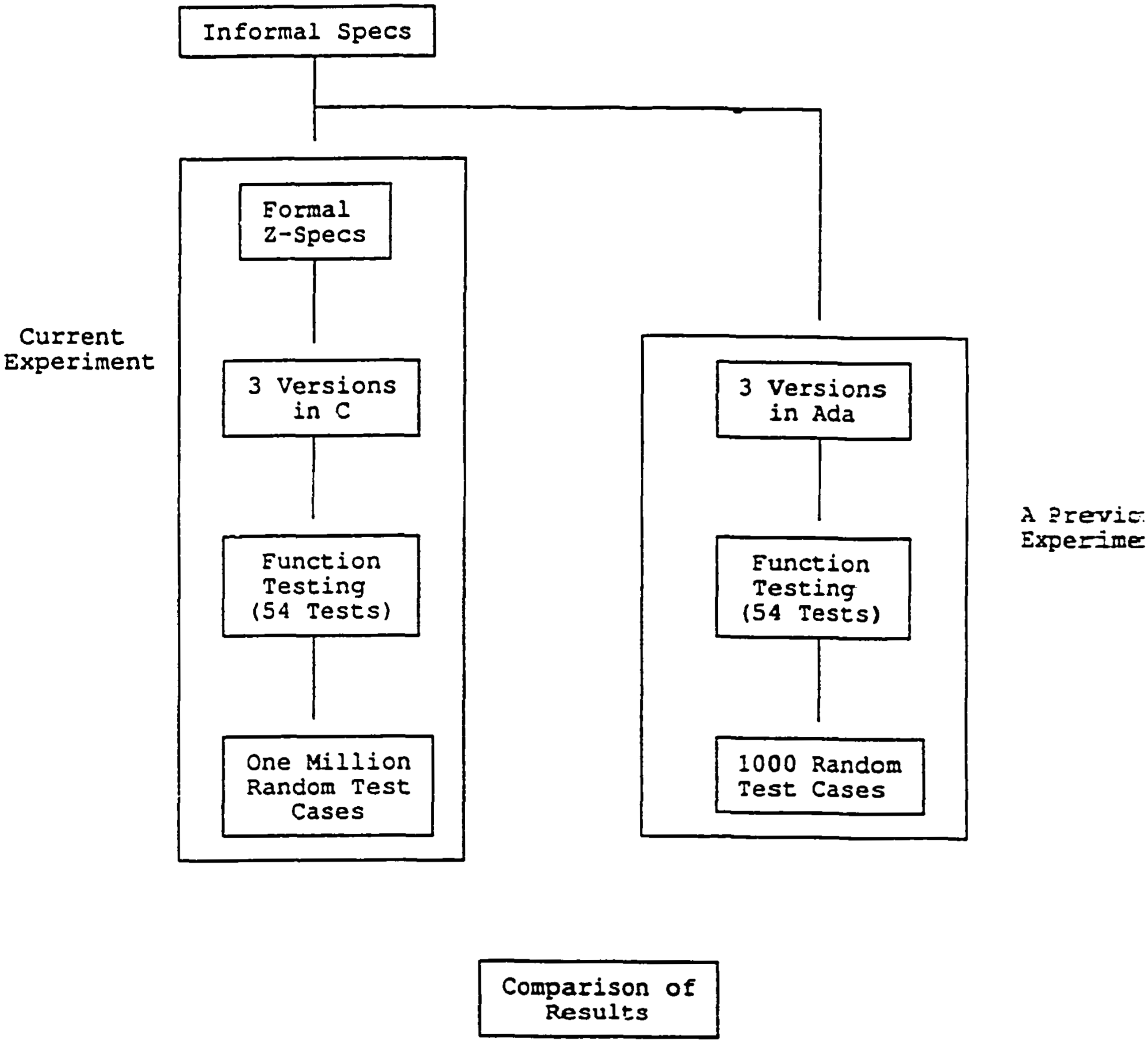
## OBJECTIVES OF STUDY

- **Investigate the effect of using formal specifications on**

  **- productivity**<br>
  **- reliability**<br>
  **- complexity**

- **Compare results with versions developed from informal specifications**

# EXPERIMENTAL APPROACH



Informal Specs

Current Experiment

Formal Z-Specs

3 Versions in C

Function Testing (54 Tests)

One Million Random Test Cases

A Previ
Experime

3 Versions in Ada

Function Testing (54 Tests)

1000 Random Test Cases

Comparison of Results

## EXPERIMENTAL APPRAOCH

- Used NASA - Launch Interceptor Problem (LIP)

- Developed Z-specifications from English specifications of LIP (Two independent Z specifications)

- Used Z-specs to develop 3 indipendent versions in C

- Each version tested for a set of 54 test cases from a previous experiment involving LIP

- Each version executed for one million random test cases to simulate operational testing

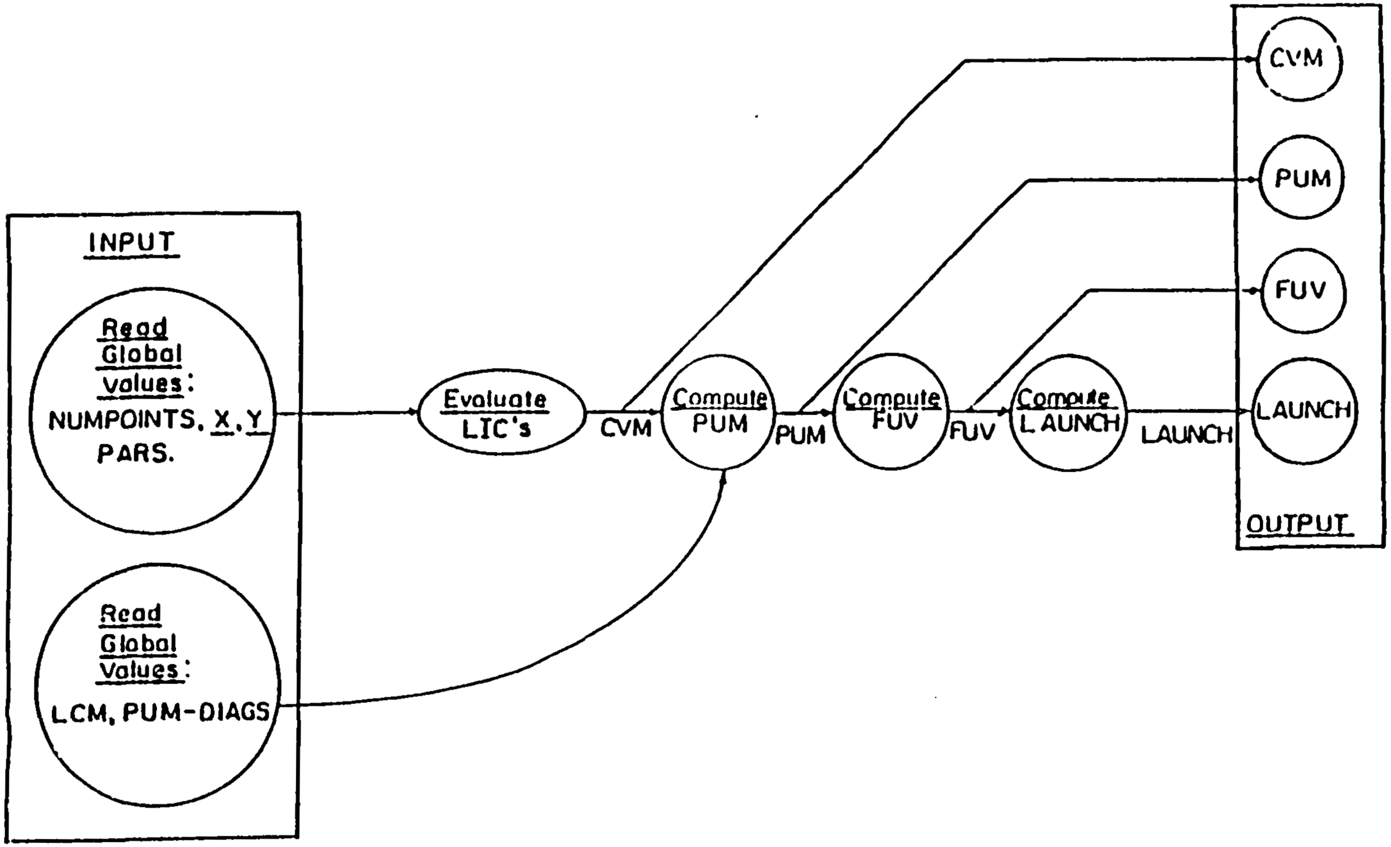
# LIP

- Simple, but realistic anti-missile system.

- Studied elsewhere* in connection with fault-tolerant and Fortran/Ada comparison software research

- Program reads inputs which represent radar reflections, checks whether some prespecified conditions are met and determines if the reflections come from an object that is a threat and if yes, signals a launch decision

* Knight and Leveson, IEEE-TSE, January 1986.
Goel, etal, COMPSAC 87 and RADC-TR-88-213.



A. Gad
Syracuse Univ.
Page 12 of 73

## EXAMPLE

### Launch Intercepter Conditions

LIC 1: There exists at least one set of two consecutive data points that are a distance greater than LENGTH 1 apart

$0 \leq = \text{LENGTH1}$

LIC 11: There exists at least one set of three data points separated by exactly E and F consecutive intervening points, respectively, that are the vertices of a triangle with area greater than AREA1

$1 \leq E$
$1 \leq F$
$E + F \leq \text{NUMPOINTS} - 3$

# Z-SPECIFICATION LANGUAGE

- Well known specification language developed by Programming Research Group at Oxford University

- Has been applied to develop specifications for several software systems but we are not aware of experimental results comparing it with informal approaches

# SOME COMMENTS ON Z FOR LIP

Z-specifications were helpful in several aspects.

Some Examples:

- In resolving certain ambiguous issues

  - whether two identical (x, y) pairs can belong to a sequence of input data points

- In expressing invariant properties

  - the LCM matrix is symmetric, can be easily expressed mathematically

- In exploiting the repetitiveness of certain launch conditions which was helpful in functional groupings for design and testing.

  - a closer look at LIC 1, 8 and 13 indicates that they are related. We exploit the similarity by defining a "prototype" schema, and then using it to define each of these separately

# Expressing Requirements in the Z Notation

## Example:LIC1

### Informal Specification

LIC 1: There exists at least one set of two consecutive data points that are a distance greater than LENGTH1 apart. (LENGTH1 ≥0)

### Formal Specification

$$LIC1[NUMPOINTS, LENGTH1]$$

$$POINTS : seq\ R \times R$$
$$cmv, cmv' : \mathcal{N} \rightarrow \mathcal{B}$$

$$cmv' = cmv \oplus$$
$$\{1 \mapsto (1 \leq \#\{(POINTS(i), POINTS(j)) | \forall i, j : 1..NUMPOINTS \bullet$$
$$j = i + 1 \wedge (LENGTH1 < edist(POINTS(i), POINTS(j)))\}$$
$$\wedge LENGTH1 \geq 0)\}$$

where $edist(p, q)$ computes the distance between points $p$ and $q$.

# Expressing Requirements in the Z Notation

## Example:LIC7

### Informal Specification

LIC 7: There exists at least one set of N_PTS consecutive data points such that at least one of the points lies a distance greater than DIST from the line joining the first and last of these points. If the first and last points of these N_PTS points are identical, then the calculated distance to compare with DIST will be the distance from the coincident point to all other points of the N_PTS consecutive points. (DIST ≥0)

### Formal Specification

$$\mathit{LIC7}[\mathit{NUMPOINTS}, N_P\mathit{TS}, \mathit{DIST}]$$

$$\begin{array}{l}
\mathit{POINTS} : \mathit{seq}\ R \times R \\
\mathit{cmv}, \mathit{cmv}' : \mathcal{N} \rightarrow \mathcal{B} \\
\hline
\mathit{cmv}' = \mathit{cmv} \oplus \\
\{7 \mapsto (1 \leq \#\{(\mathit{POINTS}(i), \mathit{POINTS}(j)) | \forall i, j : 1..\mathit{NUMPOINTS} \bullet \\
j = i + N\_PTS - 1 \land \exists k : i + 1..j - 1 \bullet \\
(\mathit{pt\_cmp}(\mathit{POINTS}(i), \mathit{POINTS}(j)) \\
\land (\mathit{edist}(\mathit{POINTS}(i), \mathit{POINTS}(k)) > \mathit{DIST})) \\
\lor (\neg \mathit{pt\_cmp}(\mathit{POINTS}(i), \mathit{POINTS}(j)) \\
\land (\mathit{pdist}(\mathit{POINTS}(i), \mathit{POINTS}(j), \mathit{POINTS}(k)) > \mathit{DIST}))\} \\
\land \mathit{DIST} \geq 0)\}
\end{array}$$

where *edist*$(p, q)$ computes the distance between points $p$ and $q$ , *pdist*$(p, q, r)$ computes the perpendicular distance from point $r$ to the line through $p$ and $q$ and *pt_cmp*$(p, q)$ returns a boolean value *true* if $p$ and $q$ are identical, and otherwise *false*

# Expressing Requirements in the Z Notation(contd.)

## Example:LIC7

$$\begin{aligned} &pdist : \mathcal{R}^2 \times \mathcal{R}^2 \times \mathcal{R}^2 \rightarrow \mathcal{R}^* \\ &\forall p, q, r \in \mathcal{R}^2 \bullet \\ &\neg(pt\_cmp(p, q)) \Rightarrow pdist(p, q) = \frac{2 \times \Delta area(p, q, r)}{edist(p, q)} \end{aligned}$$

Note that the line must be well defined, i.e, at least the points on the line must not be identical. Obviously this is a partial function.

# RESULTS OF EXPERIMENT

# SOME PROGRAM METRICS

| | C-Code From Z-Specs | | | Ada Code From Informal Specs | | |
|---|---|---|---|---|---|---|
| Programmer | A | B | C | D | E | F |
| Source Lines | 373 | 407 | 669 | 691 | 624 | 851 |
| Comment Lines | 82 | 80 | 59 | 59 | 126 | 251 |
| System Complexity* | 56 | 53 | 81 | 334 | 309 | 297 |

* See Lew et al, TSE, November 1988.

# COMPLEXITY METRIC*

- System S has n modules, each with complexity $M_i$

- System complexity = $\sqrt{\Sigma M_i}$

- $M_i$ depends on

  - Internal complexity
  - External complexity (measures module interrelationships)

- Internal complexity

  - McCabe's cyclomatic number

- External complexity

  - Amount of interaction with the environment
  - Depends on the information content of input and output parameters

* Lew et al, IEEE-TSE, November 1988, pp. 1645-1655.

## PROGRAM DEVELOPMENT EFFORT (hours)

| Versions | Develop Z-Specs | Design | Coding | Testing | Total |
|---|---|---|---|---|---|
| A | 27 | 6 | 6 | 6 | 45 |
| B | 10* | 10 | 10 | 8 | 38 |
| C | 33 | 8 | 6 | 4 | 51 |

* B used specs. developed by A

**Learning Z: A - 20 hrs.**
**C - 21 hrs.**

# NUMBER OF ERRORS*

| Programmer | Development and Unit Testing | Function Testing (54 TC) | "Operational" Testing (1 million TC) | Total |
|---|---|---|---|---|
| A | 3 | 0 | 0 | 3 |
| B | 1 | 7 | 0 | 8 |
| C | 3 | 0 | 0 | 3 |

*Does not include compilation errors

# COMPARISON OF DATA FROM C AND ADA VERSIONS

- **We compared the effort and error data from a previous experiment that used Fortran and Ada languages.**

- **We do not think that our results are biased because language dependent aspects are not under study here. Also, the programmers in these studies were reasonably proficient in the respective languages so that the choice of the language should not affect our results**

- **However, to enhance our conclusions, we plan to develop C versions from informal specifications**

# COMPARISON OF EXPERIMENTAL RESULTS: EFFORT AND ERRORS

| | | Z | | | Informal | | |
|---|---|---|---|---|---|---|---|
| Programmer | | A | B | C | D | E | F |
| Effort | | 45 | 32 | 51 | 76 | 73 | 89 |
| Errors | D&UT | 3 | 1 | 3 | 5 | 4 | 4 |
| | FT | 0 | 7 | 0 | 8 | 7 | 4 |
| | Total | 3 | 8 | 3 | 13 | 11 | 8 |

D&UT - Development and Unit Testing

FT - Function Testing

# FAULT AVOIDANCE BY USING Z

- We believe that certain types of faults can be avoided by using formal specifications

- Following are two explicit examples of faults avoided by using [illegible] for LIP*

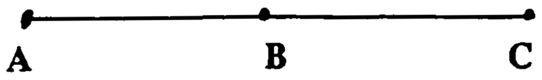
  - Caluclation of angle between π and 2π rather than between 0 and π

  - Calculation of distance from point to line when points are collinear and first point not between other two

* See Brilliant et al, TSE, February 1990, page 242.

# FAULT-AVOIDANCE - EXAMPLE LIC 7

- Consider 3 collinear points (A, B, C) as shown



- Need to compute distance from B to line AC (LIC 7)

- Computation* from informal specs can lead to

  Dist(A, C, B) = min (dist(A, B), dist(B,C))

- However, formal specifications always compute zero, the correct result

* See Brilliant et al, TSE, February 1990, p. 242.

# CONCLUDING REMARKS

- Use of Z specifications was clearly helpful in reducing errors (and hence increasing reliability)

- Based on a few metrics, it is also evident that the complexity of code developed from Z was also lower

- Total effort involved, including learning Z and development of Formal specifications, was comparable to the effort involved in developing versions from informal specifications

Yet —

- This experiment does not provide conclusive evidence about the superiority of formal specification over informal ones

- Further investigation necessary to explore the feasibility and usefulness of Z for large problems

- Reusability of such formal specifications also needs to be investigated

N92
19435
UNCLAS

P-35

# AN ANALYSIS OF DEFECT DENSITIES FOUND DURING SOFTWARE INSPECTIONS

**John C. Kelly, Ph.D, CQA**
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91101, USA

**Joseph S. Sherif, Ph.D.**
California State University, Fullerton
Fullerton, California 92634, USA

**Jonathan Hops**
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA

CC 356121

JJ 974050

## ABSTRACT

Software inspection is a technical evaluation process for finding and removing defects in requirements, design, code and tests. Software inspections have been used by a wide variety of organizations for improving software quality and productivity since their original introduction at IBM. The Jet Propulsion Laboratory (JPL), California Institute of Technology, tailored Fagan's original process of software inspections to conform to JPL software development environment in 1987. However, the fundamental rules of the Fagan inspection process were adhered-to.

Detailed data was collected during the first three years of experience at JPL on 203 inspections. Statistics are discussed for this set of inspections. Included, on a per inspection basis, are averages of: staff time expended, pages covered, major defects found, minor defects

found and inspection team size. The inspection team size varied from three to eight participants with the JPL Product Assurance Organization providing most of the moderators.

Analysis of variance (alpha = 0.05) showed a significantly higher density of defects during requirements inspections. It was also observed, that the defect densities found decreased exponentially as the work products approached the coding phase.

Increasing the pace of the inspection meeting decreased the density of defects found. This relationship was observed to hold for both major and minor defect densities, although it was more pronounced for minor defects.

This paper provides guidelines for conducting successful software inspections based upon three years of JPL experience. Readers interested in the practical and research implications of software inspections should find this paper helpful.

## INTRODUCTION

This paper describes an analysis of factors influencing the defect density of products undergoing software inspections. Software intensive projects at the Jet Propulsion Laboratory (JPL) require a high level of quality. JPL, a part of the California Institute of Technology, is funded by NASA to conduct its unmanned interplanetary space program. Software inspections were introduced at JPL in 1987 to improve software quality by detecting errors as early in the developmental lifecycle as possible.

Software Inspections are detailed technical reviews performed on intermediate engineering products. They are carried out by a small group of peers from organizations having a vested interest in the work product. The basic process is highly structural and consists of six consecutive steps: planning, overview, preparation, inspection meeting, rework and follow-up. The inspection process is controlled and monitored through metrics and checklists. One of the best fundamental descriptions of this process is Fagan's original article [Fagan, 1976].

JPL tailored Fagan's original process to improve the quality of the following technical products of a software intensive system: Software Requirements, Architectural Design, Detail Design, Source Code, Test Plans, and Test Procedures. For each of these types of products a checklist was tailored for JPL's application domain, standards and software development environment. Supplemental tailoring included the addition of a "third hour" step to Fagan's process. The "third hour" step was first discussed by Gilb [Gilb, 1987]. JPL's "third hour" step includes time for team members to discuss problem solutions and clear-up open issues raised in the inspection meeting. Other tailoring included substantial use of Software Product Assurance personnel as inspection moderators, a JPL specific training program, and new data collection forms.

The analysis of defect densities from inspections was performed for the purpose of 1) ensuring that the conditions of a quality inspection are being met by JPL inspections, 2) verifying previous research findings on inspections and 3) understanding the factors which influence inspection results. It was expected that the results would agree with previous findings on inspections, but due to slight variations in the variables collected, some differences were observed.

## Methods

Data was collected on 203 inspections performed on five software intensive projects at JPL. Practically all inspection team members were trained in a one and a half day course on formal inspections [Kelly, 1987]. Software Product Assurance supplied 70% of the moderators. The inspections took place between February 1987 and April 1990. Although the projects used Ada, C and Simula, only 16%

were performed on code. Table 1 shows the types of inspections performed in this study and the sample size for each type.

The data included in this study was recorded on the "Formal Inspection Detailed Report" and "Inspection Summary Report" forms [Appendix 1 and 2]. Each inspection produced a complete set of forms indicated in the process diagram [Figure 1]. This information was placed into a database and monitored. Occasionally, the chief moderator would contact the inspection moderator when reports were abnormal. This was done to provide feedback for inspections which were experiencing difficulties. Eleven inspection reports were rejected for analysis in this sample for the reason that they violated some of the fundamental rules of inspections as shown in [Appendix 3].

Checklists were used to 1) help inspection team members focus on what to look for in a technical work product, and 2) provide categories for classifying defects. A generic checklist was provided in the training materials for each type of inspection: R1, I0, I1, I2, IT1 and IT2. Projects may use the generic checklist or tailor this list to match their own environment and development standards. However, we encouraged projects to maintain the 15 main categories for types of defects shown in the "Formal Inspection Detailed Report". [Appendix 1].

The metrics used to monitor and analyze inspections can be classified into three prime areas: staff time, types of defects and workproduct characteristics. The staff time expended was recorded by total hours during each stage of the inspection process. Part way through our study we began collecting staff time by the role played in the inspection meeting (author, moderator, or inspector). The organizational areas, represented by these participants, were also recorded.

Each defect was classified by severity, checklist category, and "type". The severity of defect was classified either major, minor, or trivial.

Trivial defects in grammar and spelling were noted and corrected. but not included in this data analysis nor on the "Formal Inspection Detailed Report" (Appendix 1).

The "type" of defect (mission. wrong. or extra) was recorded on the forms. but not in the database. This information is not as institutionally critical. however. the authors find it to be a useful guide during the rework stage of the process.

The workproduct characteristics included size (by pages of lines of code). phase and type of product (requirements. test plan. etc.). and project Since inspections were usually introduced relatively early in the developmental lifecycle. when most products were technical documents. the preferred size reporting metric was in pages. A typical page of JPL documentation is single spaced. 38 lines per page in 10 point font size. A page containing a diagram was counted equal to a page of test. The authors felt that number of pages was a more accurate measure of material undergoing inspection than "estimated lines of code" for technical documents. since projects did not have a history of a detailed accounting of the second metrics during the early lifecycle phases. Due to most of the data being reported in pages. different relationships are found than in previous studies. It should be noted that "pages" is more of a producer oriented statistic than it is a product oriented measure. One of the key metrics that was used in this analysis is "density of defects per page". This metric was used to compare inspections of different types and their related factors.

## Results

Results showed a higher density of defects in earlier lifecycle products than in later ones. An analysis of variance was performed on data collected from the different types of inspections in the sample (R1. I0. I1. I2. IT1. and IT2). Figure 2 shows the average number of defects found per page for each of the inspection types. The analysis of variance test showed that at Alpha = 0.05. the defect density at the software requirements inspection (R1) is significantly higher than that of source code inspection (I2), and also the defect density at test plan inspection (IT1) is significantly higher than that at test procedures and function inspection (IT2). It was also observed that the defect densities found during inspections decreased exponentially as the development work products approach the coding phase [Figure 3].

The staff hours needed to fix defects were not found to be significantly different across the different phases of the lifecycle [Figure 4]. It should be noted that the defects found and fixed during these inspections originated during the lifecycle phase in which they were detected. Latent defects which were found in high level documents during inspections were recorded as "open issues" and submitted to the change control board. Since the researchers did not know the timely outcome from the control board, these potential defects are not tracked in this study. However, the average cost to fix defects during the inspection process (close to their origin) was 0.5 hours, which is considerably less than the range of 5 to 17 hours to fix defects during formal testing reported by a recent JPL project.

Previously, inspection defect counts were found to decrease as the amount of code to be inspected increased [Buck, 1981]. Figure 5 shows this trend to be sure for the total sample of inspections in this study with respect to defect density per page.

The average inspection team composition and size for this sample are shown in Figure 6 by type of inspection. For development inspection types (R1, I0, I1, and I2) the trend is for larger teams for requirements and high level documents while smaller teams are needed for code. The inspection program at JPL tried to insure that teams were comprised of members from organizations having a vested interest in the work product. The rationale for this was to keep inspections from being biased toward an organization's internal view of the product.

Figure 7 shows the distribution of defects percentage by defect types and defect categories.

## CONCLUSIONS

Experience has shown that formal inspection of software is a potent defect detection method; and thus, it enhances overall software quality by reducing rework during testing, as well as maintenance efforts.

The following items highlight the results of JPL experience with formal inspections.

1. A variety of different kinds of defects are found through inspections with Clarity, Logic, Completeness, Consistency, and Functionality being the most prevalent.

2. Increasing the number of pages to be inspected at a single inspection decreases the number of defects found.

3. Significantly more defects are found per page at the earlier phases of the software lifecycle. The highest defect density was observed during Requirements inspections.

4. The cost in staff hours to find and fix defects was consistently low across all types of inspections. On average it took 1.1 hours to find a defect and 0.5 hours to fix and check it (major and minor defects combined).

5. Larger team sizes (6 to 9) for higher level inspections (R1 and I0) are justified by data which showed an increased defect finding capability.

JPL has adopted formal inspections for many of its software intensive projects. The results are very encouraging and show very significant improvements in software quality.

## REFERENCES

[1] Buck, F. O., "Indications of Quality Inspections", IBM Technical Report, TR 21-802, September 1981.

[2] Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development", IBM Syst. J, vol 15, No. 3, pp 182-211, 1976.

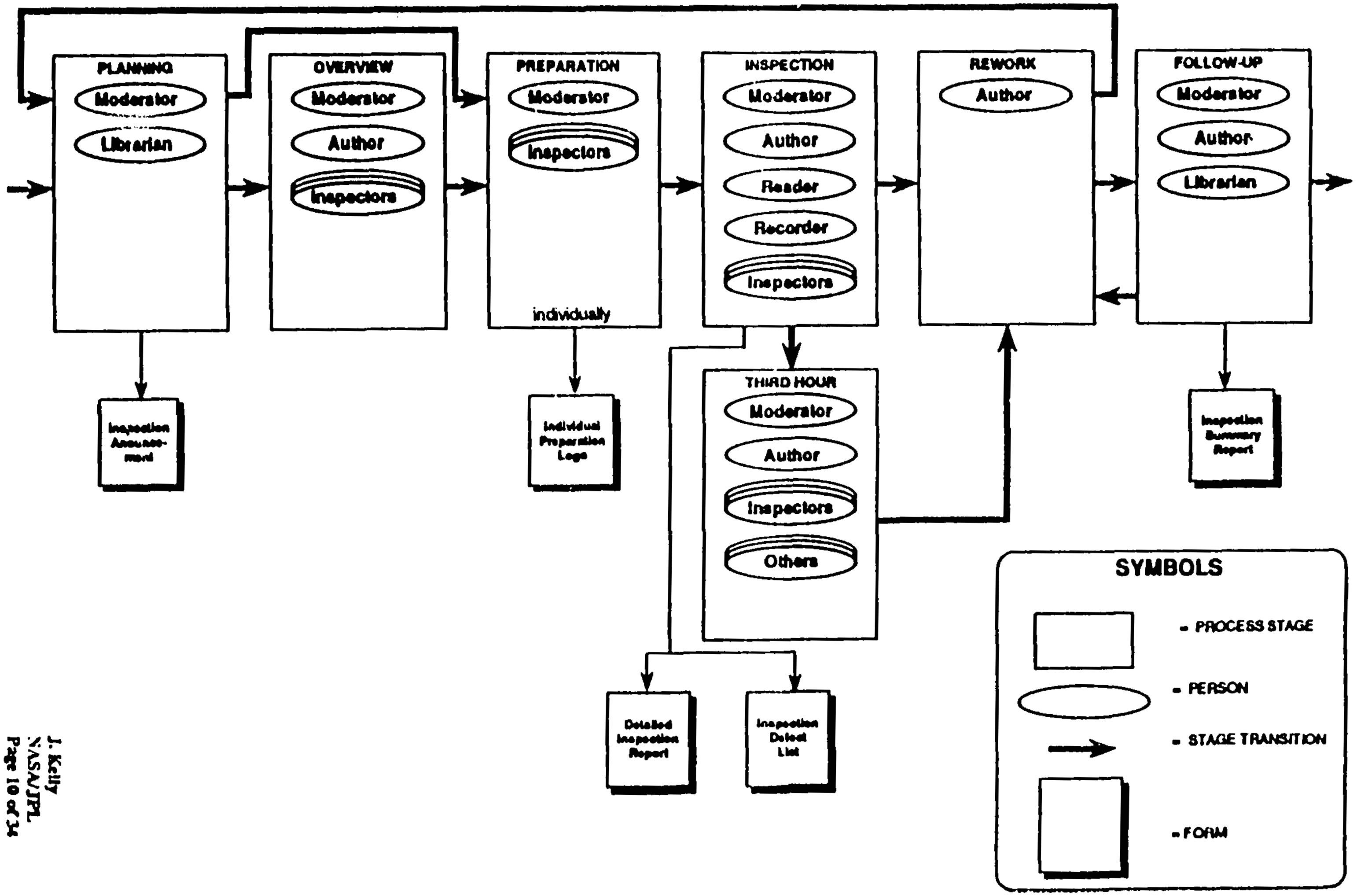
[3] Gilb, T., Principles of Software Engineering Management, Addison Wesley, Reading, MA, 1987.

[4] Kelly, J. C., "Formal Inspections For Software Development", Software Product Assurance Section, JPL, California Institute of Technology, Pasadena, CA, 1987.

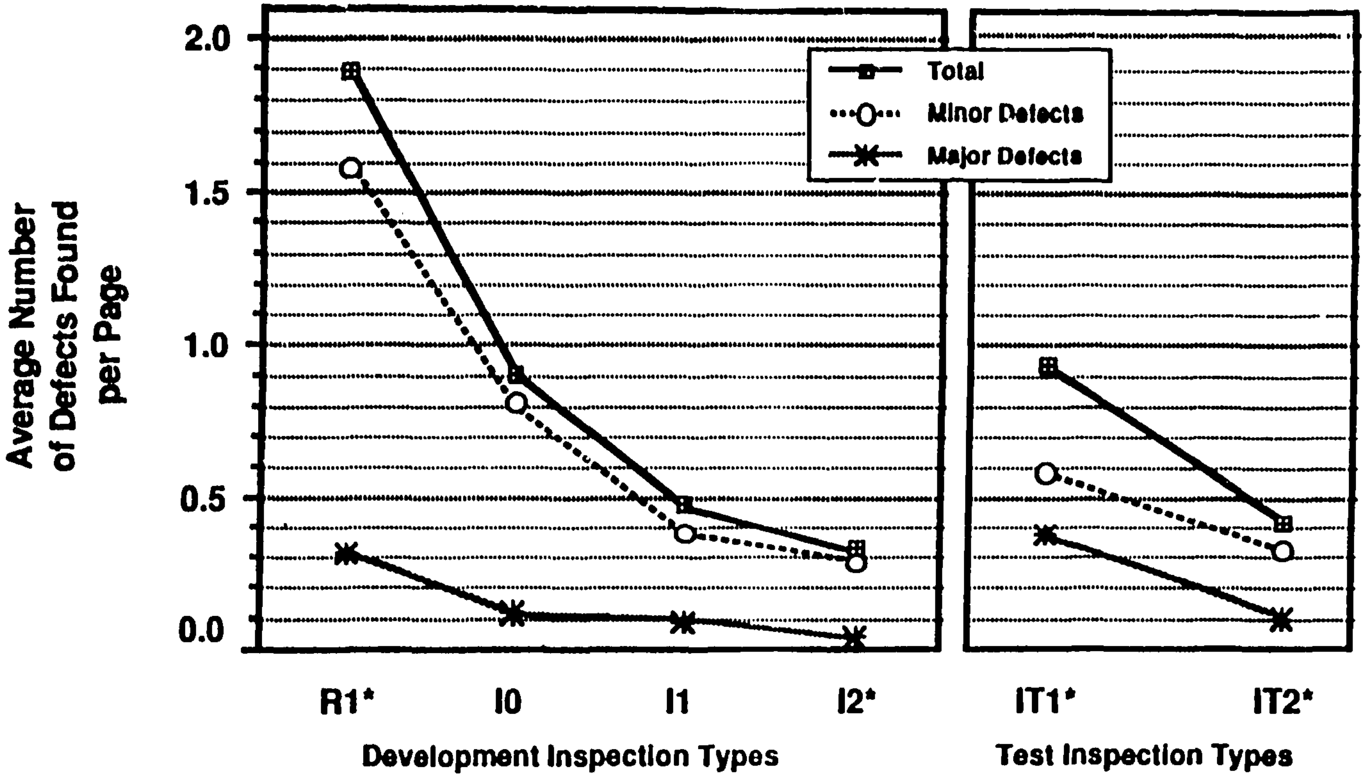
# Table 1: Types of inspections included in this analysis

| Inspection Abreviation | Inspection Type | Sample Size |
|---|---|---|
| R1 | Software Requirements Inspection | 23 |
| I0 | Architectural Design Inspection | 15 |
| I1 | Detailed Design Inspection | 92 |
| I2 | Source Code Inspection | 34 |
| IT1 | Test Plan Inspection | 16 |
| IT2 | Test Procedures & Functions Inspection | 23 |
|  | Total: | 203 |

# Figure 1: Overview of the Software Inspection Process

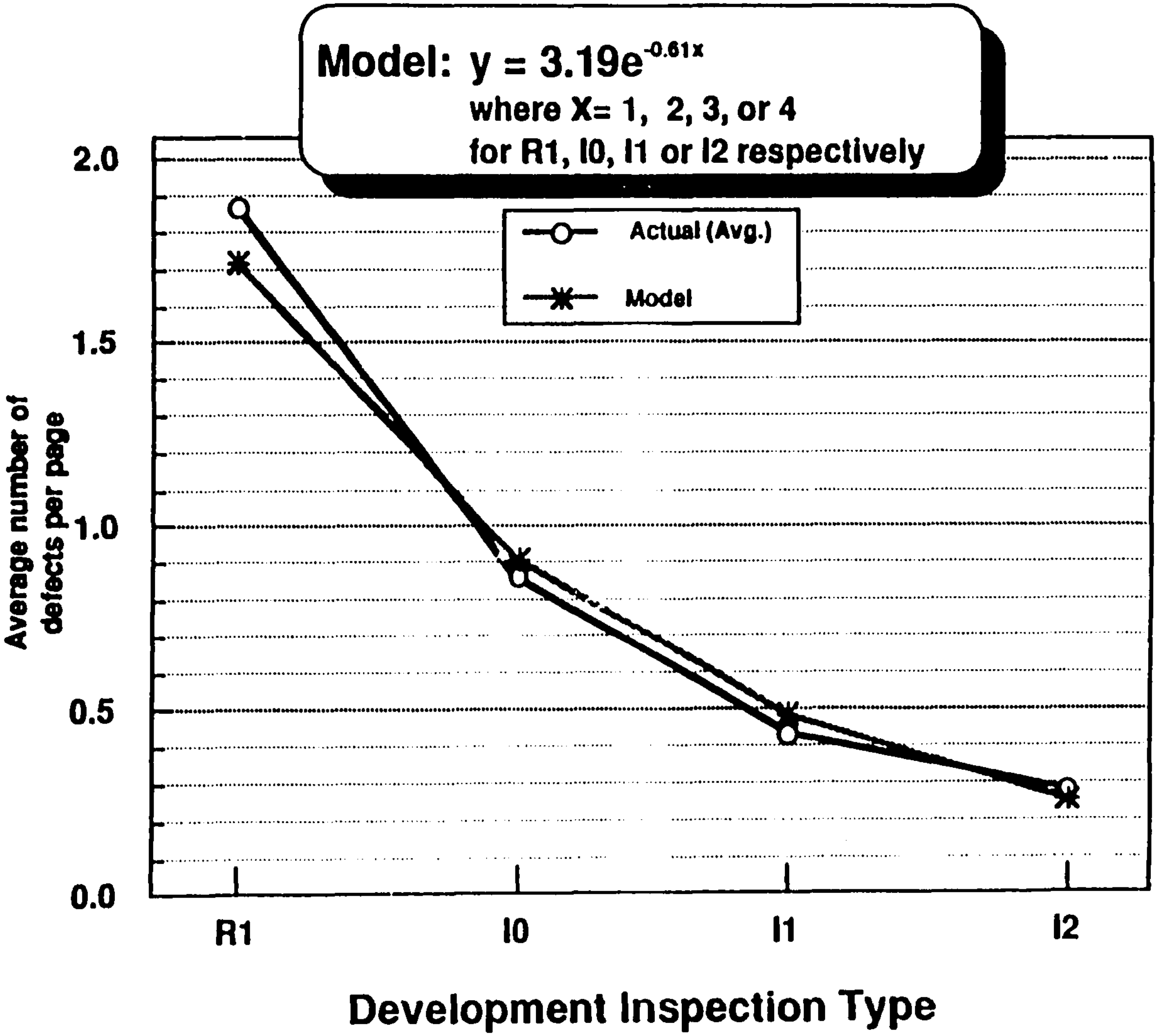


# Figure 2: Defect Density Versus Inspection Types



** At the alpha= 0.05 level of significance ANOVA F test showed a significant difference between the defect densities of R1 and I2, and between IT1 and IT2.*

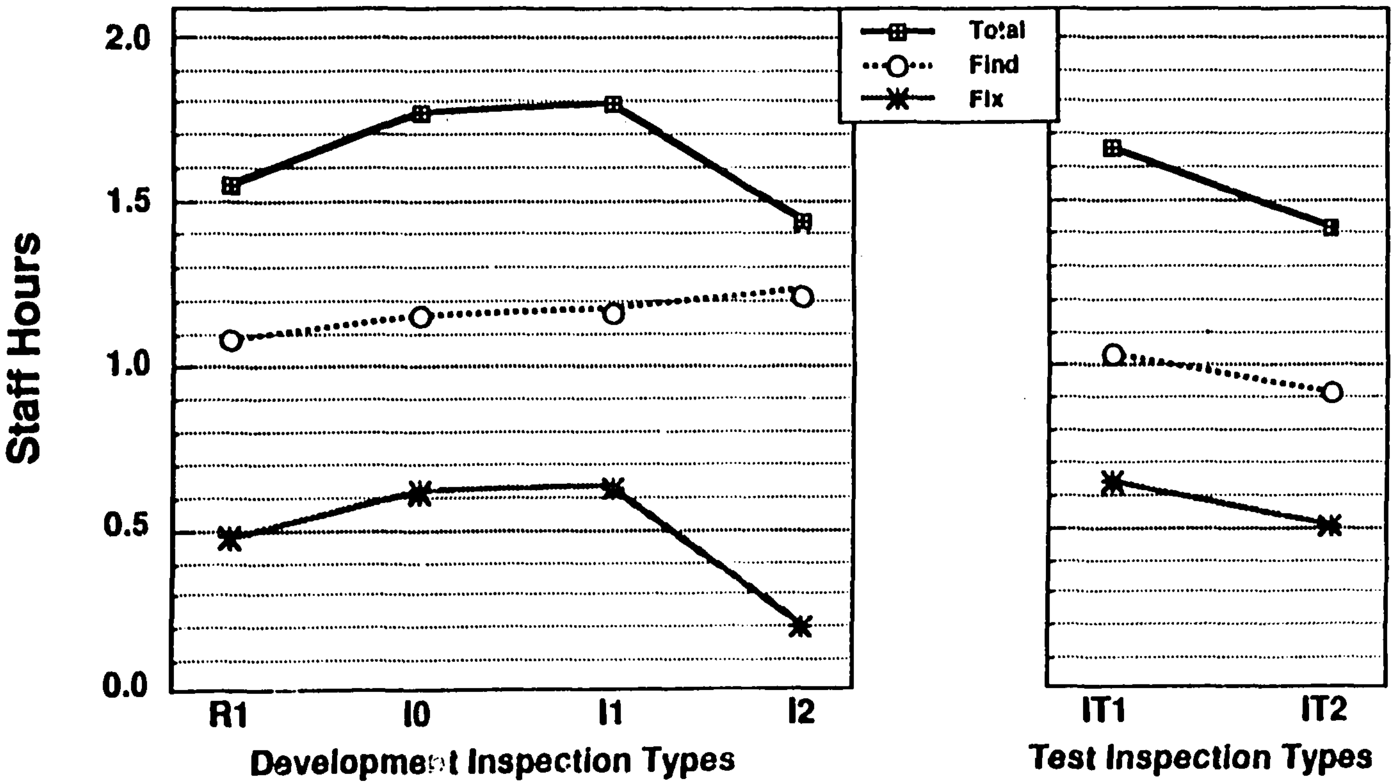
# Figure 3. A developed predictive model for defect density as a function of inspection type



Model: $y = 3.19e^{-0.61x}$
where X= 1, 2, 3, or 4
for R1, I0, I1 or I2 respectively

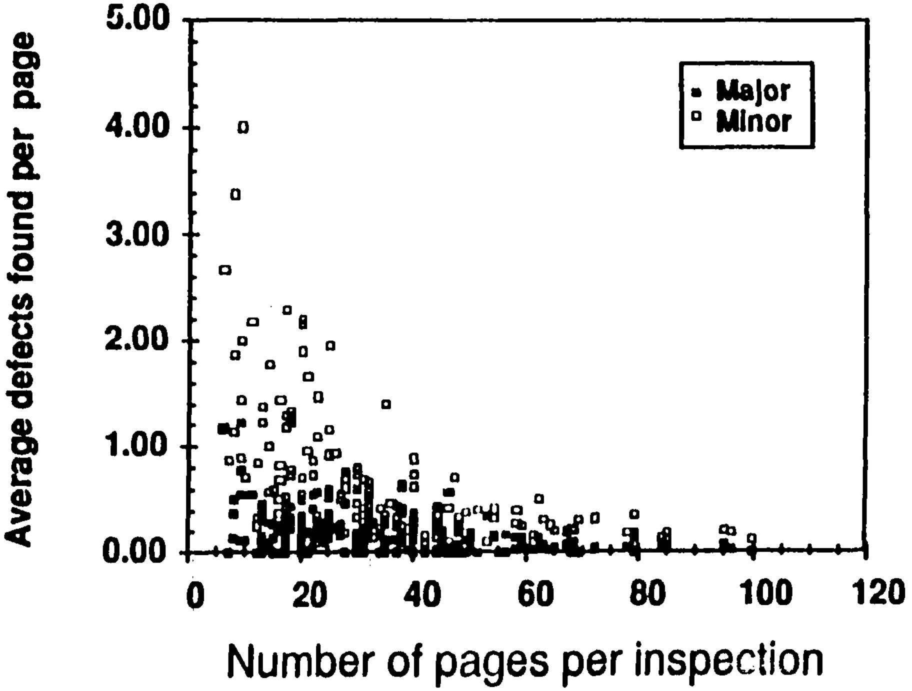
J. Kelly
NASA/JPL
Page 12 of 34

## Figure 4: Staff hours per defect.



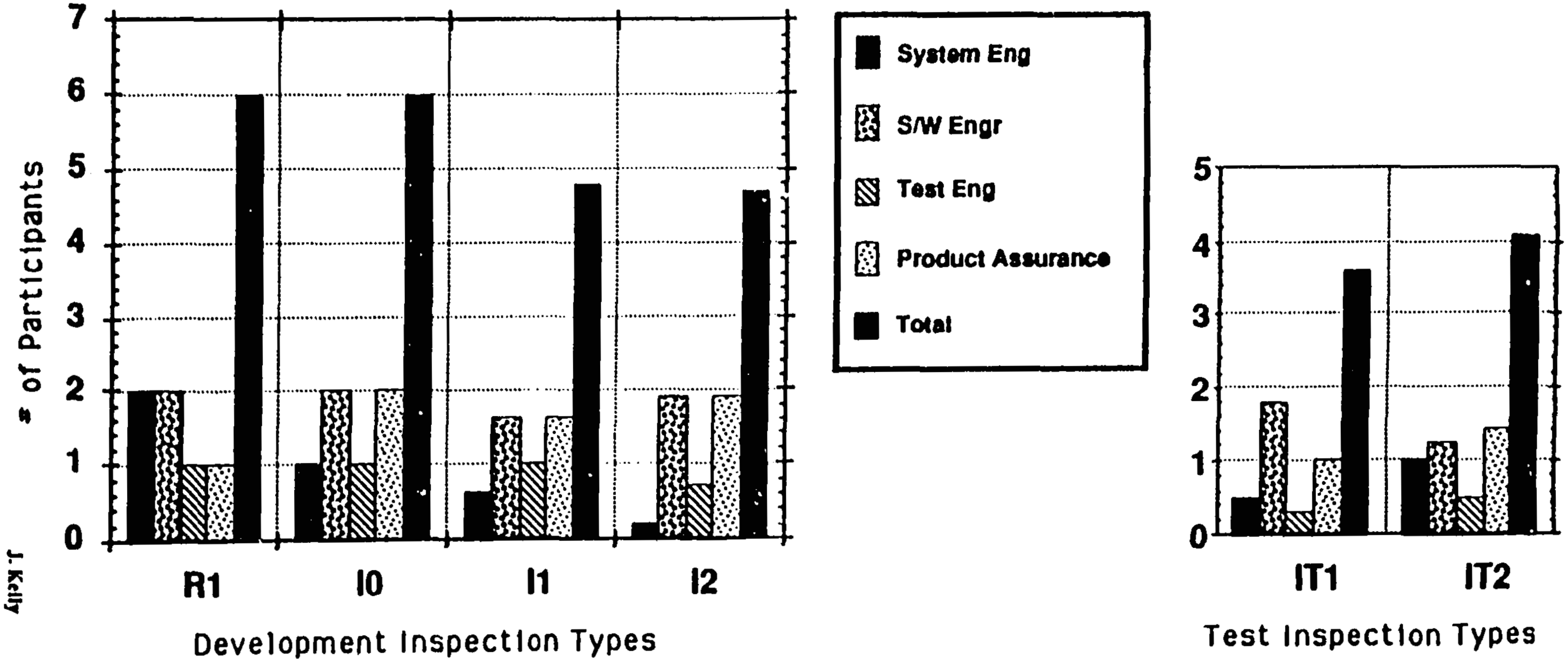
Resource hours for *finding* include all tim͏e expended during Planning, Overview, Preparation, and Meeting phases. Resource hours for *fixing* include all time expended during Rework, Third Hour, and Follow-up phases. Defects include all major and minor defects.

# Figure 5. Inspection page rate versus average defects found per page



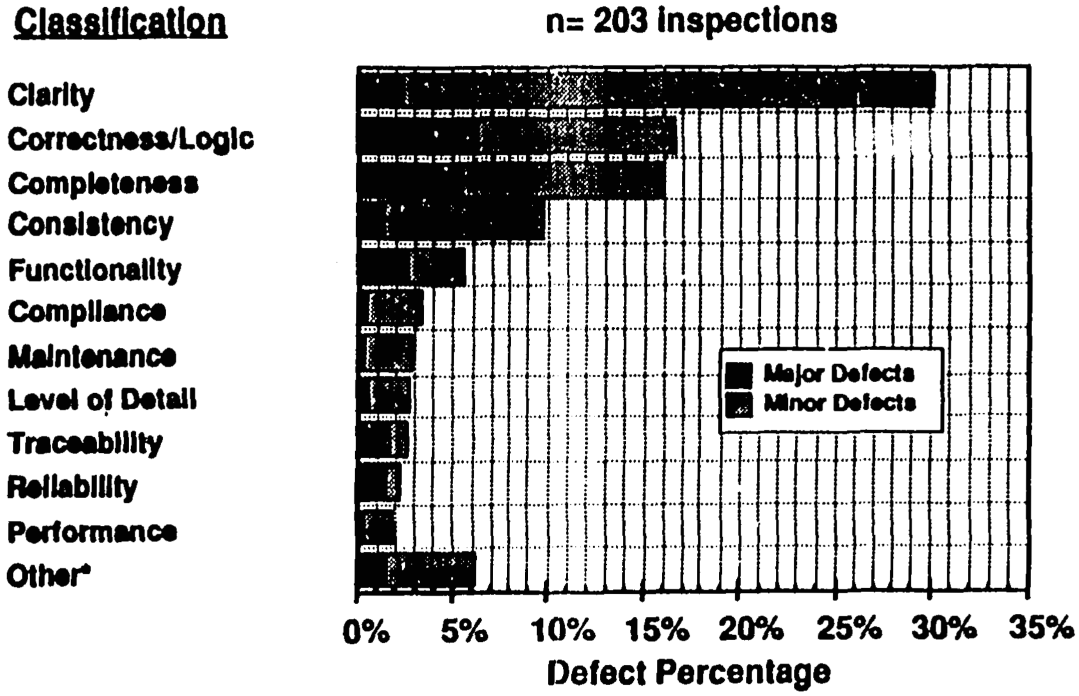
Note: Inspection "meetings" are limited to 2 hours and moderators are recommended to limit material covered to 40 pages or less.

# Figure 6: Team Composition and Size by Inspection Type

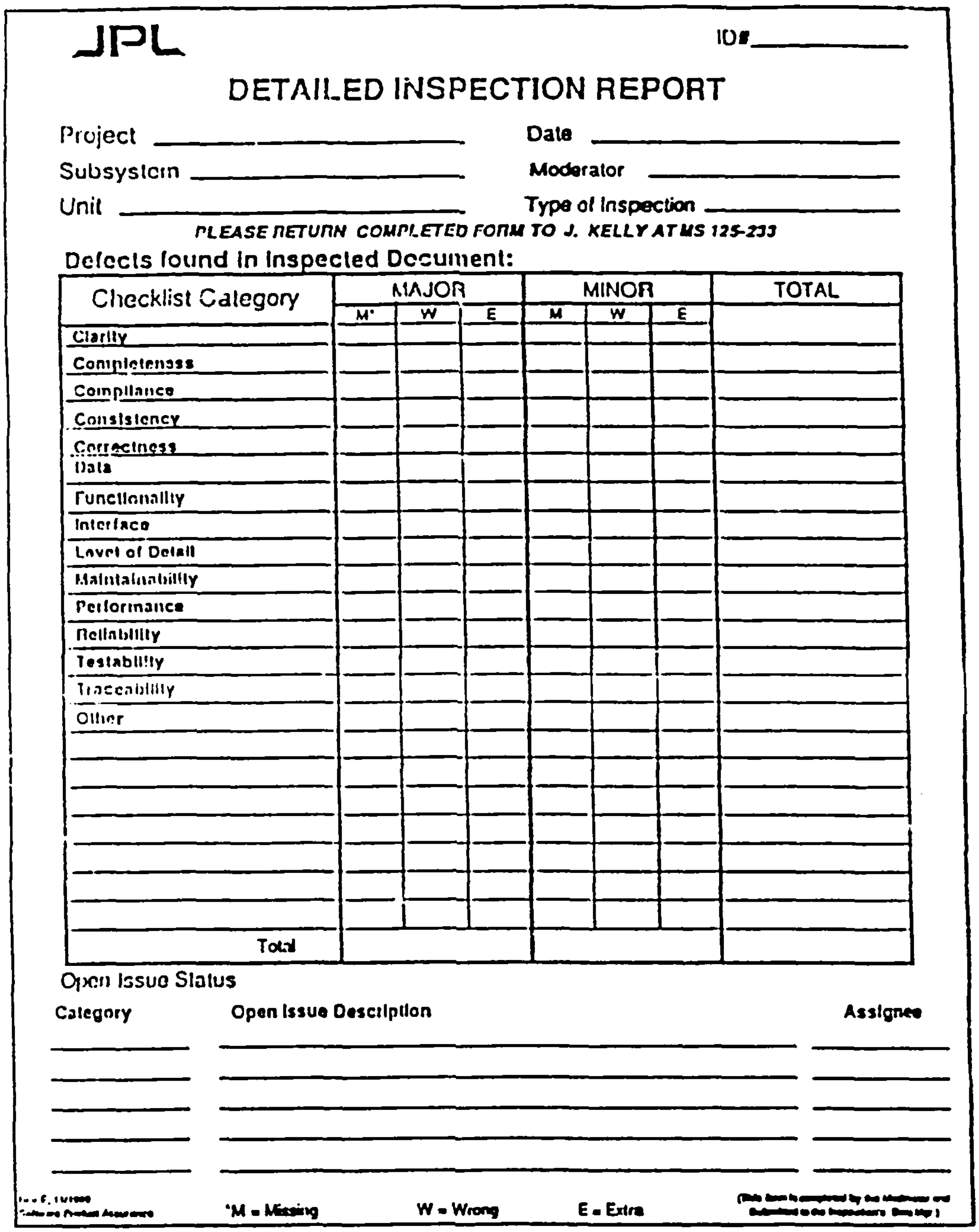


J. Kelly
NASA/JPL
Page 15 of 34

# Figure 7: Distribution of defects by classification



* "Other" includes those classifications with fewer than 20 total defects.



JPL

ID# ____________

## DETAILED INSPECTION REPORT

Project ____________________ Date ____________________

Subsystem ____________________ Moderator ____________________

Unit ____________________ Type of Inspection ____________________

*PLEASE RETURN COMPLETED FORM TO J. KELLY AT MS 125-233*

Defects found in Inspected Document:

| Checklist Category | MAJOR | | | MINOR | | | TOTAL |
|---|---|---|---|---|---|---|---|
| | M* | W | E | M | W | E | |
| Clarity | | | | | | | |
| Completeness | | | | | | | |
| Compliance | | | | | | | |
| Consistency | | | | | | | |
| Correctness | | | | | | | |
| Data | | | | | | | |
| Functionality | | | | | | | |
| Interface | | | | | | | |
| Level of Detail | | | | | | | |
| Maintainability | | | | | | | |
| Performance | | | | | | | |
| Reliability | | | | | | | |
| Testability | | | | | | | |
| Traceability | | | | | | | |
| Other | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| Total | | | | | | | |

Open Issue Status

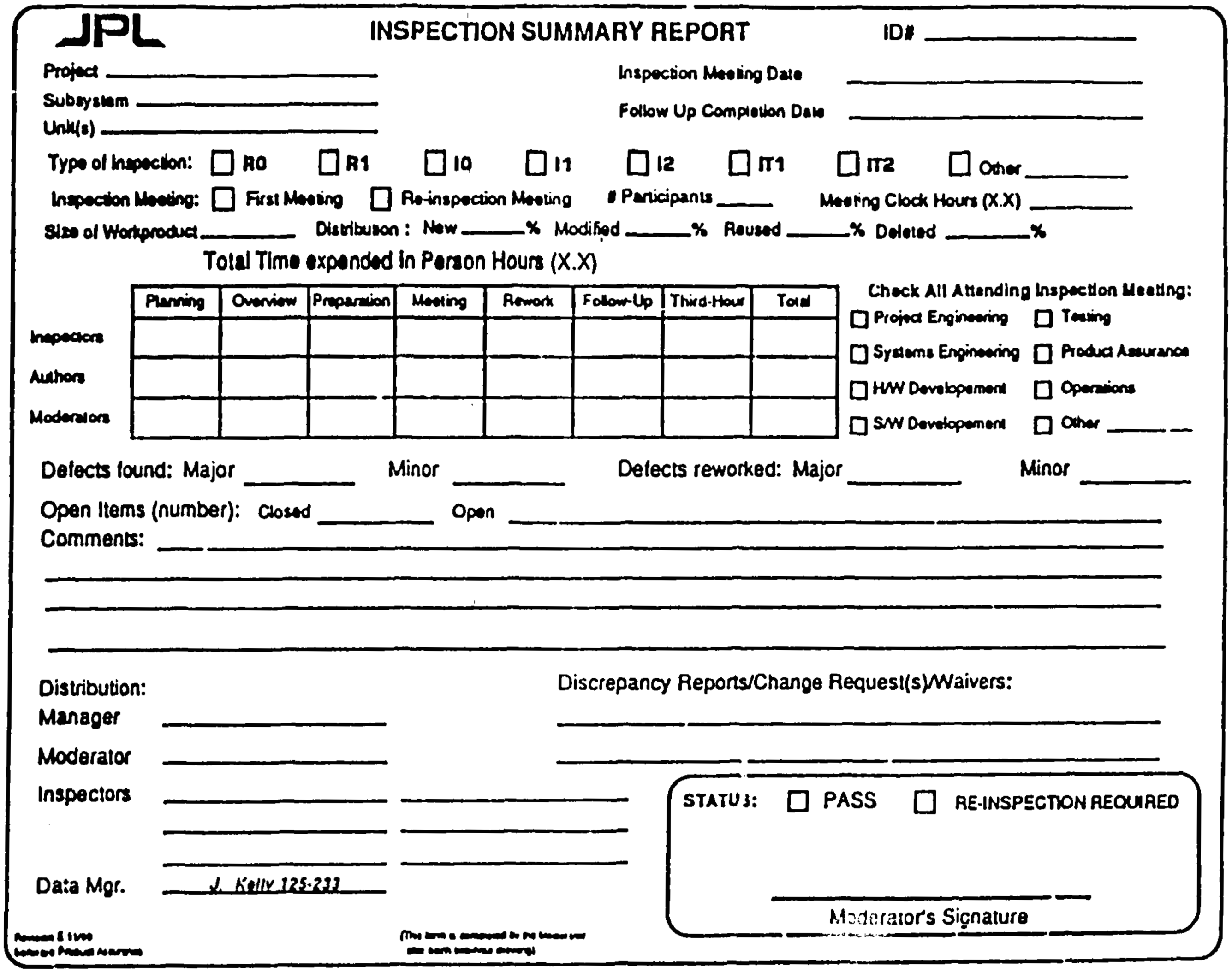
| Category | Open Issue Description | Assignee |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |

[illegible]
Software Product Assurance

*M = Missing W = Wrong E = Extra

([illegible])

Appendix 1: Formal Inspection Detailed Report



JPL

INSPECTION SUMMARY REPORT ID# ____________

Project ____________ Inspection Meeting Date ____________

Subsystem ____________ Follow Up Completion Date ____________

Unit(s) ____________

Type of Inspection: □ R0 □ R1 □ I0 □ I1 □ I2 □ IT1 □ IT2 □ Other ____________

Inspection Meeting: □ First Meeting □ Re-inspection Meeting # Participants ______ Meeting Clock Hours (X.X) ______

Size of Workproduct ______ Distribuion : New ______% Modified ______% Reused ______% Deleted ______%

Total Time expended in Person Hours (X.X)

| | Planning | Overview | Preparation | Meeting | Rework | Follow-Up | Third-Hour | Total |
|---|---|---|---|---|---|---|---|---|
| Inspectors | | | | | | | | |
| Authors | | | | | | | | |
| Moderators | | | | | | | | |

Check All Attending Inspection Meeting:

□ Project Engineering □ Testing

□ Systems Engineering □ Product Assurance

□ H/W Developement □ Operations

□ S/W Developement □ Other ______

Defects found: Major ______ Minor ______ Defects reworked: Major ______ Minor ______

Open Items (number): Closed ______ Open ____________

Comments: ____________

Distribution: Discrepancy Reports/Change Request(s)/Waivers:

Manager ____________

Moderator ____________

Inspectors ____________

Data Mgr. *J. Kelly 125-233*

STATUS: □ PASS □ RE-INSPECTION REQUIRED

Moderator's Signature

[illegible]

[illegible]

Appendix 2: Inspection Summary Report

# Appendix 3
# The 10 basic Rules of Inspections:

1. Inspections are carried out at a number of points inside designated phases of the software lifecycle. Inspections are not substitutes for major milestone reviews

2. Inspections are carried out by peers representing the areas of the life cycle affected by the material being inspected (usually limited to 6 or less people). Everyone participating should have a vested interest in the work product.

3. Management is not present during inspections. Inspections are not to be used as a tool to evaluate workers.

4. Inspections are led by a trained moderator.

5. Trained inspectors are assigned specific roles.

# Appendix 3
# The 10 basic Rules of Inspections:

(Continued)

6. Inspections are carried out in a prescribed series of steps (as shown in figure 1).

7. Inspection meetings are limited to two hours.

8. Checklists of questions are used to define the task and to stimulate defect finding.

9. Material is covered during the Inspection meeting within an optimal page rate range which has been found to give maximum error finding ability.

10. Statistics on the number of defects, the types of defects and the time expended by engineers on the inspections are kept.

# VIEWGRAPH MATERIALS

# FOR THE

# J. KELLY PRESENTATION

JPL

# An Analysis of Defect Densities Found During Software Inspections

Fifteenth Annual Software Engineering Workshop
November 28-29, 1990
Goddard Space Flight Center,
Greenbelt, Maryland

John C. Kelly, Ph.D.
&
Joseph Sherif, Ph.D.

Section 522
Software Product Assurance
Jet Propulsion Laboratory
Pasadena, California

# What are Software Inspections?

Software Inspections are:

- Detailed Technical Reviews
- Performed on intermediate engineering products
- A highly structured and well defined process
- Carried out by a small group of peers from organizations having a vested interest in the work product
- Controlled and monitored through metrics and checklist

JPL

# Types of Software Inspections Included in this Analysis

| | SAMPLE SIZE |
|---|---|
| ● R1 Software Requirements Inspection | 23 |
| ● I0 Architectural Design Inspection | 15 |
| ● I1 Detailed Design Inspection | 92 |
| ● I2 Source Code Inspection | 34 |
| ● IT1 Test.Plan Inspection | 16 |
| ● IT2 Test Procedures & Functions Inspection | 23 |
| TOTAL | 203 |

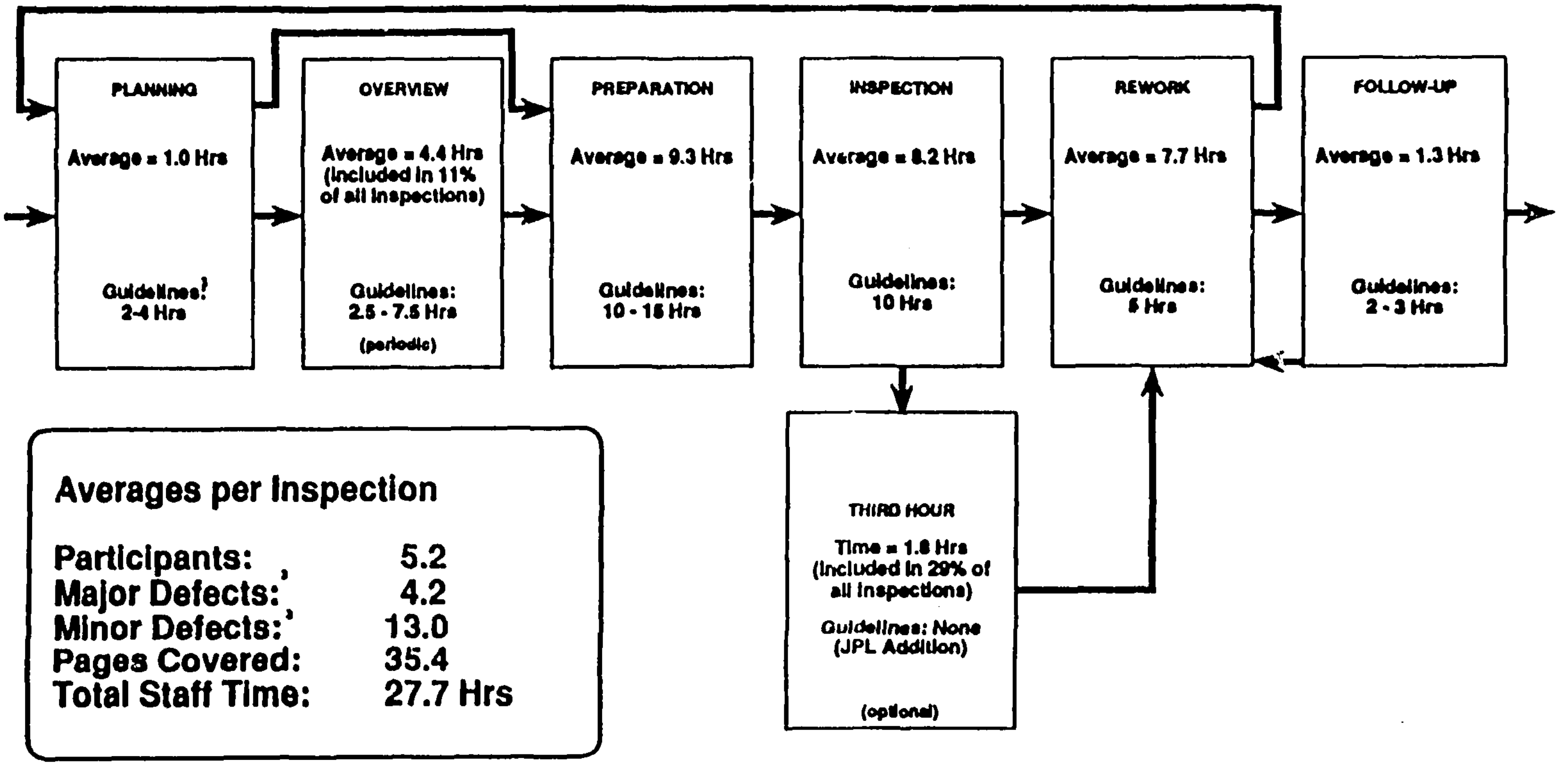
JPL

# JPL Tailoring of Existing Inspection Techniques

- Participants and Team Composition
- Third Hour
- Training
- Support Documentation

JPL

# Software Inspection Data Summary[1]



## Averages per Inspection

| | |
|---|---|
| Participants: | 5.2 |
| Major Defects:[3] | 4.2 |
| Minor Defects:[3] | 13.0 |
| Pages Covered: | 35.4 |
| Total Staff Time: | 27.7 Hrs |

[1] All times are averages from a sample of 203 JPL inspections.

[2] Guidelines were set in 2/88 based on outside organizations' experience and a team of five inspectors.

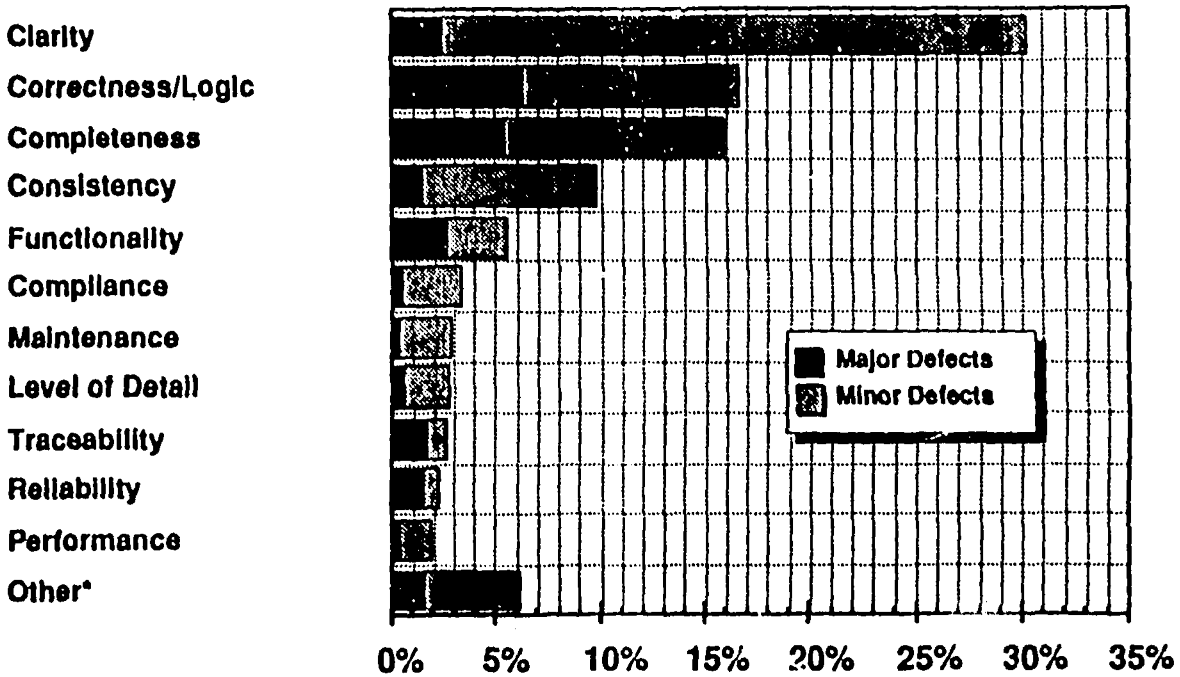
[3] A major defect is an error that would cause the system to fail during operations, or prevent the system from fulfilling a requirement. Minor defects are all other defects which are non-trivial. Trivial defects in grammar and spelling were noted and corrected, but not included in this data analysis.

J. Kelly
NASA/JPL
Page 25 of 34

JPL

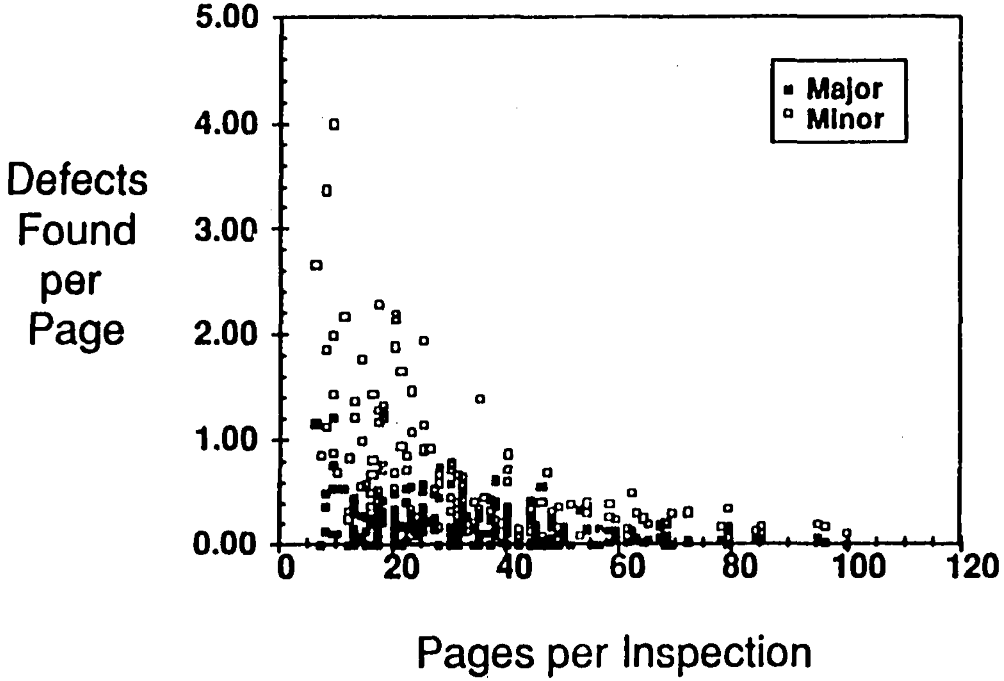
# Distribution of Defects By Classification

n= 203 inspections



* "Other" includes those classifications with fewer than 20 total defects.

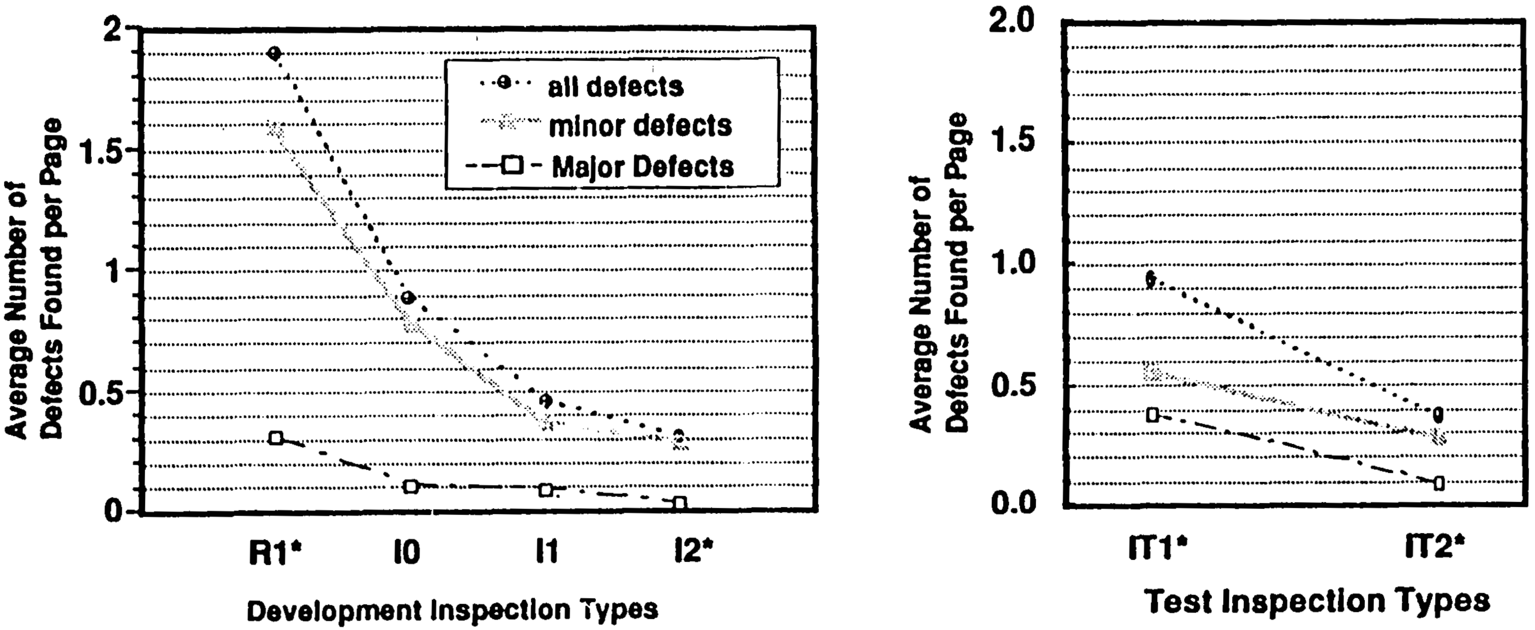
# Inspection Page Rate vs. Defect Density



Note: Inspection "meetings" are limited to 2 hours and moderators are recommended to limit material covered to 40 pages or less.

JPL

# Defect Density vs. Inspection Type

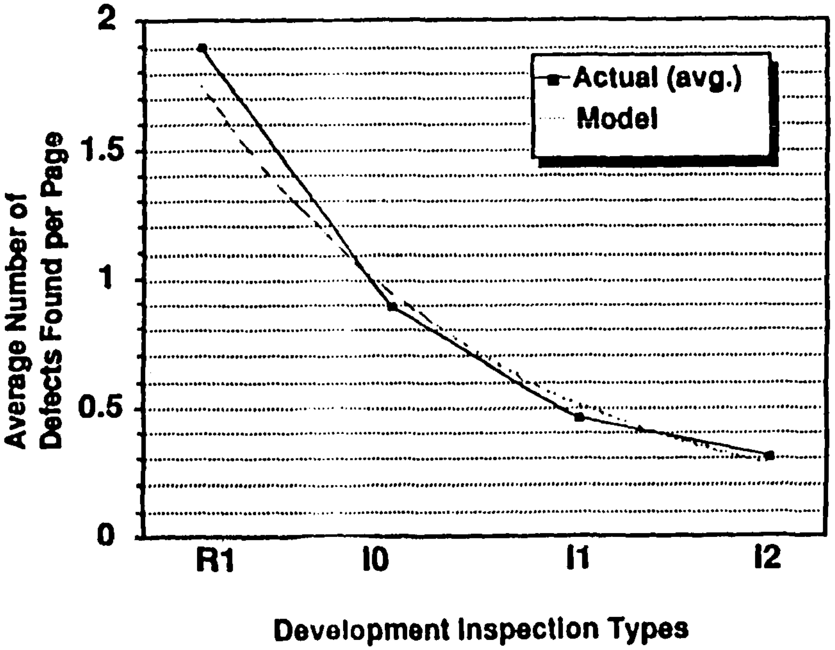


** At the alpha=0.05 level of significance ANOVA F test showed a significant difference between the defect densities of R1 and I2, and between IT1 and IT2.*

JPL

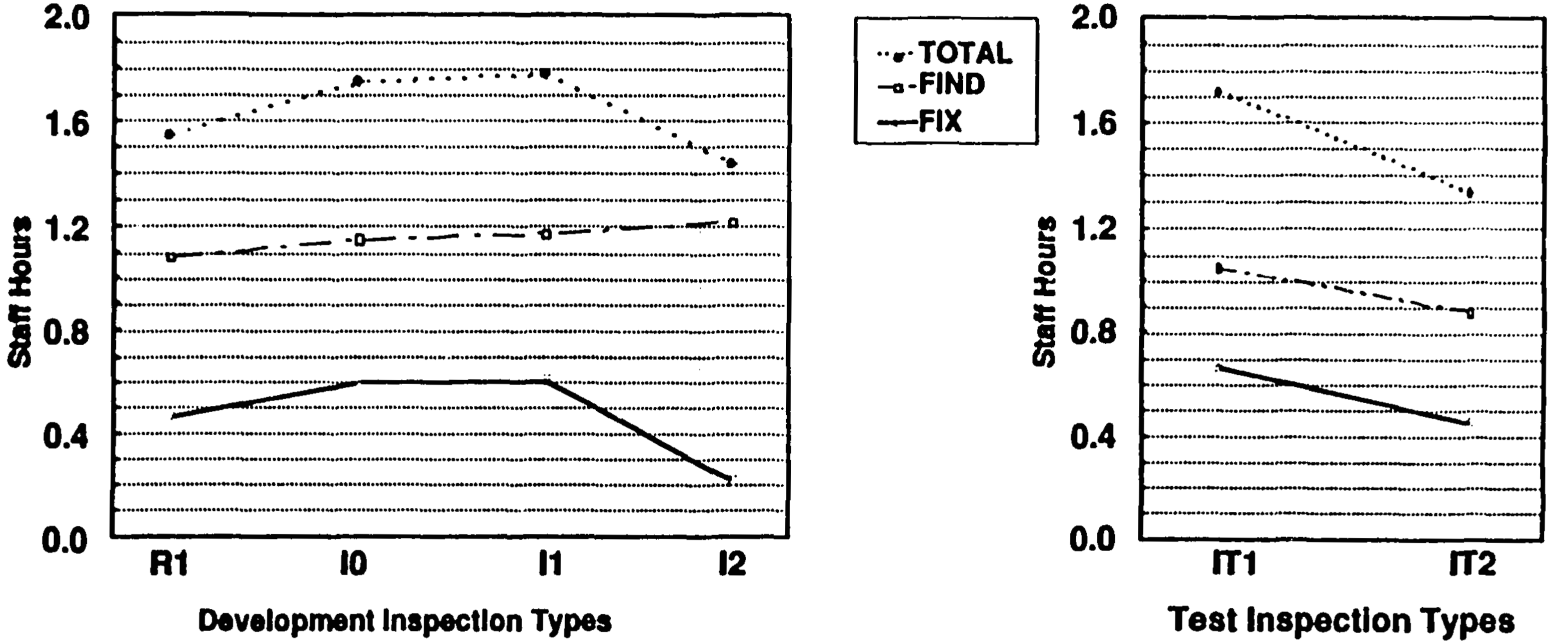
# Predictive Model for Defect Density vs. Inspection Type

*Model :* $y = 3.19 \exp(-0.61x)$
*where x= 1,2,3, or 4*
*for R1, I0, I1, or I2 respectively*



JPL

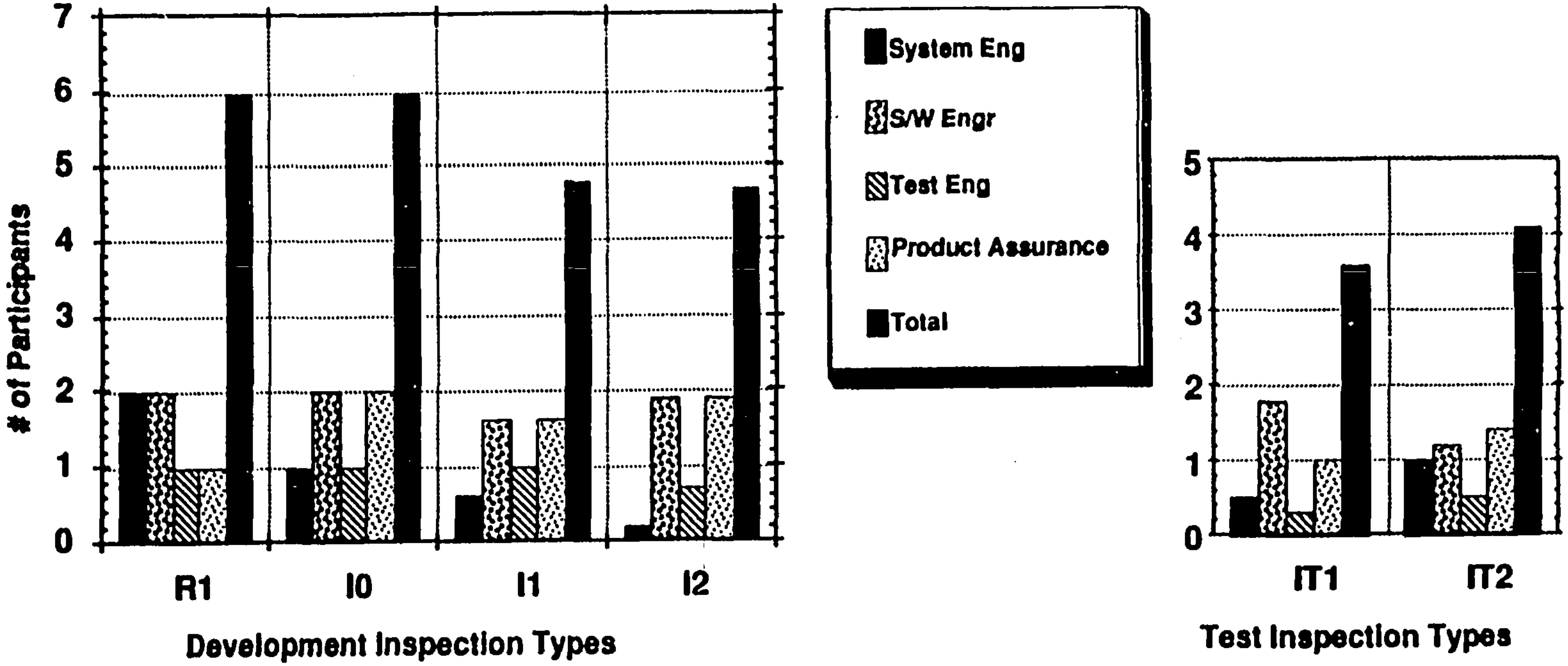
# Resource Hours per Defect



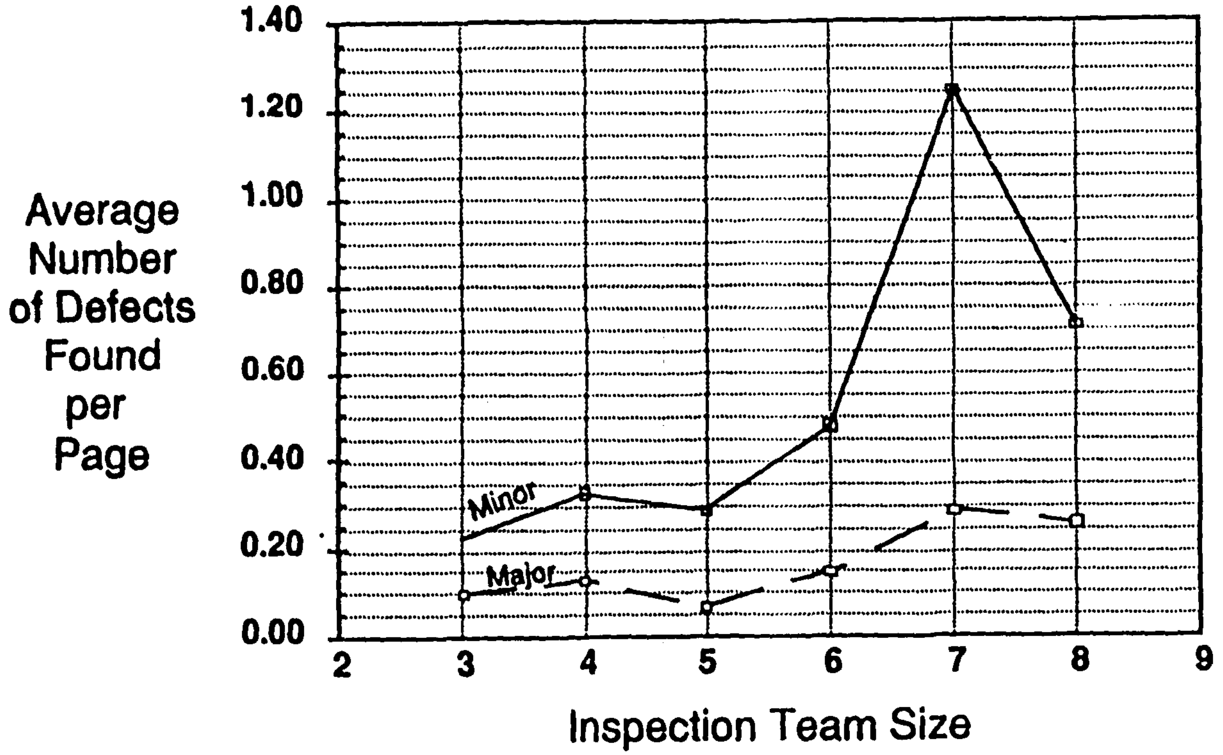
***In contrast, recent JPL projects reported spending an average of 5 to 17 staff hours to <u>fix</u> each defect during the <u>test phases</u>.***

Resource hours for *finding* include all time expended during Planning, Overview, Preparation, and Meeting phases. Resource hours for *fixing* include all time expended during Rework, Third Hour, and Follow-up phases. Defects include all major and minor defects.

# Team Composition and Size by Inspection Type



# Team Size vs. Defect Density



Note: I1 inspections are the most frequent for team sizes 3, 4, 5, & 6
R1 inspections are the most frequent for team sizes 7 & 8

# Code Inspections vs. Code Audits

| | Avg. Number of Defects Found per Page | | |
|---|---|---|---|
| | Major | Minor | Sample Size |
| Code Inspections | 0.022 | 0.250 | 34 |
| Code Audits | 0.007 | 0.111 | 15 |

Note: 1. The work product history for code inspection sample was: 41% new, 55% reused, and 4% modified. The work product history for code audits sample was: 100% new.

2. For all types of inspections combined the average number of defects found per page was much higher than what was found in code inspections (refer to slide # 8). The overall averages were; Major = 0.119 and Minor = 0.377, for a sample size of 203 inspections.

JPL

# CONCLUSIONS

- A variety of different kinds of defects are found through inspections with Clarity, Logic, Completeness, Consistency, and Functionality being the most prevalent.

- Increasing the number of pages to be inspected at a single inspection decreases the number of defects found.

- Significantly more defects were found per page at the earlier phases of the software lifecycle. *The highest defect density was observed during Requirements inspections.*

- The cost in staff hours to find and fix defects was consistently low across all types of inspections. On average it took 1.1 hours to find a defect and 0.5 hours to fix and check it (major & minor defects combined).

- Larger team sizes (6 to 8 engineers) for higher level inspections (R1 & I0) are justified by data which showed an increased defect finding capability.

- Code Inspections were superior in finding defects over Code Audits (single reviewer) by a factor of 3.

## VIEWGRAPHS FOR PANEL 1

6289-0

# VIEWGRAPH MATERIALS

# FOR

# MICHAEL DASKALANTONAKIS,
# Motorola

# EXPERIENCES IN ESTABLISHING AND MAINTAINING AN EFFECTIVE MEASUREMENT PROGRAM

by

Michael K. Daskalantonakis

Software Research and Development
Motorola, Inc.

November 29, 1990

# THE SOFTWARE METRICS PROGRAM IN MOTOROLA

- Corporate–wide activities since 1988 (parts of the company have started earlier than that)
- **Emphasis on building an organizational infrastructure:**
  - Metrics Working Group (MWG)
  - Metrics Users Group (MUG)
  - Metrics champions within Corporate R&D and within business units (metrics entities)
  - Two–day training workshop on software metrics
  - Consulting on the use of software metrics for process improvement
- A set of software metrics has been defined and it is **required** by the Quality Policy for Software Development
- Additional metrics are used by projects as necessary
- **The goal is not measurement. The goal is improvement through measurement, analysis, and feedback**

# OBSTACLES WE HAD TO OVERCOME

- Setting up a "**system**" to capture the data; initial **lack of tools that automate metrics;** some projects wanted to use metrics, but did not have the tools

- Getting people to track the data; **fear that other projects do not report the data consistently**

- **Middle–level management** resistance in implementing the metrics; **fear of overhead and extra cost; perception that the data may be used against them**

- Required a **cultural change** in the software community: it met resistance, although Motorola has already implemented measurement systems in other areas, and change is part of the everyday business

# COST OF THE METRICS PROGRAM

- **Cost is involved in terms of establishing the program, as well as in terms of operational costs**

- The MWG meets twice a quarter (about 8 people present)

- The MUG meets quarterly (about 15 people present)

- Tool development cost

- Example in a Division: 3 person–years per year for approximately 350 software engineers (**less than 1% of resources**)

- Example in a Division: 0.75 person years per year for approximately 70 software engineers (**about 1% of resources**)

- **Post–release metric costs have been insignificant compared to benefits;** in–process metrics are very useful, but they have been more costly and need to be automated

- **In general, the overall cost is acceptable and justified**

# BENEFITS SO FAR

- People started thinking about **software process and quality;** the data has **helped understand several problems and show how bad these problems were**

- **Metrics helped establish baselines, and focus on actions with quantitative results;** there are cases of significant quality and productivity improvements (example within a Division: 50X quality improvement in the last 3.5 years)

- **Presenting the metrics charts did not improve the quality by itself; it is the quality initiatives** taken as a **result of looking at the charts that made the difference**

- **Many indirect benefits** (e.g., helped improve ship-acceptance criteria, helped improve schedule estimation accuracy, etc.)

# LONG RANGE BENEFITS EXPECTED

- Software groups **learn from mistakes** of previous projects and take action to avoid them
- Significant **improvement in customer satisfaction** due to improved quality
- Significant **cost reduction** due to improved quality (reduced cost of rework; resources are freed up to work on new software development)
- Productivity improvement is expected to **reduce cycle time,** allowing the products to reach the market at the right time
- Remember: metrics can only show problems and give ideas as to what can be done; it is the **actions** taken that bring the benefits

# VIEWGRAPH MATERIALS

# FOR

# BOB GRADY,
# Hewlitt-Packard

# Experiences in Establishing and Maintaining an Effective Measurement Program

- What does our measurement program look like?
- What obstacles did we have to overcome?
- What are the costs of such a program?
- What are the benefits so far?
- What long-range benefits are expected?

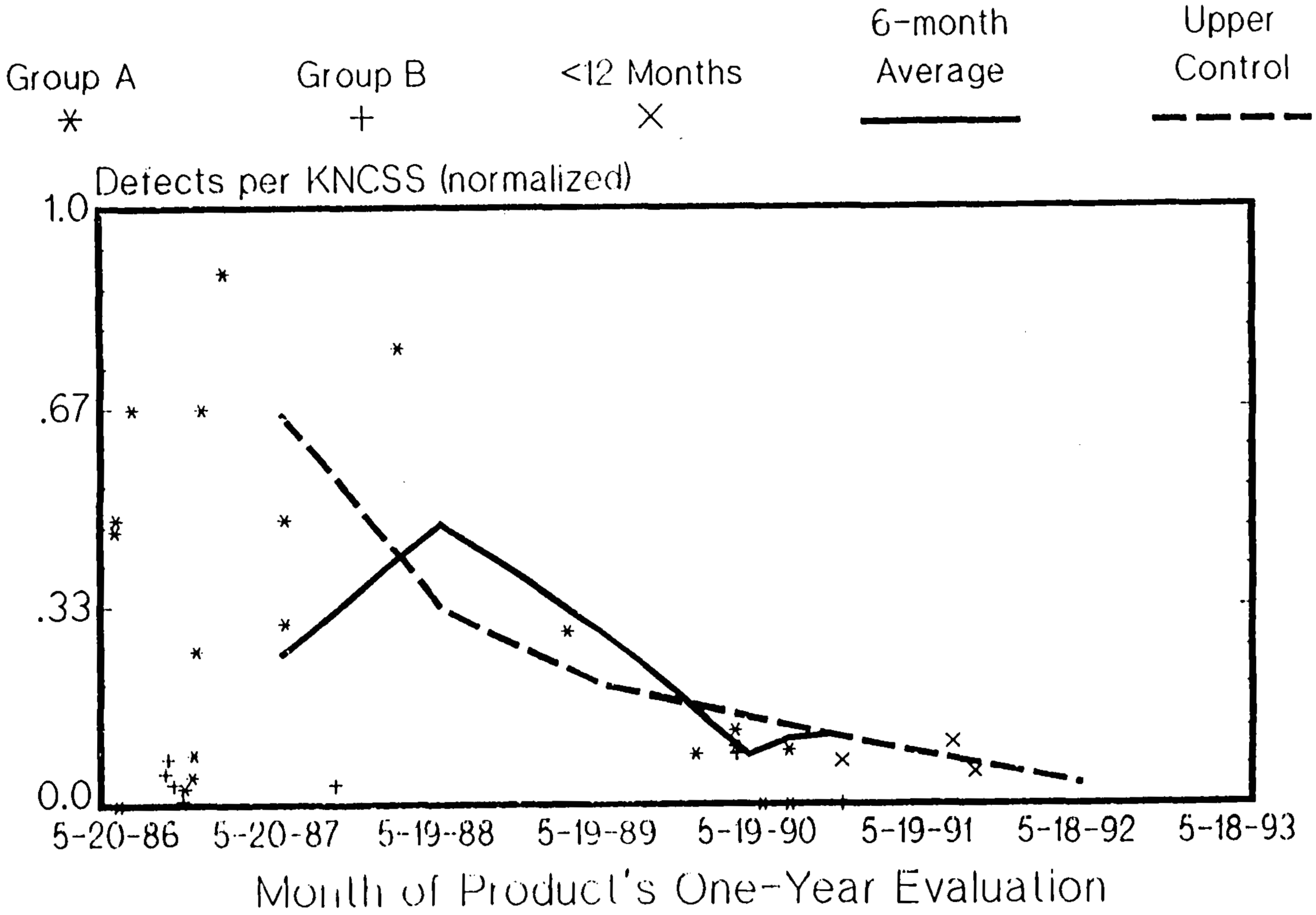
# HP's STANDARD SET OF METRICS

## Consists of Code Volume, Effort, and Defect Definitions:

- *NonComment Source Statement (NCSS)* – non-comment physical lines of code including compiler directives, data declarations, print statements and executable code. Each include file counted only once. [Pp58]
- *Engineering Month (EM)* – sum of calendar payroll months, attributed to each project engineer, including testing, adjusted to exclude extended vacations and leaves; excludes time project managers spend on management tasks. [Pp54]
- *Defect* – any deviation from the specification and any errors in the specification. [Pp.56]

R. Grady
Hewlett-Packard
Page 2 of 10

# Postrelease Discovered Defect Density

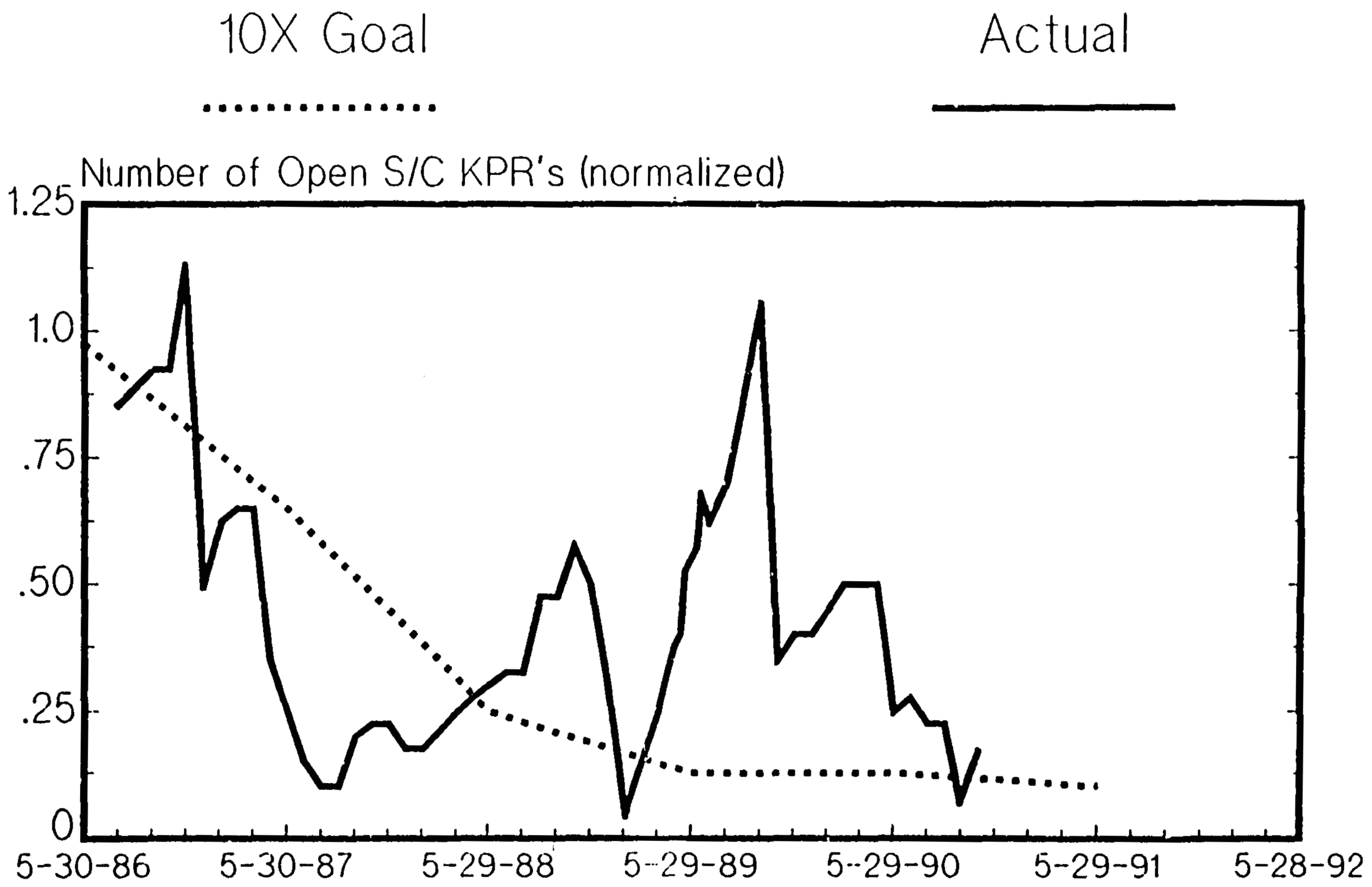


R. Grady
Hewlett-Packard
Page 3 of 10

Software Development Technology Lab
enxddsl-11/26/90

HEWLETT PACKARD

# Open Serious & Critical KPR's



Software Development Technology Lab

tenxscsl-11/26/90

HEWLETT PACKARD

# What Obstacles Did We Have to Overcome?

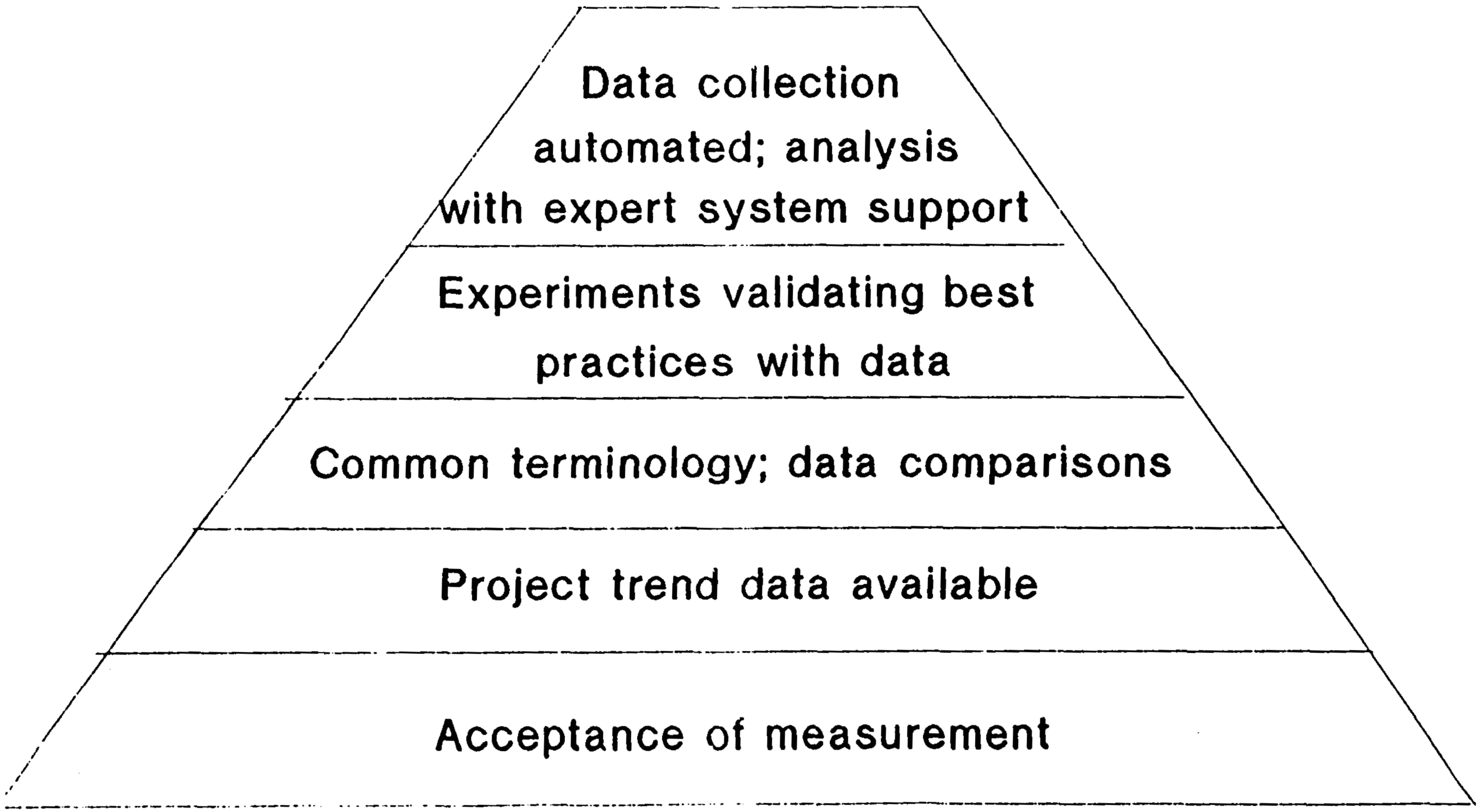
- **Perceptions of metrics.**
- **Selling.**
  - Top management
  - Project managers
  - Engineers
- **Too-rapid change.**
  - Leap before you look
  - More is better
  - Desperation for a breakthrough
- **Organizational changes.**

R. Grady
Hewlett-Packard
Page 5 of 10

# What Did We Do Right?

- Council.
- Started small.
- Created an environment for reinforcing success.
  - top management training
  - Software Engineering Productivity Conferences
  - productivity managers
- Metrics class.
- Good tool support.

# Hierarchy of Metrics Acceptance and Practice



Data collection automated; analysis with expert system support

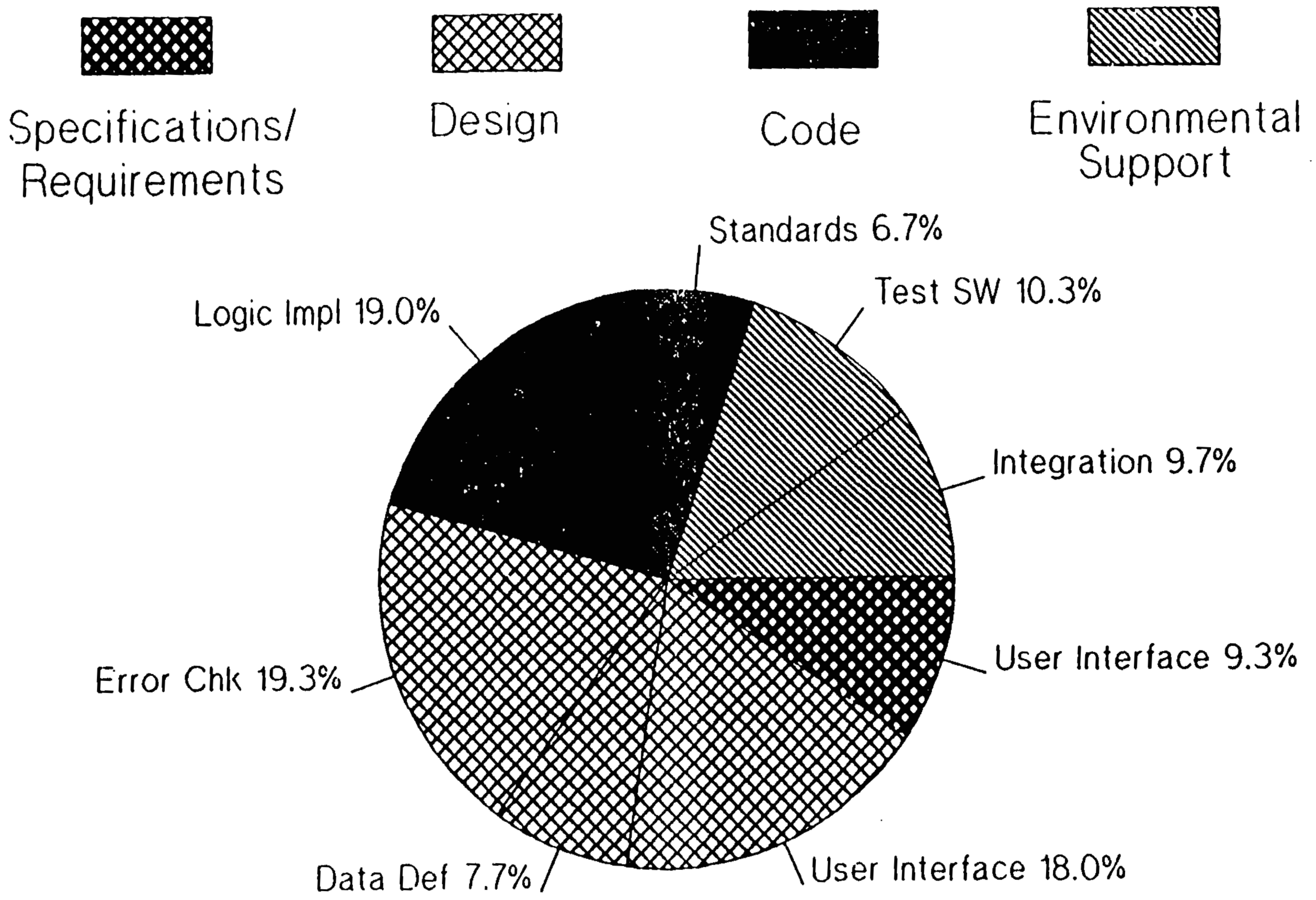
Experiments validating best practices with data

Common terminology; data comparisons

Project trend data available

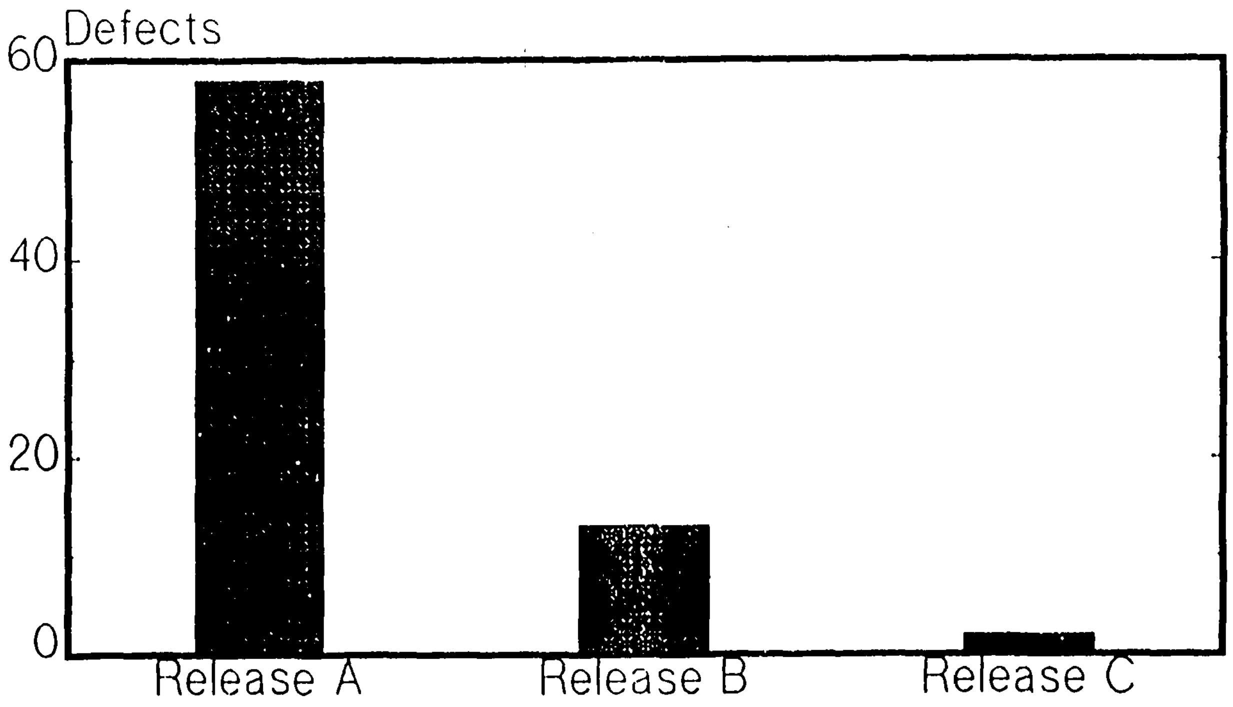
Acceptance of measurement

# Top Eight Causes of Defects for One Division



# ERROR HANDLING DEFECTS

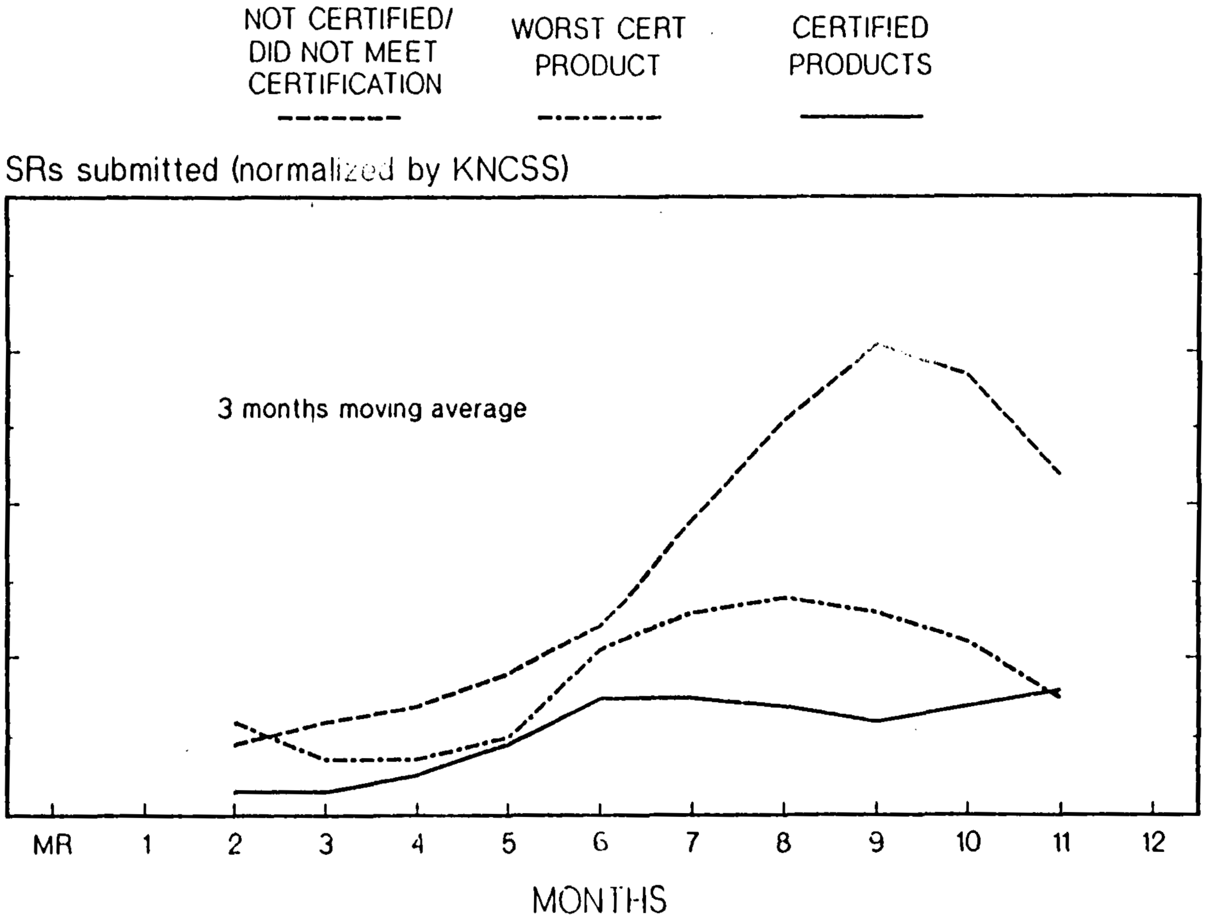
## (Results of New Standards)



R. Grady
Hewlett-Packard
Page 9 of 10

# SPECTRUM TECHNICAL PROGRAM

## Post-release Incoming SRs by Customers



R. Grady
Hewlett-Packard
Page 10 of 10

CSG/DLD/LMO/Software Methods Lab
sepcgp2g-06/10/88

HEWLETT PACKARD

VIEWGRAPH MATERIALS

FOR

RAY WOLVERTON,
Hughes Aircraft Company

# EXPERIENCES IN ESTABLISHING AND MAINTAINING AN EFFECTIVE MEASUREMENT PROGRAM

**Based on Managing Programming
Measurement, ITT Programming, Advanced Technology Center
from 1981 to 1986 at Stratford, Connecticut**

**Ray Wolverton
Hughes Aircraft Co.
Los Angeles
(213) 414-5515**

**A Presentation to the 15th Annual Software Engineering SEL Workshop,
NASA/Goddard
November 28-29, 1990**

## Programming Measurement Strategy

- Develop yearly baselines for progress comparison
- Develop project performance reporting system for management
- Develop forecasting and diagnostic measurement procedures for use by project personnel
- Establish and enhance integrated measurement system through yearly growth in the type and accuracy of measurements reported

# CONTEXT AND CIRCUMSTANCES

- Took three years to collect data and show trends
  - 29 companies
  - 106 projects
  - 7.2 million developed statements
  - 12.4 million delivered statements
  - 3,073 personyears
- Productivity, calculated as developed statements/personyear, measures the efficiency of developing new and modified code
- Project productivity is averaged by application type and year completed
- Telecommunication projects represent 45% of the projects and 71% of the effort in our last baseline
- Quality trends, quality improvements, and quality-productivity relationship was examined indepth

## Programming Measurement Objectives

- Develop project performance measures relating to economic productivity, costs and profits
- Reduce programming production costs thru increased efficiency
- Provide an early warning of project productivity, quality, schedule or cost difficulties
- Improve the development and defense of competitive bids
- Track programming productivity and quality trends
- Compare ITT to industry performance
- Measure the effectiveness of alternative programming methods/tools
- Understand the effect of management decisions on the programming process
- Track the progress of programming projects
- Analyze the impact and effectiveness of tools, methodology and technology
- Improve the allocation of resources

# Measurement Programs

- Baselines
- Performance factors
- Leading indicators
- Resource estimating
- Quality profile
- Programmer manager development
- Programmer competency and task analysis
- Product cost system
- Programming cost estimating

# Programming Productivity, Quality and Cost

## Major Factors

- Modern programming practices
- Programming personnel
- Organizational structure
- Tools available
- Restarts and direction changes
- Attrition
- Number of locations
- Project complexity
- Computer availability
- Project objectives & requirements
- Physical environment
- Initial plans, schedules & cost estimates

# PROGRAMMING MEASUREMENT TEAM

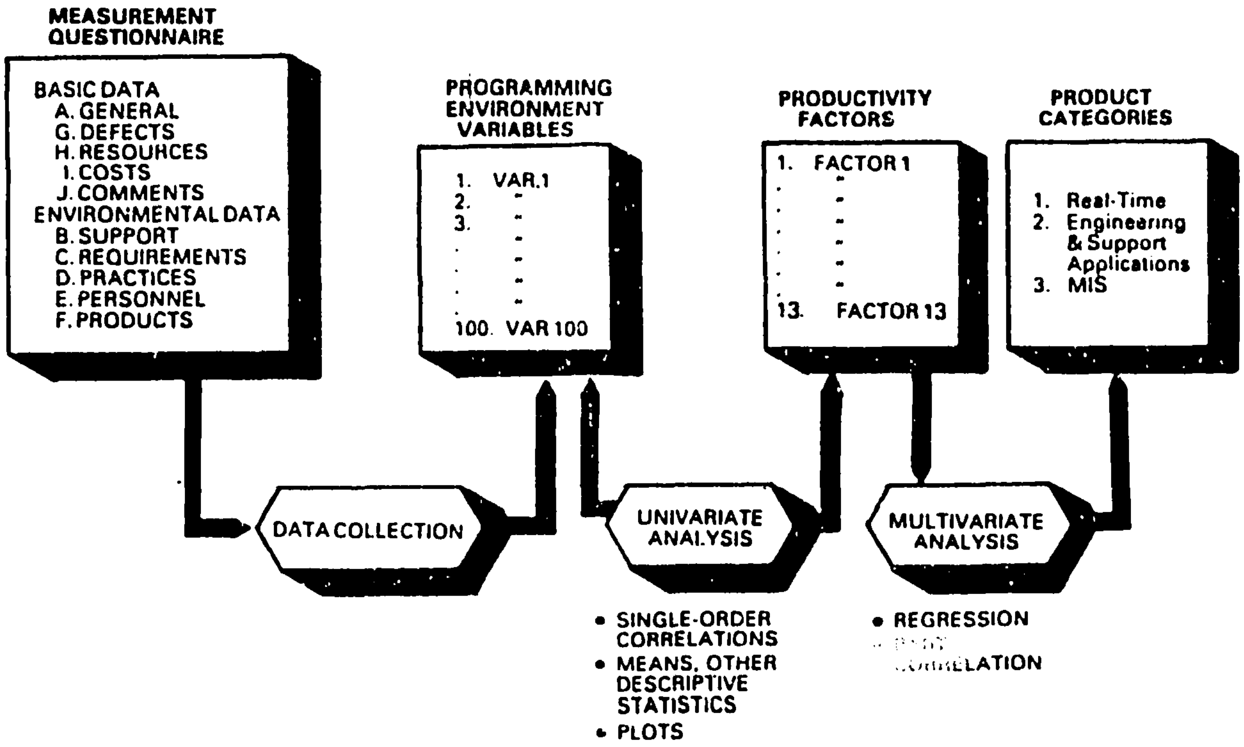
## ITT

| | |
|---|---|
| Bert Albert | Data Coordinator |
| Bill Curtis | Human Factors |
| Sue Hoben | MIS Specialist |
| Yuan Liu | Statistical Modeling |
| Hank Malec | Quality Issues |
| John Vosburgh | Statistical Analysis |
| Ray Wolverton | Management |

## CONTRACT

| | |
|---|---|
| Tom Jopling | FOCUS Programmer |
| Bruce Roberts | Yale Grad Student |
| Lynn Truss | Yale Grad Student |
| Barbara Conway | Graphics |

## Overall Approach to the Study of Productivity Performance Factors



MEASUREMENT QUESTIONNAIRE

BASIC DATA
A. GENERAL
G. DEFECTS
H. RESOURCES
I. COSTS
J. COMMENTS
ENVIRONMENTAL DATA
B. SUPPORT
C. REQUIREMENTS
D. PRACTICES
E. PERSONNEL
F. PRODUCTS

PROGRAMMING ENVIRONMENT VARIABLES

1. VAR.1
2. "
3. "
. "
. "
. "
100. VAR 100

PRODUCTIVITY FACTORS

1. FACTOR 1
. "
. "
. "
. "
. "
13. FACTOR 13

PRODUCT CATEGORIES

1. Real-Time
2. Engineering & Support Applications
3. MIS

DATA COLLECTION

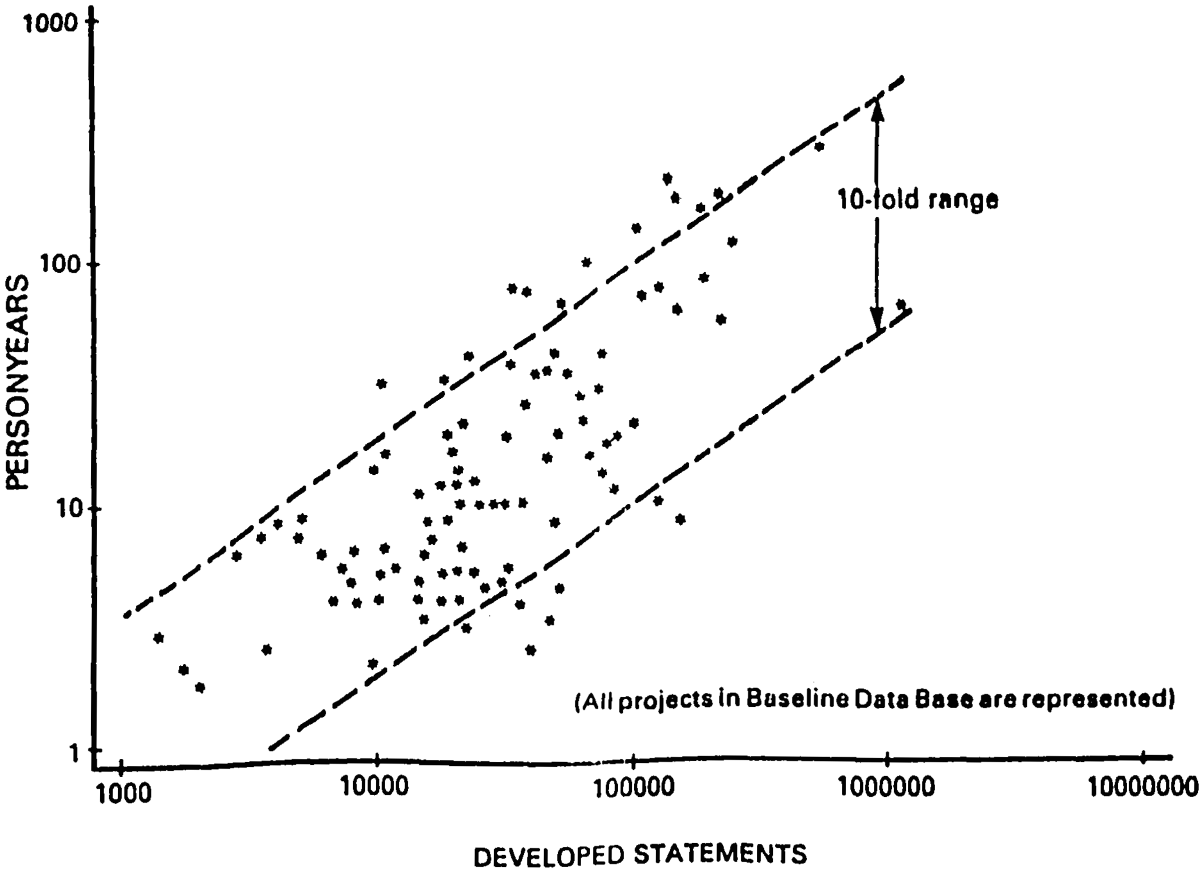
UNIVARIATE ANALYSIS

- SINGLE-ORDER CORRELATIONS
- MEANS, OTHER DESCRIPTIVE STATISTICS
- PLOTS

MULTIVARIATE ANALYSIS

- REGRESSION
- PART CORRELATION

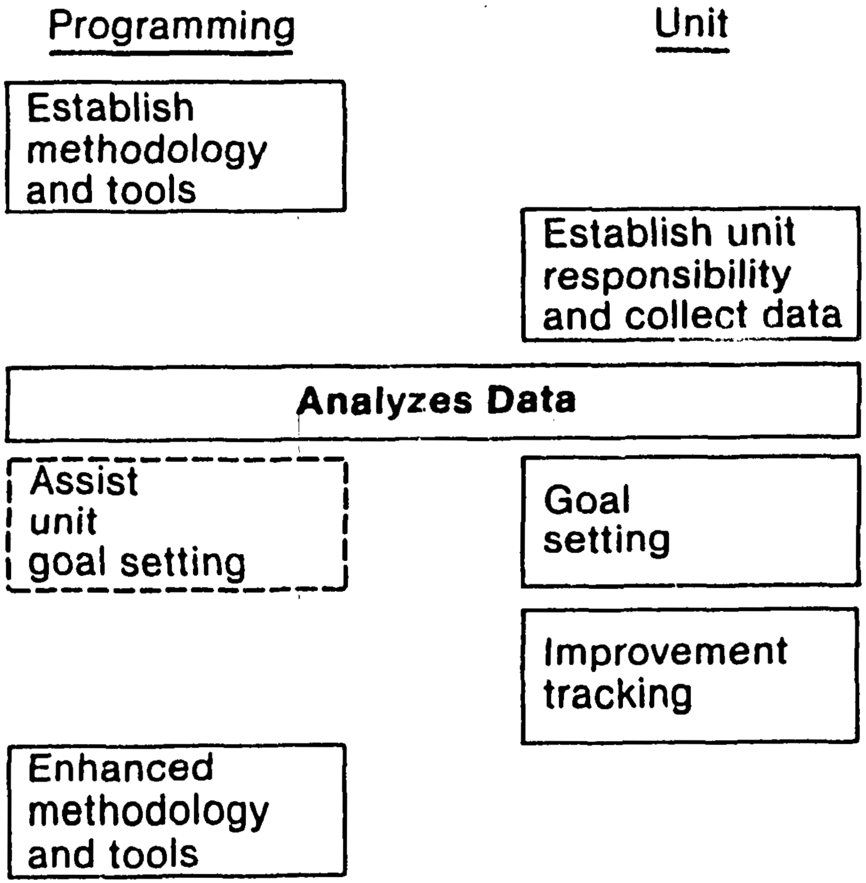
## Plotting of Personyears versus Developed Statements



(All projects in Baseline Data Base are represented)

# Measurement Strategy

## Responsibilities



Programming

Unit

Establish methodology and tools

Establish unit responsibility and collect data

Analyzes Data

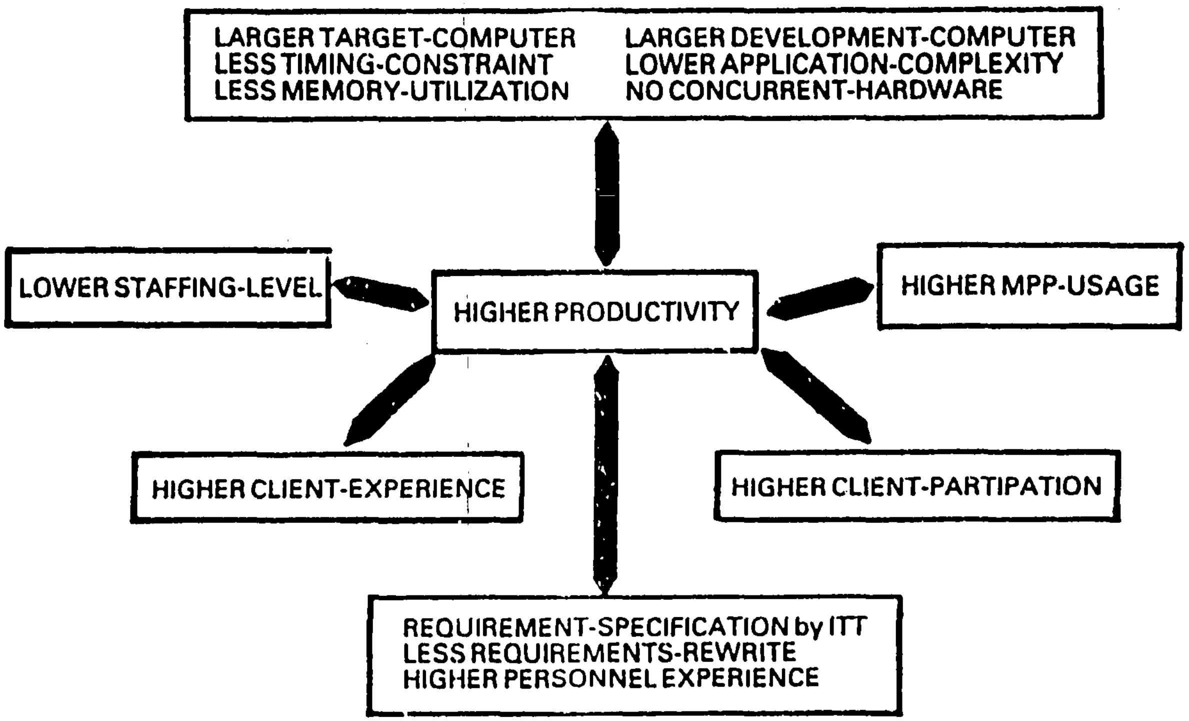
Assist unit goal setting

Goal setting

Improvement tracking

Enhanced methodology and tools

## Strongly Correlated Groups of Productivity Factors



LARGER TARGET-COMPUTER
LESS TIMING-CONSTRAINT
LESS MEMORY-UTILIZATION

LARGER DEVELOPMENT-COMPUTER
LOWER APPLICATION-COMPLEXITY
NO CONCURRENT-HARDWARE

LOWER STAFFING-LEVEL

HIGHER PRODUCTIVITY

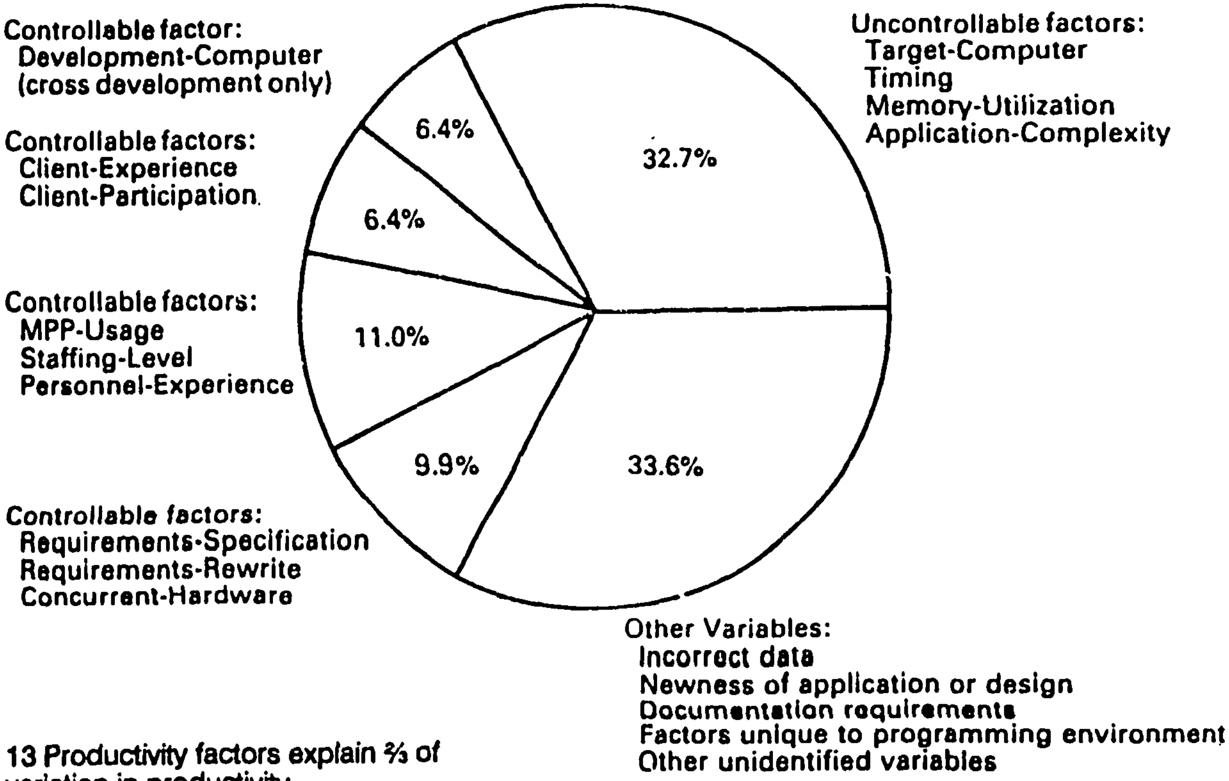
HIGHER MPP-USAGE

HIGHER CLIENT-EXPERIENCE

HIGHER CLIENT-PARTIPATION

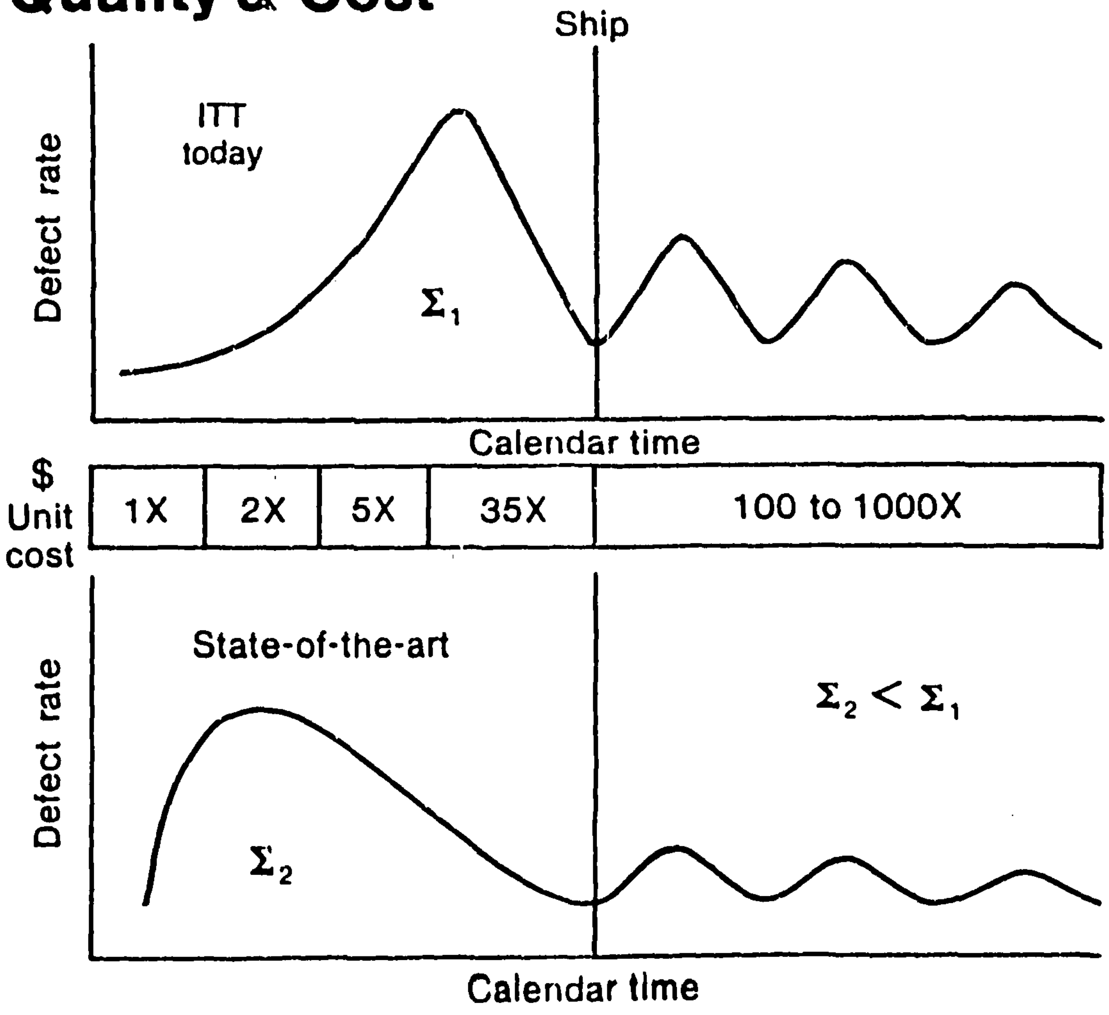
REQUIREMENT-SPECIFICATION by ITT
LESS REQUIREMENTS-REWRITE
HIGHER PERSONNEL EXPERIENCE

## Breakdown in the Variation of Productivity of Non-MIS Programming Projects in ITT



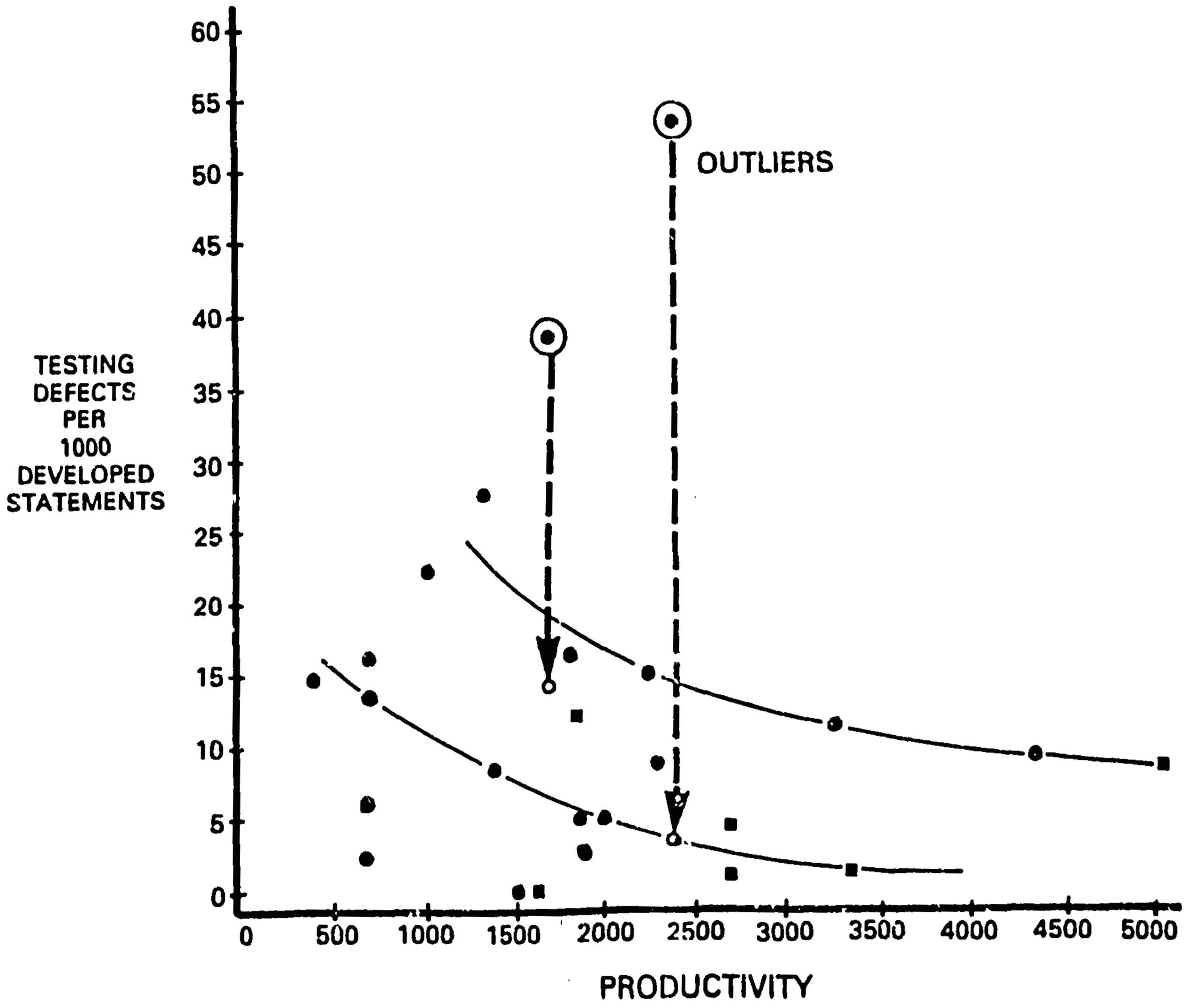
- 13 Productivity factors explain ⅔ of variation in productivity
- When combined with developed statements, these 13 productivity factors explain 90 percent of the variation in effort

# Quality & Cost

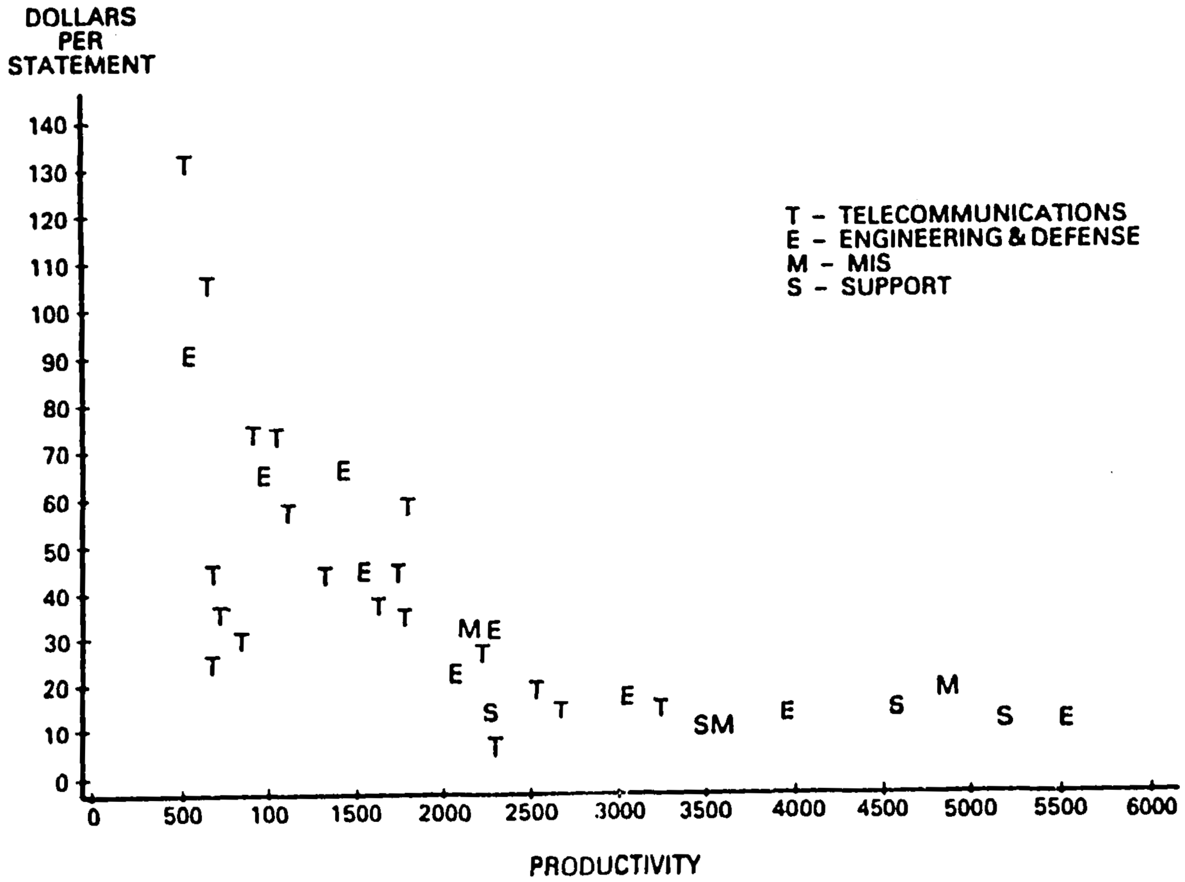


| $ Unit cost | 1X | 2X | 5X | 35X | 100 to 1000X |
|---|---|---|---|---|---|

## Relationship Between Productivity and Quality

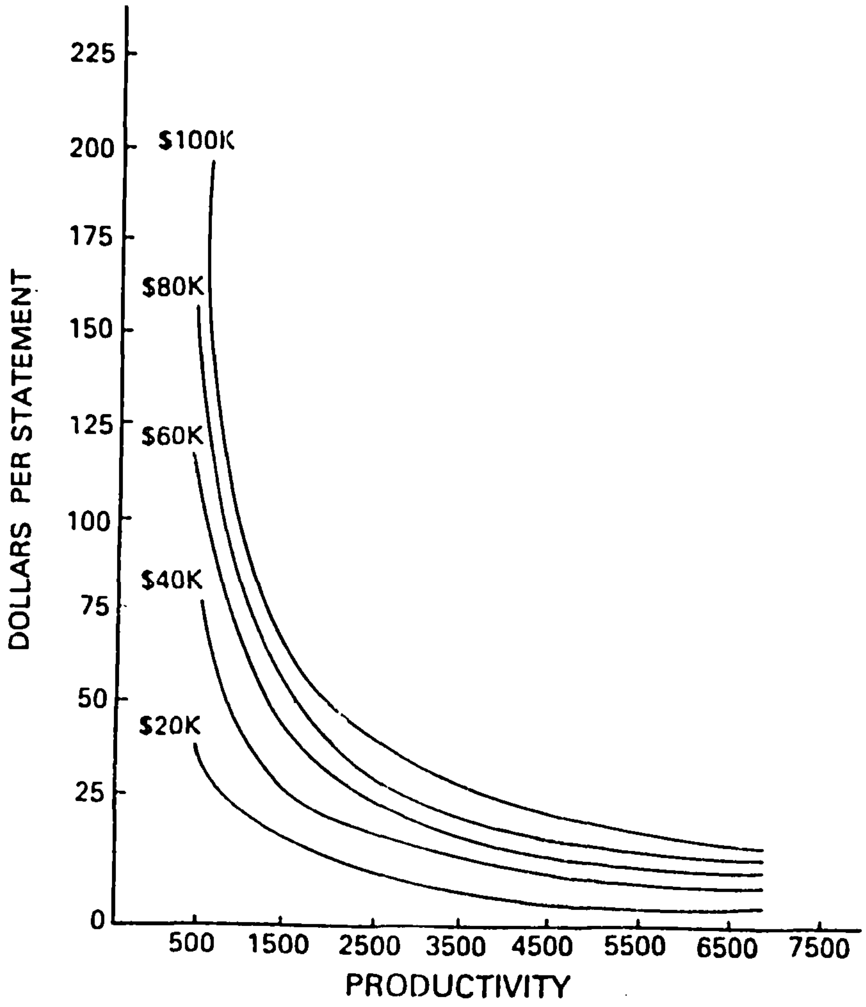


## Scatterplot of Cost Per Statement Against Productivity

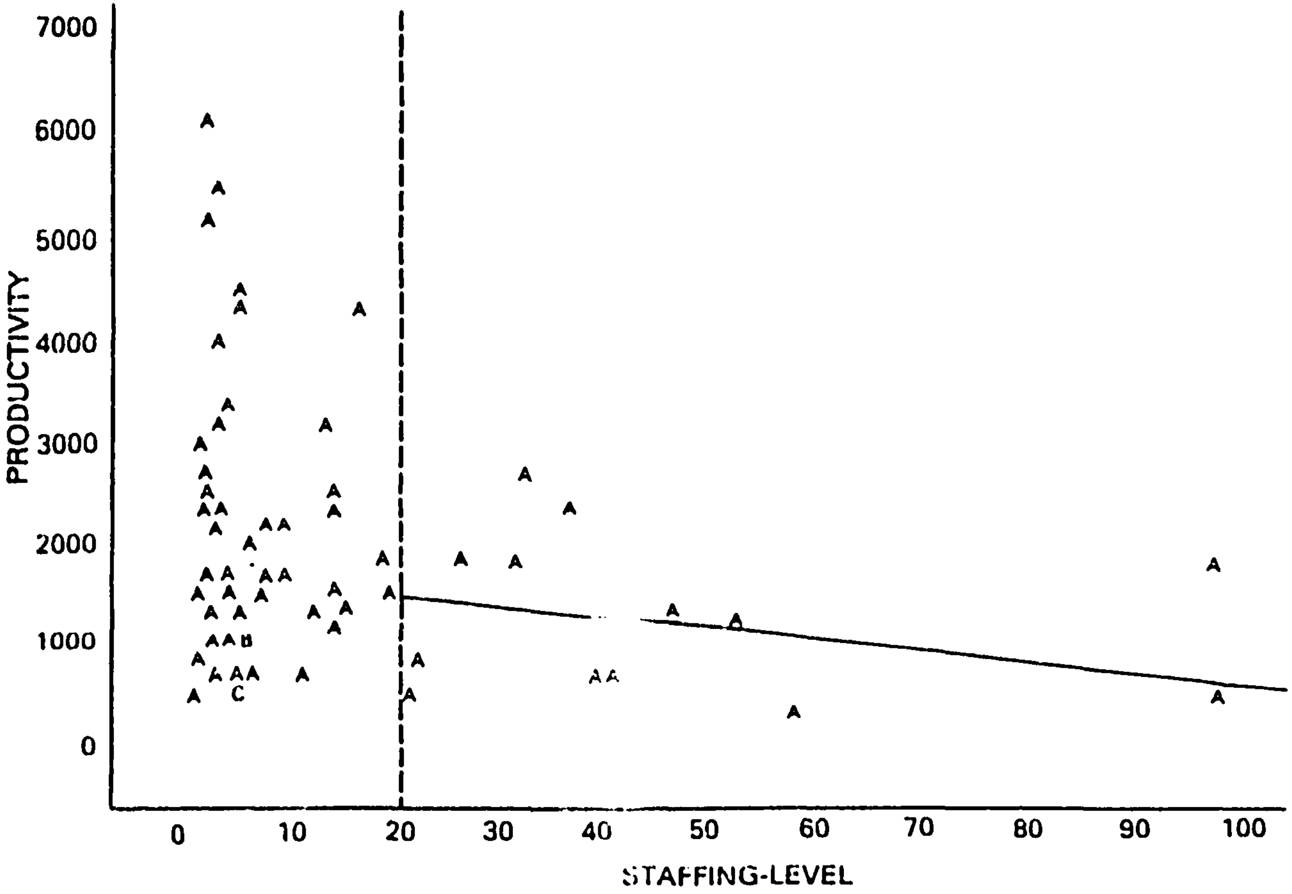


* 7 MIS projects with productivity above 6000 statements per personyear are out of range.

## Relationship Between Dollars Per Statement and Productivity for Various Levels of Burdened Personyears



## Distribution of Productivity with Respect to Staffing-Level



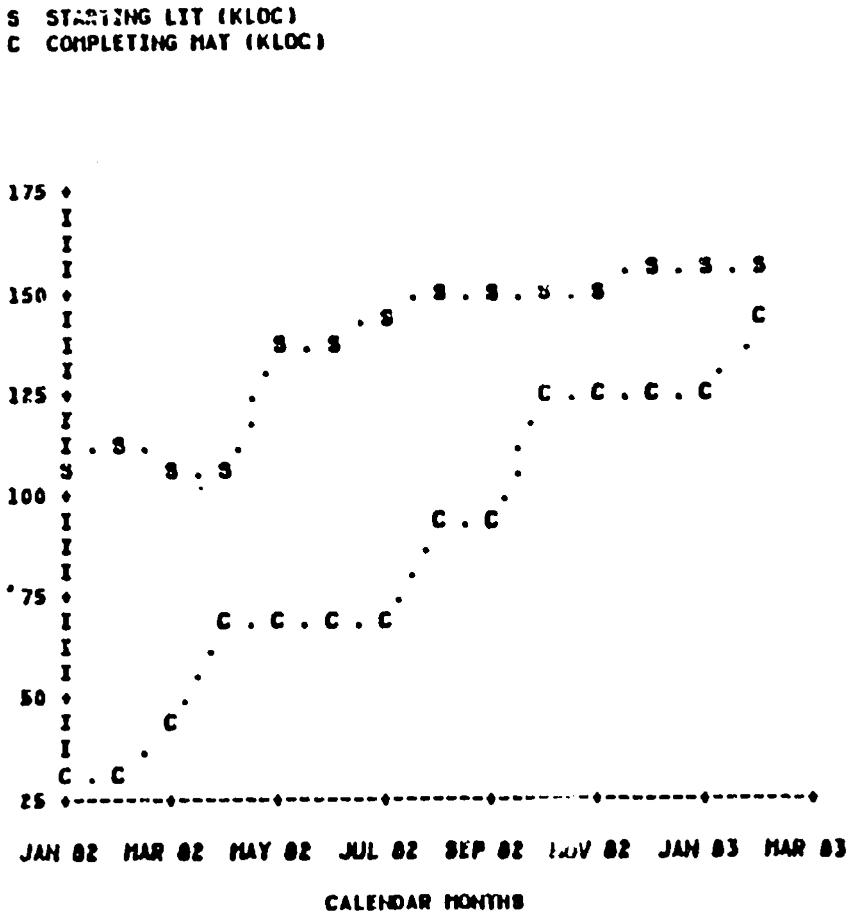
# Programming Measurements

## 10 Leading Indicators

- Code production
- Defect removal
- Test achievement
- Defect rates
- Test effectiveness index
- Schedule index
- Problem index
- Development hours index
- Development cost index
- Work performed index

CODE PRODUCTION VERSUS DATE - INDICATOR #1 RUN DATE: 02/25/83

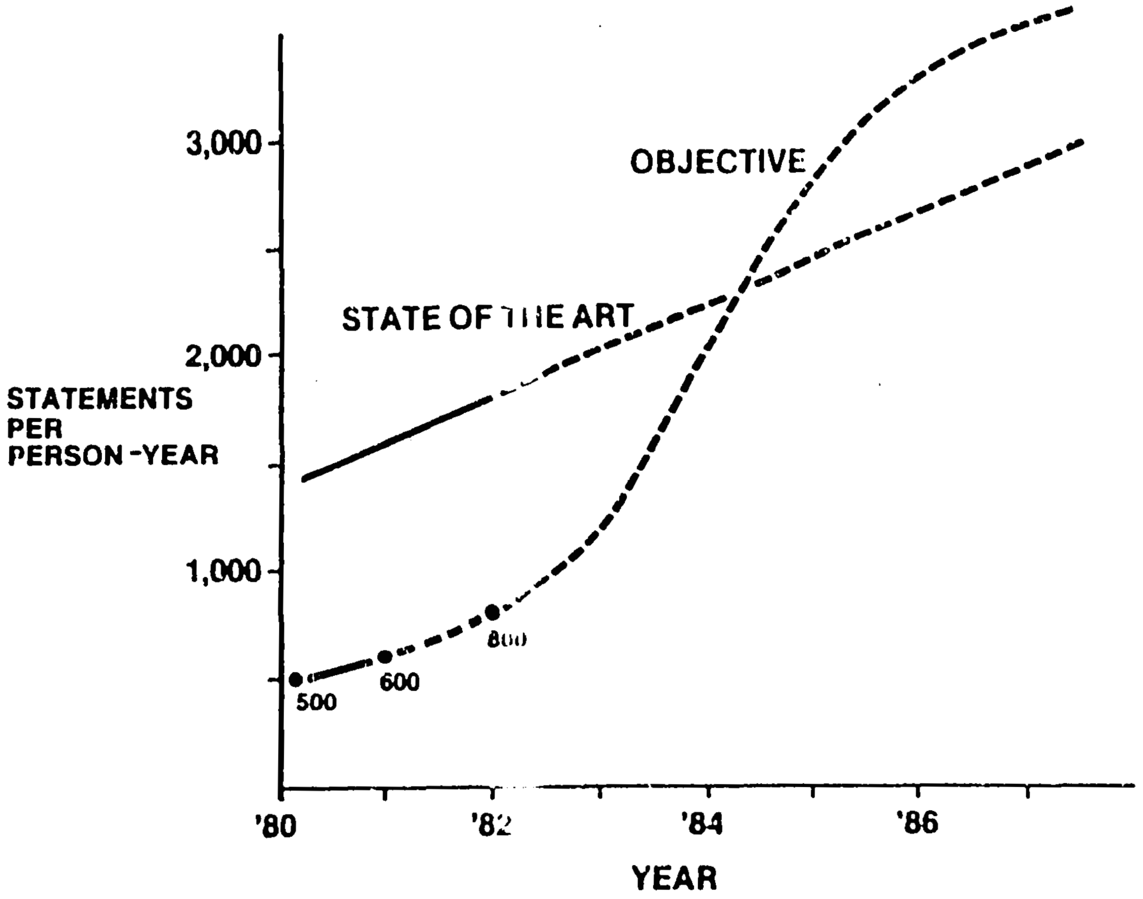
COMPANY: SEL PROJECT: 1240



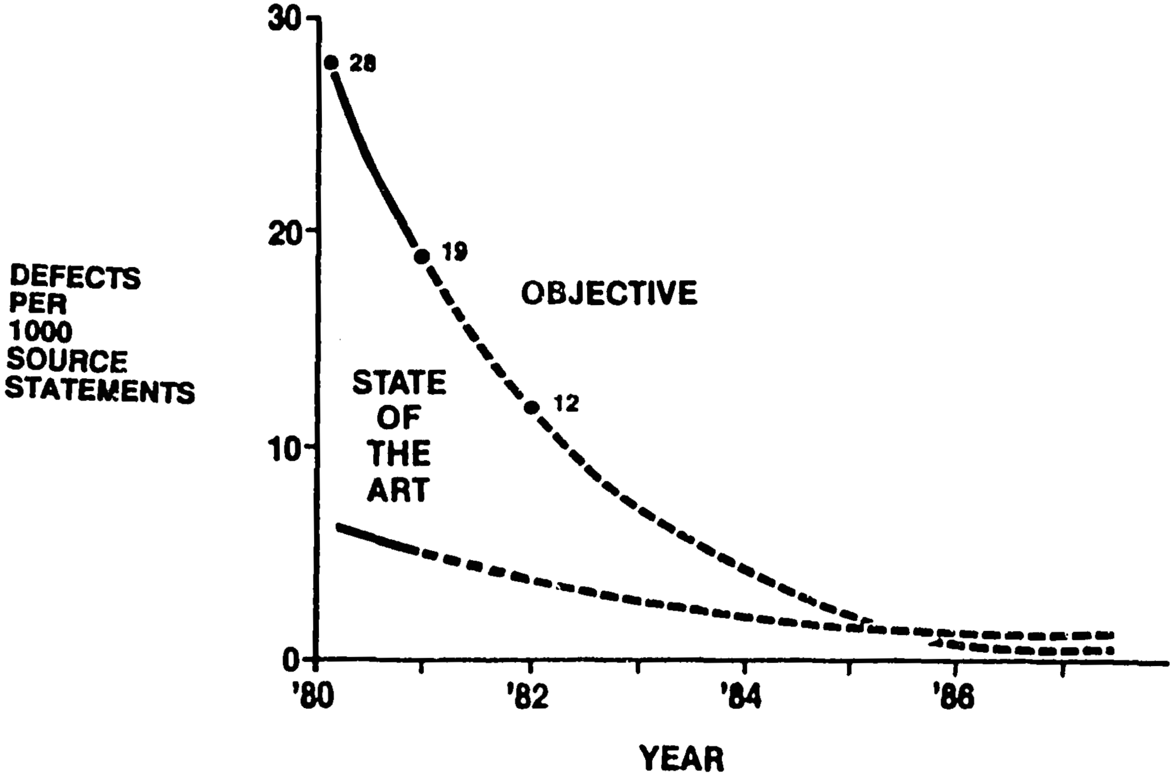
| | |
|---|---|
| JANUARY-JUNE 1982 | THE LAG TIME IS BUILDING. THE MODULE MANAGER SHOULD TAKE ACTION. |
| OCTOBER | LAG TIME IS DOUBLE THE 20 WEEKS IN THE PLAN. <u>TAKE ACTION NOW</u> |
| NOVEMBER - FEBRUARY 1983 | GOOD WORK. ACTION TAKEN HAS GOOD RESULT STAY ON TOP OF THE SITUATION. |

**Programming Measurement**

# Programming Productivity Target



# Programming Quality Target



## Unit Reaction to Program Introduction

"Need It"
"Want It"

**BUT**

"In The Middle of Project"
"No Resources"
"Takes Time to Collect Data"

## Unit Reaction to Initial Testing & Installation

"Can't Get All The Data"
"Can't Show Data – No Actions in Place"
"Data Not Accurate"
"Internal Use Only"

**VIEWGRAPH MATERIALS**

**FOR**

**MITSURU OHBA,**
**IBM/Japan**

# Experiences in Implementing an Effective Measurement Program

November 29, 1990

Mits Ohba

IBM Japan

## 1. Introduction

Japanese believe:

***"What other persons do are right things to do."***

In this context, "other persons" could be:

- Other organizations in a division

- Other divisions in a company

- Other companies in the industry

- Other industries in Japan

- US

## 2. What do Japanese measurement programs look like?

The most common and basic measures are:

- Size: KLOC
  non-commentary source lines of code including reused source code

- Productivity: LOC per Programmer Month
  indirect activities (e.g., tool development) not included

- Quality: Errors per KLOC
  errors reported during the development phases and 12 month after release

Though there are differences in details, the measures are conceptually same as those in US.

## 3. What obstacles did Japanese have to overcome?

We had to agree upon what should be measured:

- what kind of data should be collected?
- how should data be analyzed?
- how should results be fed back?

The answers are different -> no standard measures exist.

## 4. Who defines a measurement system for an organization?

There are two cases:

- Central software technology support group
  e.g., NEC, Toshiba, Mitsubishi, NTT

- Quality assurance or equivalent organizations
  e.g., Fujitsu, Hitachi, Oki, IBM Japan

Based on:

- De facto standard measures (e.g., KLOC)

- Working papers by various committees (e.g., JSA)

## 5. What is the cost of such programs?

Needs a centralized organization which is responsible for:

- defining measures and evaluation systems
- defining data to be collected
- defining "simple" methods for analyzing data
- developing tools for collecting and analyzing data
- maintaining the database
- providing education

It is expensive.

## 6. What are the benefits so far?

"Takes at least three years to see the changes."

- Management by quantitative objectives
  by setting objectives and reviewing achievements
  (e.g., software reliability growth estimation)

- Standard and consistent control of software process
  by defining the upper and the lower control limits
  (e.g., quality probe by Hitachi)

- Incremental and continuous process improvement
  by setting an annual goal for an organization
  (e.g., "Ayumi (growth)" program by Fujitsu)

As a result, the defect rate has gone down from 5 Errors/KLOC to 0.1 Errors/KLOC during the last ten years.

## 7. What long range benefits are expected?

"Optimization" of software process:

- Design the best process for a project based on past experiences
- Monitor and manage the process properly based on data analysis
- Reconfigure the process dynamically if needed based on data analysis results and experiences

This is the ideal "Software Factory" and what engineers in other Japanese industries have done last 25 years.

# VIEWGRAPHS FOR PANEL 2

# VIEWGRAPH MATERIALS

# FOR

# LARRY DRUFFEL, SEI

# 15th Annual Software Engineering Workshop

November 29, 1990

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Sponsored by the U.S. Department of Defense

# Disappointments

Ada acceptance - military, industry, education

Acceptance of software engineering in education

Standards for integration of CASE tools

Code reuse - again

# Achievements

Ada standardization and evolution of supporting technology

Focus on process supported by measurement and education

Emergence of software engineering in education

Realization of importance of software architecture

Emergence of software engineering environments

Object-oriented design notions

# Next Five Years

Software architecture

Maturation of object-oriented design

Data collection - consistent measures

## VIEWGRAPH MATERIALS

## FOR

## MANNY LEHMAN,
## Imperial College

# SOFTWARE ENGINEERING IN THE 1980's
# Most Significant Achievements / Greatest Disappointments

(Panel)

## 15TH ANNUAL SOFTWARE ENGINEERING WORKSHOP
## GODDARD SPACE CENTER

28 - 29 November 1990

M M Lehman
Department Of Computing
Imperial College Of Science Technology & Medicine
&
Lehman Software Technology Associates Ltd
London SW7 2BZ

MOST SIGNIFICANT ACHIEVEMENT

# CASE

GREATEST DISAPPOINTMENT

# CASE

## MOST SIGNIFICANT ACHIEVEMENTS

NOTE THAT TERM IS ***MOST SIGNIFICANT*** NOT ***GREATEST***

- INCREASING APPRECIATION OF THE INTRINSICALLY **EVOLUTIONARY** NATURE OF SOFTWARE

⇓ (IN MY VIEW)

- INCREASING RECOGNITION OF THE **SIGNIFICANCE** & **ROLE** IN SOFTWARE DEVELOPMENT & EVOLUTION OF THE **PROCESS** & OF **PROCESS MODELS**

⇓ (IN MY VIEW)

- **MUST** EVENTUALLY LEAD TO WIDER APPRECIATION AND ACCEPTANCE OF THE IMPORTANCE OF **DISCIPLINE, METHOD, FORMALITY** & **MECHANISATION** IN THAT PROCESS AND TO THEIR GENERAL APPLICATION

⇓ (IN MY VIEW)

- THE DEVELOPMENT OF SIGNIFICANT **CASE TOOLS** & OF THE CONCEPT OF **INTEGRATED SUPPORT ENVIRONMENTS**

THE NEED FOR CASE DEVELOPMENT IS A LOGICAL CONSEQUENCE OF THE FIRST LISTED ADVANCES BUT, IN PRACTICE, HAS PRIMARILY ARISEN INDEPENDENTLY FROM THE SEARCH FOR **PRODUCTIVITY** GROWTH IN SOFTWARE DEVELOPMENT

# GREATEST DISAPPOINTMENT

- THE FAILURE, BY & LARGE, OF CASE TO DELIVER **EVIDENT** PRODUCTIVITY GROWTH

⇓ (IN MY VIEW)

- THE LACK OF LARGE SCALE **PENETRATION** OF CASE WIDELY OR DEEPLY INTO INDUSTRIAL & COMMERCIAL SOFTWARE DEVELOPMENT

- THE VERY SLOW DEVELOPMENT OF **SATISFACTORY, COMPREHENSIVE, TRANSFERABLE & USABLE** SUPPORT ENVIRONMENTS

- FAILURE TO ACHIEVE WIDE INDUSTRY APPRECIATION OF THE **TRUE** MEANING OF SOFTWARE ENGINEERING & THE **ROLE** OF SOFTWARE ENGINEERS

# SOFTWARE ENGINEERING - SOFTWARE ENGINEER

⇓ (IN MY VIEW)

## *PROCESS ENGINEERING - PROCESS ENGINEER*

CONTRASTS WITH

***PROGRAMMING* AS *PRODUCT ENGINEERING***

- SOFTWARE ENGINEERS' FOCUS - ***PROCESSES*** BY WHICH SYSTEMS ARE ***DEVELOPED, PRODUCTS CREATED*** & ***MAINTAINED*** SATISFACTORY

- PRIMARY CONCERN: ***DESIGN, CONTROL, SUPPORT*** OF DEVELOPMENT & EVOLUTION ***PROCESS***
    - PROCESS ITSELF
    - METHODS
    - TOOLS

- **SELECT, DEVELOP,** REDUCE TO **PRACTICE**
    - **METHODS & TECHNIQUES**
    - **PRACTICES & PROCEDURES**
    - **DIRECT TOOLS**
    - **GENERAL SUPPORT**

- **INTEGRATE & INSTALL** METHODS, TOOLS & IPSEs TO PROVIDE SUPPORT & ***INFORMATION RETENTION*** OVER **ORGANISATION, APPLICATION, PROJECT**

- INVOLVEMENT WITH **SPECIFIC** SYSTEM
    - **PROJECT & PROCESS DESIGN**
    - **PLANNING**
    - DEVELOPMENT OF PROJECT-SPECIFIC ⇒ METHODS  
      ⇒ TOOLS
    - **MANAGEMENT SUPPORT**
    - PROCESSES MANAGEMENT

# THE ROLE OF CASE

- SUCH OVERALL ATTRIBUTES CANNOT BE ACHIEVED BY **CLASSICAL** PROCESS

- OR BY **UNCO-ORDINATED** INTRODUCTION OF INDIVIDUAL METHODS OR TOOLS

- DEMANDS **CO-ORDINATED, PROCESS WIDE,** INTRODUCTION OF
  - **SYSTEMATIC & DISCIPLINED** METHODS,
  - CASE TOOLS FOR THEIR SUPPORT
  - AN INFORMATION PRESERVING **REPOSITORY,**
  - **ACTIVE** PROCESS **GUIDANCE**

- PRIMARY GOAL OF CASE CANNOT BE IMMEDIATE **PRODUCTIVITY** GROWTH, COST **REDUCTION,** OR **VISIBLE** IMPROVEMENTS IN PRODUCT **QUALITY**

- IMPROVED **COST-EFFECTIVENESS** ULTIMATE BENEFIT

- **VISIBILITY** DEPENDS ON MEETING ALL ABOVE **REQUIREMENTS** OVER PERIOD OF TIME

- INTRODUCTION OF CASE MUST SEEK TO YIELD TOTAL PROCESS THAT AS FAR AS POSSIBLE, **ACHIEVES & MAINTAINS,** USER **SATISFACTION** WITH PRODUCT

- THIS, IN TURN, WILL EVENTUALLY PRODUCE **PRODUCTIVITY INCREASE & REVENUE GROWTH**

## SOME OBSTACLES - TO RAPID PENETRATION OF CASE TECHNOLOGY

- LONG LEAD TIME TO VISIBLE BENEFIT
  - ⇒ INDIVIDUAL & COLLECTIVE LEARNING & TRAINING
  - ⇒ AND CHANGE OF ATTITUDES, HABITS & PRACTICES
  - ⇒ WRITE OFF OF CURRENT PRODUCER INVESTMENT
    eg TOOLS, PROCEDURES, EXPERIENCE
  - ⇒ ACHIEVING SIGNIFICANT BENEFIT DEPENDS ON COHERENT COVERAGE OF MAJOR PORTION OF DEVELOPMENT PROCESS
  - ⇒ BENEFITS OF CASE BECOME SIGNIFICANT ONLY TOWARDS SYSTEM COMPLETION OR AFTER EXTENSIVE USAGE EXPERIENCE

- MAJOR INVESTMENT BY SOFTWARE PRODUCERS
  - ⇒ CAPITAL - eg. TOOLS
  - ⇒ MANPOWER - eg. DEVELOPMENT &TRAINING
  - ⇒ DISRUPTION IN PRODUCTION
  - ⇒ RETURN DELAYED TILL USERS PERCEIVE FISCAL BENEFIT

- IMPONDERABLES IN COST/BENEFIT ANALYSIS

  *PRODUCERS*
  - ⇒ HIDDEN BENEFITS
    eg CONSERVATION OF SKILLED MANPOWER
  - ⇒ ANTIREGRESSIVE BENEFITS
    eg POTENTIAL REDUCTION IN EMERGENCY RESPONSE TO USER PROBLEMS
  - ⇒ MARKETING VALUE OF PRODUCT QUALITY IMPROVEMENT
    eg SIMPLER TO LEARN, MORE RESPONSIVE MAINTENANCE

  *USERS*
  - IMPROVED QUALITY
    eg SIMPLER TO LEARN, INSTAL, MAINTAIN, ADAPT
  - ANTIREGRESSIVE BENEFITS
    eg REDUCTION IN USER DOWN TIME & LOSSES
  - HIDDEN BENEFITS
    eg FASTER RESPONSE TO CHANGE AND EVOLUTION REQUESTS

- INSUFFICIENT EXPERIENCE, DATA OR THEORY FOR CONVINCING DETERMINATION OF BENEFIT

# VIEWGRAPH MATERIALS

# FOR

# HARLAN MILLS, SET

# Software Engineering Achievements and Disappointments of the Past Decade

15th Annual Software Engineering Workshop
November 28-29, 1990
NASA/Goddard

Harlan D. Mills
Software Engineering Technology, Inc.
Vero Beach, Florida

# Significant Achievements Past Decade

Spiral Model of Software Development
in Place of Waterfall Model

Significant Developments of Metrics
for Software Technical Management

Establishment of National Resource in
Software Engineering Institute

Cleanroom Engineering of Software
under Statistical Quality Control

# Greatest Disappointments Past Decade

Use of Software Engineering as a Buzzword,
not as a Real Engineering Discipline

Continued, Widespread, but Unnecessary

- Poor Quality, Unreliable Software
- Low Productivity Software Development
- Missed Schedule Software Deliveries

# Cleanroom Engineering of Software under Statistical Quality Control

Statistical Usage Specifications as well as
Function and Performance Specifications

Software Development in a Pipeline of
Increments with Separate Certification

Scaled Up Informal Verification of Software
to Meet Specifications

Producing Software without Private Debugging
before Public Certification Testing

# Significant Achievements Next Decade

Formalization of Spiral Models of Software Development for Procurement/Management

Continued Developments of Metrics for Software Technical Management

Continuation of National Resource in Software Engineering Institute

Expanded Cleanroom Engineering of Software under Statistical Quality Control

# VIEWGRAPH MATERIALS

# FOR

# VIC BASILI,

**University of Maryland**

# Software Engineering in the 1980's:
# The most Significant Achievements and Greatest Disappointments

What have been the most significant achievements for software engineering in the past 10 years?

What have been the greatest disappointments for software engineering in the past 10 years?

What is the objective or subjective criteria supporting your assessments?

What software engineering advances will make the most significant contribution in the next five years?

# Significant Achievements

**Maturing:**

Recognition of importance of Process, Formal Methods

Recognition of need for Multiple Life Cycle Models, Methods, etc.

**Technologies:**

Measurement

Use of Data Abstractions and Object Oriented methods

Use of Ada

# Greatest Disappointments

That the maturing has taken so long

That some people are still looking for magic

The lack of wide spread use of measurement, and formal methods

The lack of effective automated support for software development

The lack of advance in testing practices

# Future Achievements

**Next 5 years:**

Focus on Engineering

Wider spread use of process improvement through measurement

Reuse of Packaged Experience

**Next 10 years:**

Real automated support

Maturing of personnel with consistent background

# APPENDIX A—ATTENDEES

## FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES

| | |
|---|---|
| Acton, Dorothy | IBM |
| Adams, Neil | Bendix Field Engineering Corp. |
| Addelston, Jonathan D. | Planning Research Corp. |
| Agresti, Bill W. | Mitre Corp. |
| Alban, David | Computer Sciences Corp. |
| Alexander, Linda C. | CECOM Center for Software Engineering |
| Allison, Don R. | TRW |
| Ammann, Paul E. | George Mason University |
| Anderson, Frances | Stanford Telecommunications, Inc. |
| Angier, Bruce | Institute for Defense Analyses |
| Arend, Mark | McDonnell Douglas |
| Arnoff, Barbara | Social Security Administration |
| Arnold, Jo Lynn | IRS |
| Arthur, James D. | Virginia Tech University |
| Astill, Pat | Centel Federal Services |
| Auernheimer, Brent | California State University |
| Ayers, Everett | Arinc Research Corp. |
| Bailin, Sidney | Computer Technology Associates, Inc. |
| Basili, Vic | University of Maryland |
| Beach, Jim | IBM |
| Beard, Robert M. | Computer Sciences Corp. |
| Beck, Hank | Jet Propulsion Lab |
| Benjamin, Chuck | SAIC |
| Bennett, Keith H. | University of Durham, UK |
| Berrey, Linda | IBM |
| Beswick, Charlie A. | Jet Propulsion Lab |
| Biddle, John M. | Martin Marietta |
| Biow, Christopher | Defense Communications Agency |
| Bisignani, Margaret | Mitre Corp. |
| Bissonette, Michele | Computer Sciences Corp. |
| Blagmon, Lowell E. | Naval Center for Cost Analysis |
| Blake-Hedges, Wayne | OAO |
| Blonchek, Robert M. | Booz, Allen & Hamilton, Inc. |
| Blum, Bruce I. | Applied Physics Lab |
| Bobzien, Gale | Grumman |
| Boehm, Barry W. | DARPA/ISTO |
| Boger, Jacqueline | Computer Sciences Corp. |
| Bond, Roy | DoD |
| Bongianino, Jeffrey R. | General Dynamics |
| Booth, Eric | Computer Sciences Corp. |
| Bourne, William | American Systems Corp. |
| Bowen, Gregory M. | Computer Sciences Corp. |
| Bowen, John B. | Hughes Aircraft Co. |
| Bowen, Robert G. | Social Security Administration |
| Bowerman, Rebecca E. | Pragma Systems Corp. |
| Boyce, Glenn W. | Grumman |
| Boyce, Mary-Ann | Dimensions International |
| Bozenski, Richard | DoD |
| Bozoki, George J. | Lockheed |
| Bradshaw, Howard | Nav Com Tel Com |
| Brechbiel, Fred | Softech, Inc. |
| Bredeson, Mimi | Space Telescope Science Institute |
| Bredeson, Richard W. | Omitron, Inc. |

## FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES
(Continued)

| | |
|---|---|
| Briand, Lionel | University of Maryland |
| Briggs, A. R. | SRC |
| Bristow, John | NASA/GSFC |
| Brophy, Carolyn | Naval Research Lab |
| Brown, Gerald R. | U .S. Army CECOM |
| Brown, James W. | Jet Propulsion Lab |
| Bruno, Kristin J | Jet Propulsion Lab |
| Bunch, Aleda | Social Security Administration |
| Bush, Marilyn | Jet Propulsion Lab |
| Button, Janice | DoD |
| Buys, Ruth | Mitre Corp. |
| Cake, Spencer C | HQ USAF/SCXS |
| Caldiera, Gianluigi | University of Maryland |
| Caplan, Lawrence C. | Hughes Aircraft Co. |
| Carra, Debbie | Computer Sciences Corp. |
| Carter, Bradley D | Mississippi State University |
| Cary, John | George Washington University |
| Casucci, Dan | IRS |
| Cernosek, Gary J | McDonnell Douglas |
| Chen, Andrew J | |
| Chmura, Louis J | Naval Research Lab |
| Chu, Richard | Ford Aerospace Co. |
| Church, Vic | Computer Sciences Corp. |
| Cichowicz, Diana L. | Social Security Administration |
| Clark, James D. | Naval Surface Weapons Center |
| Coates, Ann | Social Security Administration |
| Colberg, John | General Dynamics |
| Conover, Robert A. | Jet Propulsion Lab |
| Cook, John F | NASA/GSFC |
| Cooke, Vic | Loral Aerosys |
| Copps, Stephen L. | Intermetrics, Inc. |
| Cordrey, Glen | Loral Aerosys |
| Cover, Donna | Computer Sciences Corp. |
| Creecy, Rodney | Hughes Aircraft Co. |
| Crehan, Dennis | Ford Aerospace Co. |
| Creswell, Doug | DoD |
| Cuesta, Ernesto | Computer Sciences Corp. |
| D'Elia, Barbara | IRS |
| Dahmen, Steve | Vigyan Research, Inc. |
| Daney, William E. | NASA/GSFC |
| Daskalantonakis, Michael K. | Motorola, Inc. |
| DeMaio, Louis | NASA/GSFC |
| Decker, William | Computer Sciences Corp. |
| Diaz-Herrera, Jorge L. | George Mason University |
| Dikel, David | Focused Ada Research |
| Dirks, John | Computer Sciences Corp. |
| Diskin, David | Contel Technology Center |
| Do, Michael T. | Computer Sciences Corp. |
| Dortenzo, Don | Fairchild Space Co. |
| Dortenzo, Megan | Loral Aerosys |
| Dowen, Andrew | Jet Propulsion Lab |

## FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES
(Continued)

| | |
|---|---|
| Druffel, Larry | Software Engineering Institute |
| DuVall, Lorraine | Duvall Computer Technologies, Inc. |
| Dulaney, Gilbert | Social Security Administration |
| Duncan, Scott P. | BELLCORE |
| Dungan, Larry | Computer Sciences Corp. |
| Duniho, Mickey | DoD |
| Duttine, Valerie | NASA/GSFC |
| Dyer, Michael | IBM |
| Earl, Michael | Intermetrics, Inc. |
| Easterling, Sue | IBM |
| Ebersberger, Marc | Social Security Administration |
| Edlund, Sheryl J | USAISSOCW |
| Edwards, John | IIT Research Institute |
| El-Sahragty, Ahmed | Computer Technology Associates, Inc. |
| Ellis, Walter | IBM |
| Elwadhi, Manisha | Computer Sciences Corp. |
| Emery, Richard D | Vitro Corp. |
| Emig, Pamela W | Logicon, Inc. |
| Engelmeyer, William J | Computer Sciences Corp. |
| Esker, Linda | Computer Sciences Corp. |
| Ferg, Stephen | Stephen Ferg Associates |
| Fessler, Jim | Loral Aerosys |
| Fink, Mary Louise A | EPA |
| Forsythe, Ron | NASA/Wallops Flight Facility |
| Fosaaen, Ardyth | IRS |
| Fox, Raymond | DoD |
| Franklin, Carla B | Lockheed |
| Franklin, Jude E | Planning Research Corp. |
| Franks, Kelly A | NASA/GSFC |
| Frawley, Joanna | SOHAR, Inc. |
| Fridge, Ernie | NASA/JSC |
| Friend, Gregg | Computer Sciences Corp. |
| Frizzell, Jim | Rockwell International |
| Futcher, Joseph M | Naval Surface Weapons Center |
| Gaither, Melissa | CRMI |
| Gallagher, Barbara | DoD |
| Galloway, Densel | Bendix Field Engineering Corp. |
| Garcia, Enrique A | Jet Propulsion Lab |
| Geil, Esther | Westinghouse |
| Gilliland, Denise E | Stanford Telecommunications, Inc. |
| Girard, Pat | IBM |
| Glass, Robert L | Computing Trends |
| Godfrey, Sally | NASA/GSFC |
| Goel, Amrit L | Syracuse University |
| Goldberg, Nancy | Computer Sciences Corp. |
| Golden, John R | Rensselaer Learning Systems, Inc. |
| Goldsmith, Larry | Bureau of Labor Statistics |
| Goldstein, Steven | IIT Research Institute |
| Gordon, Hayden H | Computer Sciences Corp. |
| Gordon, Judith J | Pragma Systems Corp. |

## FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES
(Continued)

| | |
|---|---|
| Gordon, Marc D. | Fairchild Space Co. |
| Graddy, John | DoD |
| Grady, Robert | Hewlett Packard |
| Graham, Marcellus | NASA/MSFC |
| Grant, Ralph | New Technology, Inc. |
| Grasso, Barry | Ford Aerospace Co. |
| Graves, Russell J | DoD |
| Green, Daniel R. | GSA/FOAC |
| Green, David | Computer Sciences Corp. |
| Green, Scott | NASA/GSFC |
| Gregory, John G. | Westinghouse |
| Grier, Jackie | Colsa, Inc. |
| Grondalski, Jean | Computer Sciences Corp. |
| Gross, Stephen | Naval Center for Cost Analysis |
| Groveman, Brian S | Computer Sciences Corp. |
| Groves, Paula | Computer Sciences Corp. |
| Groves, Robert T. | NASA/GSFC |
| Guillebeau, Michael | TRW |
| Guzek, Joseph | Hughes Aircraft Co. |
| Gwynn, Thomas R. | Computer Sciences Corp. |
| Haas, Michael | Naval Sea Systems Command |
| Hall, Jr., John | General Dynamics |
| Halterman, Karen | NASA/GSFC |
| Hamilton, John R. | FAA |
| Hamstreet, Roger A | USAF |
| Handley, Thomas H. | Jet Propulsion Lab |
| Hankins, Dick | General Dynamics |
| Harrison, Carolyn | Fairchild Space Co. |
| Hashmi, Awais | Digital Systems |
| Hawkins, George | FBI |
| Heasty, Richard | Computer Sciences Corp. |
| Heffernan, Henry G. | EDP News Services |
| Heller, Gerry | Computer Sciences Corp. |
| Hendrick, Robert B. | Computer Sciences Corp. |
| Herman, Lorrie | Fairchild Space Co. |
| Herrin, Ben | Dynamics Research Corp. |
| Heusinger, Hugo | MBB Munich |
| Hill, Carroll R. | IRS |
| Holland, Denny | IIT Research Institute |
| Holmes, Barbara | CRMI |
| Holt, Nancy | Social Security Administration |
| Hon, Samuel E. | BELLCORE |
| Hood, Thomas L. | STX Corp. |
| Hormby, Tom W. | Johns Hopkins University |
| Hosler, George | OAO |
| Houston, Frank | FDA |
| Hryn, Ed | OAO |
| Hunter, Paul | NASA |
| Hurt, Tom | Grumman |
| Jay, Elizabeth | NASA/GSFC |
| Jeane, Shirley | Jet Propulsion Lab |

## FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES
(Continued)

| | |
|---|---|
| Jeletic, Jim | NASA/GSFC |
| Jenkins-Bnafa, Jovita | TRW |
| Joesting, David | Bendix Field Engineering Corp. |
| Jordano, Tony J | IBM |
| Kardatzke, Owen C | NASA/GSFC |
| Karlin, Jay | Project Engineering, Inc. |
| Keil-Slawik, Reinhard | University of Maryland |
| Kelly, John C | Jet Propulsion Lab |
| Kelly, Kim R | IBM |
| Kemp, Kathryn M | NASA/HQ |
| Kennedy, Elizabeth A | Rockwell International |
| Kester, Rush | Computer Sciences Corp. |
| Kile, Thomas | Dept. of the Army |
| Kim, Robert D | Computer Sciences Corp. |
| Kimminau, Pamela S | DoD |
| Kishan, Sushma | Stanford Telecommunications, Inc. |
| Kistler, David M | Computer Sciences Corp. |
| Kleis, Karen | Computer Sciences Corp. |
| Koerner, Kathy | Computer Sciences Corp. |
| Koropka, Joseph J | Computer Sciences Corp. |
| Kouchakdjian, Ara | Software Engineering Technology |
| Krinsky, Neal | Computer Sciences Corp. |
| Kuhn, Rick | National Institute of Standards & Tech. |
| Kulik, Constance E | Mitre Corp. |
| Kurihara, Tom | Dept. of Transporation |
| Landis, Linda | Computer Sciences Corp. |
| Largent, Jim | Loral Aerosys |
| Larman, Brian | Jet Propulsion Lab |
| LaVallee, David | Ford Aerospace Co. |
| Lawrence-Pfleeger, Shari | Contel Technology Center |
| Layne, Eva | NASA/HQ |
| Lehman, Manny M | Imperial College |
| Levitt, David S | Computer Sciences Corp. |
| Lin, Chi Y | Jet Propulsion Lab |
| Lippert, Richard | Grumman |
| Liskey, John W | Social Security Administration |
| Liu, Jean C | Computer Sciences Corp. |
| Liu, Kuen-san | Computer Sciences Corp. |
| Loesh, Bob E | Jet Propulsion Lab |
| Lohfeld, Robert | OAO |
| Lott, Chris M | University of Maryland |
| Lowrey, Carla | DoD |
| Luczak, Ray | Computer Sciences Corp. |
| Lydon, Tom | Raytheon |
| Lynch, Craig | USAISSC |
| Lyons, Peter | Hughes Aircraft Co. |
| Maddock, Karen R | Technology Planning, Inc. |
| Mahal, Becky A | Mitre Corp. |
| Maher, Stephen | NASA/GSFC |
| Malthouse, Nancy | Logicon, Inc. |

FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES
(Continued)

| | |
|---|---|
| Margono, Johan | Computer Sciences Corp. |
| Marinaro, Patricia M. | Coopers & Lybrand |
| Marshall, Dale | DoD |
| Mathiasen, Candy | Unisys |
| Matusow, David | NASA/GSFC |
| Maury, Jesse | NASA/GSFC |
| Mazzola, Ray | Loral Aerosys |
| McCabe, Richard S. | Software Productivity Consortium |
| McCauley, Robert | Martin Marietta |
| McConnell, Dave | McDonnell Douglas |
| McDermott, Tim | Computer Sciences Corp. |
| McDonald, Beth | DoD |
| McGarry, Frank | NASA/GSFC |
| McGarry, Peter | General Electric |
| McGowan, Clement | Contel Technology Center |
| McGraw, Diane | DoD |
| McHenry, Ronald C. | Stanford Telecommunications, Inc. |
| McLaughlin, Byron | IBM |
| Mehler, Steve | IIT Research Institute |
| Meick, Douglas | Library of Congress |
| Mendelsohn, Chad | NASA/GSFC |
| Merifield, James | Advanced Technology, Inc. |
| Merry, Paul | Harris Corp. |
| Mickel, Susan | General Electric |
| Miller, Andy | Loral Aerosys |
| Miller, Don | OAO |
| Miller, John | Computer Sciences Corp. |
| Mills, Harlan | Software Engineering Technology |
| Mingee, Lynn U | Lockheed |
| Minninger, John R. | DoD |
| Mitchell, Michael S. | Computer Sciences Corp. |
| Mohrman, Carl C. | Martin Marietta |
| Moleski, Laura | CRMI |
| Montgomery, Helen | Intermetrics, Inc. |
| Moore, Cal | Loral Aerosys |
| Moore, Paula | Loral Aerosys |
| Morusiewicz, Linda M. | Computer Sciences Corp. |
| Murray, William M. | General Dynamics |
| Myers, Philip I. | Computer Sciences Corp. |
| | |
| Nakagiri, Howard T. | Hughes Aircraft Co. |
| Naleszkiewicz, John | Technology Planning, Inc. |
| Nance, Richard E. | Virginia Tech University |
| Narrow, Bernie | Bendix Field Engineering Corp. |
| Ng, Susan | Spar Aerospace Lmtd. |
| Nolan, Sandra K. | Lockheed |
| Norcio, Tony F. | University of Maryland |
| Nutting, Bruce | Unisys |
| | |
| O'Neill, Lawrence A. | AT&T Computer Systems |
| Odt, Thomas A. | Computer Sciences Corp. |
| Ohba, Mitsuru | Tokyo Research Laboratory |
| Ohlmacher, Jane | Social Security Administration |

## FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES
(Continued)

| | |
|---|---|
| Oivo, Markleu | University of Maryland |
| Osman, Mohamed | Jet Propulsion Lab |
| Packett, Lisa | Census Bureau |
| Page, Gerald | Computer Sciences Corp. |
| Pajerski, Rose | NASA/GSFC |
| Panzer, David | Stanford Telecommunications, Inc. |
| Pappas, Eugene | Stanford Telecommunications, Inc. |
| Park, Robert | Computer Sciences Corp. |
| Paules, David R. | |
| Pavnica, Paul | Census Bureau |
| Pearson, Boyd | NASA/GSFC |
| Pecore, Joseph N. | Vitro Corp. |
| Perez, Frank | Unisys |
| Perkins, Dorothy | NASA/GSFC |
| Petranka, Frank J. | Naval Surface Weapons Center |
| Pettijohn, Margot | IRS |
| Petty, Judy | Computer Sciences Corp. |
| Pincosy, John F. | Data Systems Analysis, Inc. |
| Plett, Michael E. | Computer Sciences Corp. |
| Polly, Mike | Raytheon |
| Potter, Marshall R. | Dept. of the Navy |
| Preston, David | The Catholic University of America |
| Prince, Andy | Planning Research Corp. |
| Proctor, Martina | DoD |
| Provenza, Clint | Booz, Allen & Hamilton, Inc. |
| Pumphrey, Karen | Computer Sciences Corp. |
| Quann, Eileen S. | Fastrak Training, Inc. |
| Rahmani, D. | Computer Sciences Corp. |
| Randolph, J. C. | Martin Marietta |
| Randolph, Lynwood P. | NASA/HQ |
| Raney, Dale L. | Eagle Systems Inc. |
| Reed, Teresa S. | Mitre Corp. |
| Reid, Diane | Grumman |
| Reifer, Don J. | Reifer Consultants, Inc. |
| Repsher, Marie | IRS |
| Reynolds, Clarence M. | Mitre Corp. |
| Rifkin, Stan | Master Systems Inc. |
| Riggs, Bruce | IRS |
| Rinearson, Linda | GTE |
| Roberts, Becky L. | Oracle Complex Systems Corp. |
| Roberts, Charles R. | TRW |
| Robillard, Pierre N. | University of Montreal |
| Roe, Bob L. | Boeing Aerospace Co. |
| Rohr, John A. | Jet Propulsion Lab |
| Rombach, Dieter H. | University of Maryland |
| Rone, Kyle Y. | IBM |
| Roselius, Tom | IRS |
| Roy, Dan M. | Software Engineering Institute |
| Roy, Robert C. | General Electric |
| Royce, Walker | TRW |

FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES
(Continued)

| | |
|---|---|
| Rumerman, Howard | DoD |
| Ryan, Bob | IBM |

| | |
|---|---|
| Sahoo, Swarupa N. | Syracuse University |
| Samadani, Hamid | Loral Aerosys |
| Sandford, James W. | IBM |
| Santiago, Richard | Jet Propulsion Lab |
| Sava, Monica | Ford Aerospace Co. |
| Savolaine, Catherine G | AT&T Bell Lab |
| Sears, Barry | IBM Canada Ltd. |
| Seaver, David P. | Project Engineering, Inc. |
| Segal, Jeffrey H. | NASA/GSFC |
| Seidewitz, Ed | NASA/GSFC |
| Severino, Tony E. | General Electric/RCA |
| Shammas, Barbara | IRS |
| Sheckler, John D. | Bendix Field Engineering Corp. |
| Shekarchi, John | Stanford Telecommunications, Inc. |
| Shell, Allyn M. | Information Dynamics, Inc. |
| Shirah, Greg | NASA/GSFC |
| Shoan, Wendy | NASA/GSFC |
| Siebenthall, Bruce A. | FAA |
| Siegel, Karla | Mitre Corp. |
| Silberberg, David | DoD |
| Singer, Carl A. | BELLCORE |
| Singhal, Sushma | Stanford Telecommunications, Inc. |
| Smith, Brian | Loral Aerosys |
| Smith, Carl | Naval Surface Weapons Center |
| Smith, Cassandra M. | Mitre Corp. |
| Smith, Donald | NASA/GSFC |
| Smith, Elizabeth J. | Computer Sciences Corp. |
| So, Maria | Computer Sciences Corp. |
| Solomon, Carl | ST Systems Corp. |
| Sparmo, Joe | NASA/GSFC |
| Spence, Bailey | Computer Sciences Corp. |
| Spencer, Mike | Naval Surface Weapons Center |
| Spiegel, Doug | NASA/GSFC |
| Spiegel, Jim | Loral Aerosys |
| Sporn, Patricia A. | NASA/HQ |
| Squire, Jon S. | Westinghouse |
| Squires, Burton E. | Mnemonic Systems Inc. |
| Stackhouse, Will | JPL/U.S. Air Force |
| Stampfl, Sue | Booz, Allen & Hamilton, Inc. |
| Stark, Michael | NASA/GSFC |
| Steinbacher, Jody | NASA/JPL |
| Steube, Jerry | Technology Planning, Inc. |
| Stewart, Debra | Computer Sciences Corp. |
| Strano, Caroline | FAA |
| Straub, Pablo | University of Maryland |
| Sun, Pamela | AT&T Bell Lab |
| Swain, Barbara | University of Maryland |
| Szot, Stephen J. | Software Productivity Consortium |
| Szulewski, Paul | C. S. Draper Labs, Inc. |

## FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES
(Continued)

| | |
|---|---|
| Tai, K. C. | National Science Foundation |
| Tasaki, Keiji | NASA/GSFC |
| Tavassoli, Naz | Computer Sciences Corp. |
| Taylor, Robert E | Social Security Administration |
| Terry, Bradlee | Terry Consultants |
| Terry, Georgene | OAO |
| Thackrey, Kent | Planning Analysis Corp. |
| Thomas, Donna | Computer Sciences Corp. |
| Thomas, William | Mitre Corp. |
| Thompson, Charles N | FAA Technical Center |
| Thornton, Thomas | NASA/JPL |
| Tominovich, Gary | Intermetrics, Inc. |
| Tran, Lan T | Jet Propulsion Lab |
| Ulery, Bradford | University of Maryland |
| Ulrich, Carol | Hughes Aircraft Co. |
| Valett, Jon | NASA/GSFC |
| Valett, Susan | NASA/GSFC |
| Van Meter, David | Logicon, Inc. |
| Vaughan, Joe | Social Security Administration |
| Verbeck, Joan | NASA/HQ |
| Vladavsky, Luba | Logicon, Inc. |
| Voigt, David | Bendix Field Engineering Corp. |
| Vuolo, Bob | NASA/JPL |
| Wade, David M | Computer Sciences Corp. |
| Waligora, Sharon R | Computer Sciences Corp. |
| Walker, Carolyn V | IBM |
| Walker, Ron | IBM |
| Wallace, Dolores | National Institute of Standards & Tech. |
| Wartik, Steven | Software Productivity Consortium |
| Weidow, David | NASA/GSFC |
| Weisman, David | Unisys |
| Weiss, Dave | Software Productivity Consortium |
| Weldon, Karen M | General Electric |
| Weszka, Joan | IBM |
| Wheeler, J. L. | Computer Sciences Corp. |
| Whitesell, Steven A | Computer Sciences Corp. |
| Wilbert, Carl K | NASA/HQ |
| Williams, Roger | Software Productivity Consortium |
| Williamson, Phillip | Boeing Computer Support Services Co. |
| Wilson, Dyna | FBI |
| Wilson, William M | Information Dynamics, Inc. |
| Witzgall, W. K | The Arinc Companies |
| Wolfe, Dan | Hughes Aircraft Co. |
| Wolverton, Ray | Hughes Aircraft Co. |
| Wood, Dick | Computer Sciences Corp. |
| Wood, James M | Siemans Corporate Research |
| Yaramanoglu, Melih | Logicon, Inc. |
| Yee, Mary | Mitre Corp. |
| Youman, Charles | Cey Enterprises |

## FIFTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES
(Continued)

| | |
|---|---|
| Young, Andy | Bendix Field Engineering Corp. |
| Zavage, Jerry | Computer Sciences Corp. |
| Zaveler, Saul | U.S. Air Force |
| Zelkowitz, Marv | University of Maryland |
| Zhou, Xiaodong | University of Maryland |
| Zimet, Beth | Computer Sciences Corp. |
| Zoch, David | Loral Aerosys |
| Zygielbaum, Art | Jet Propulsion Lab |

# APPENDIX B – SEL BIBLIOGRPHY

# 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-77-004, *A Demonstration of AXES for NAVPAK*, M. Hamilton and S. Zeldin, September 1977

SEL-77-005, *GSFC NAVPAK Design Specifications Languages Study*, P. A. Scheffer and C. E. Velez, October 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 and W. A. Taylor, July 1986

SEL-79-002, *The Software Engineering Laboratory: Relationship Equations*, K. Freburger and V. R. Basili, May 1979

SEL-79-003, *Common Software Module Repository (CSMR) System Description and User's Guide*, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 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-80-003, *Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study*, T. Welden, M. McClellan, and P. Liebertz, 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-81-008, *Cost and Reliability Estimation Models (CAREM) User's Guide*, J. F. Cook and E. Edwards, February 1981

SEL-81-009, *Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation*, W. J. Decker and F. E. McGarry, March 1981

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 From 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-81-104, *The Software Engineering Laboratory*, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-107, *Software Engineering Laboratory (SEL) Compendium of Tools*, W. J. Decker, W. A. Taylor, and E. J. Smith, February 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-81-205, *Recommended Approach to Software Development*, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

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 From 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, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-906, *Annotated Bibliography of Software Engineering Laboratory Literature*, P. Groves and J. Valett, November 1990

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-83-006, *Monitoring Software Development Through Dynamic Variables*, C. W. Doerflinger, November 1983

SEL-83-007, *Proceedings From 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-84-101, *Manager's Handbook for Software Development, Revision 1*, L. Landis, F. McGarry, S. Waligora, et al., November 1990

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 From the Ninth Annual Software Engineering Workshop*, November 1984

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., May 1985

SEL-85-005, *Software Verification and Testing*, D. N. Card, C. Antle, and E. Edwards, December 1985

SEL-85-006, *Proceedings From 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 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, October 1986

SEL-86-006, *Proceedings From 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-87-008, *Data Collection Procedures for the Rehosted SEL Database*, G. Heller, October 1987

SEL-87-009, *Collected Software Engineering Papers: Volume V*, S. DeLong, November 1987

SEL-87-010, *Proceedings From 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-89-003, *Software Management Environment (SME) Concepts and Architecture*, W. Decker and J. Valett, August 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-89-101, *Software Engineering Laboratory (SEL) Database Organization and User's Guide (Revision 1)*, M. So, G. Heller, S. Steinberg, K. Pumphrey, and D. Spiegel, February 1990

SEL-90-001, *Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide*, M. Buhler and K. Pumphrey, 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. J. Decker, R. Hendrick, and J. Valett, February 1991

## SEL-RELATED LITERATURE

[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

[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]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

[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

[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

[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

Basili, V. R., and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

[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 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., 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., and H. D. Rombach, "T A M E: Tailoring an Ada Measurement Environment," *Proceedings of the Joint Ada Conference*, March 1987

[5]Basili, V., and H. D. Rombach, "T A M E: 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

[2]Basili, V. R., R. W. Selby, Jr., and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983

[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, Jr., *Comparing the Effectiveness of Software Testing Strategies*, University of Maryland, Technical Report TR-1501, May 1985

[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

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, July 1986

[5]Basili, V., and R. Selby, Jr., "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[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

[5]Brophy, C., W. Agresti, and V. 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., and W. Agresti, "Resolving the Software Science Anomaly," *The Journal of Systems and Software*, 1987

[6]Card, D. N., and W. Agresti, "Measuring Software Design Complexity," *The Journal of Systems and Software*, June 1988

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

[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., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D., F. McGarry, and G. 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

[5]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

Hamilton, M., and S. Zeldin, *A Demonstration of AXES for NAVPAK*, Higher Order Software, Inc., TR-9, September 1977 (also designated SEL-77-005)

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

[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., and 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

National Aeronautics and Space Administration (NASA), *NASA Software Research Technology Workshop* (Proceedings), March 1980

[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

[5]Ramsey, C., and V. R. Basili, *An Evaluation of Expert Systems for Software Engineering Management*, University of Maryland, Technical Report TR-1708, September 1986

[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

[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

[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

[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

[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

[8]Stark, M., "On Designing Parametrized Systems Using Ada," *Proceedings of the Seventh Washington Ada Symposium*, June 1990

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," *Proceedings of TRI-Ada 1989*, October 1989

Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," *Proceedings of the Joint Ada Conference*, March 1987

[8]Straub, P. A., and M. 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

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

Turner, C., G. Caron, and G. Brement, *NASA/SEL Data Compendium, Data and Analysis Center for Software*, Special Publication, April 1981

[5]Valett, J., and F. 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. 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: Experience With Syntax Editors," *Information and Software Technology*, April 1990

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

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.

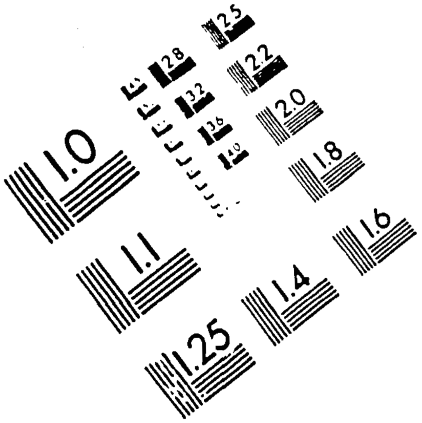
[8] This article also appears in SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990.

END

DATE

FILMED

MAR 31 1992

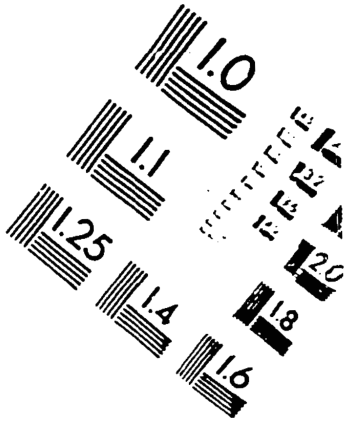


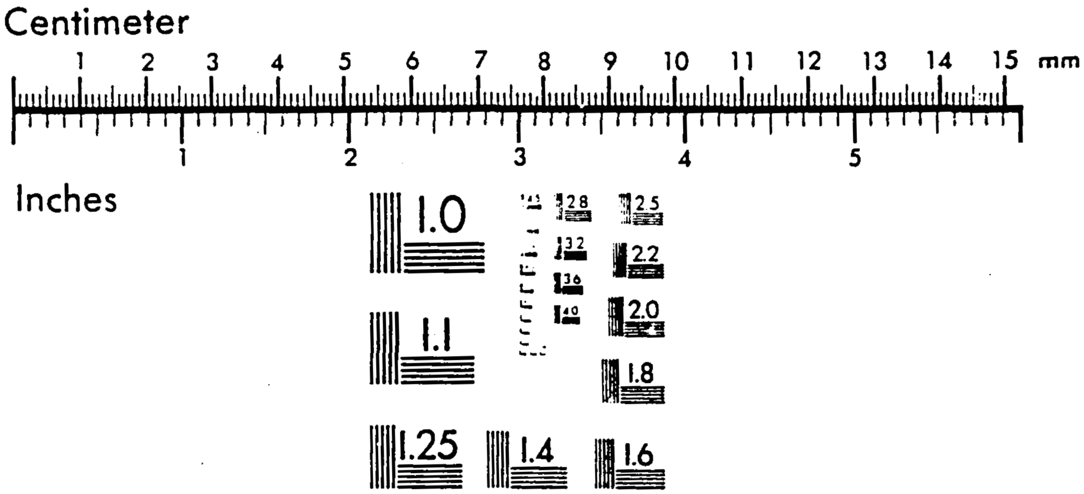
AIIM

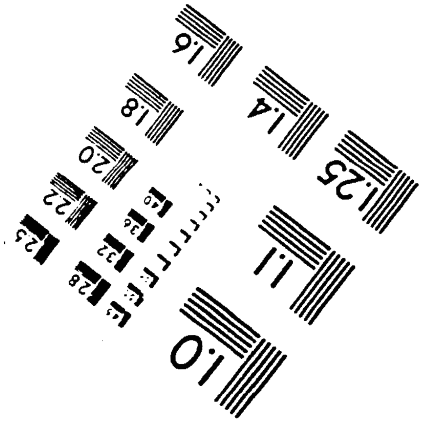
**Association for Information and Image Management**

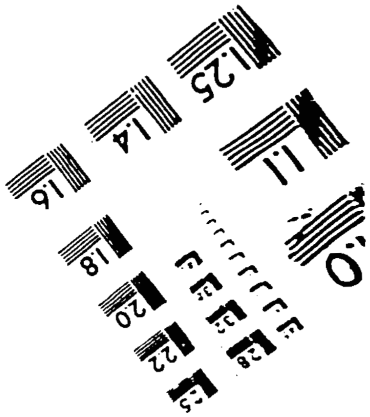
1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910

301/587-8202









MANUFACTURED TO AIIM STANDARDS
BY APPLIED IMAGE, INC.