Proceedings of the Eighteenth Annual Software Engineering Workshop

December 1-2, 1993

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

FOREWORD

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

NASA/GSFC, Software Engineering Branch

The University of Maryland, Department of Computer Science

Computer Sciences Corporation, Software Engineering Operation

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

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

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

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

CONTENTS

Materials for each session include a summary of the live presentation and selected questions and answers, as well as any viewgraphs, abstracts, or papers submitted for inclusion in these *Proceedings*.

Page

0117 70
P39

SUMMARY OF THE EIGHTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

The Eighteenth Annual Software Engineering Workshop, sponsored by the Software Engineering Laboratory (SEL), was held on 1 and 2 December 1993 at the National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center in Greenbelt, Maryland.

The Workshop provides a forum for software practitioners from around the world to exchange information on the measurement, use, and evaluation of software methods, models, and tools. This year, approximately 450 people attended the Workshop, which consisted of six sessions on the following topics: The Software Engineering Laboratory, Measurement, Technology Assessment, Advanced Concepts, Process, and Software Engineering Issues in NASA. Three presentations were given in each of the topic areas. The content of those presentations and the research papers detailing the work reported are included in these *Proceedings*. The Workshop concluded with a tutorial session on how to start an Experience Factory. (Copies of those slides were distributed at the Workshop.)

In his introductory remarks, Frank McGarry of NASA/Goddard provided the audience with a background of the SEL organization and set the stage for the Session 1 presentations on the SEL as well as for other presentations that were derived from SEL experiences.

The SEL is a virtual organization that includes NASA/Goddard's Software Engineering Branch, the University of Maryland's Department of Computer Science, and the Computer Sciences Corporation's Software Engineering Operation. The primary goals of the SEL, as stated by McGarry, are to better understand software and to determine the impact that changing technologies have on the way an organization goes about the business of producing software. Every project that is developed in the SEL's parent organization, the Flight Dynamics Division (FDD) of NASA/Goddard, is a potential experiment where the SEL can study some technology and measure the impact that this technology has on some specific goals within the organization. In this way, the SEL can determine the appropriate use of technologies in its environment.

McGarry pointed out that the concept of learning from experience in a production environment has been modeled as the "Experience Factory" and has been written about extensively by Dr. Victor Basili, a cofounder of the SEL. The experience factory is, in essence, a well-defined model or paradigm of how one learns from one's own experiences, sets goals, measures results, and then uses these results to continue to change, evolve, and mature better ways of doing business.

One of the key insights that the experience factory has brought to light, McGarry asserted, is that not all software is the same. You have to understand your own domain and your own organization before making changes and expecting to make improvements. You have to understand what your own goals, strengths, weaknesses, processes, products, people, and problems are before you can conclude what particular technologies are most appropriate to embrace to make measurable improvements in your organization.

McGarry then showed the audience some of the characteristics of the projects that the SEL has studied since 1976. All of the projects were in the flight dynamics domain. He explained that the point of showing the characteristics of the FDD environment was twofold: first, to explain that the results that the audience would hear during the SEL workshop presentations were based on experiments in this particular environment; and second, to note that the SEL findings may not be directly applicable to other environments. McGarry emphasized that the process—the experience factory concept (of continual process improvement)—was indeed applicable, but that the results, the specific technologies they found to be appropriate, were domain specific. "You have to find those [effective technologies] in your own particular domain," McGarry said.

Next, McGarry briefly described the basic improvement model or paradigm that the SEL uses, which consists of three iterative phases: understanding, assessing, and packaging. Understanding is where the SEL strives to better understand the environment, including the domain, tools, products, and existing processes. In the assessment phase, the SEL examines the impact that technologies have on the environment as measured against predetermined goals, such as decreased cost or increased reliability. McGarry stressed that you can only assess when you're measuring against something you understand. That's why the understanding phase must come first. The third phase of the paradigm is packaging. In this phase, the SEL captures the favorable experiences (technologies and processes) and infuses them back into the organization on a broader level, through standards, policies, training, tools, and environments.

McGarry noted that over the course of the workshop the audience would hear several speakers talk about data and information extracted from the SEL environment. He then introduced Vic Basili, who expanded on the discussion of the SEL process improvement model.

Session 1: The Software Engineering Laboratory

THE MATURING OF THE QUALITY IMPROVEMENT PARADIGM IN THE SEL

presented by Victor Basili, University of Maryland

Vic Basili presented an historical account of how the SEL quality improvement paradigm (QIP) first came into being and how it has evolved through the years. He explained what activities were performed and what lessons were learned during several 5-year intervals from the SEL's inception in 1976 to the present, and then offered a glimpse of where SEL research is headed in the near future.

Basili began by sharing some of the false starts the SEL has made (e.g., trying to assess before understanding, collecting data before establishing goals); how the SEL has learned from these false starts; and how, when the SEL verified by experimentation that a particular technology or process was effective, the experience was packaged and infused into the organization. Basili shared some of the philosophy of the SEL (discover what works and then apply it) and explained the interdependent relationship between the SEL's NASA, University of Maryland, and Computer Sciences Corporation partners.

Next, Basili presented the current QIP. This paradigm is a more detailed model of McGarry's understand-assess-package paradigm. The expanded model consists of six steps: (1) characterize the current project and environment; (2) set quantifiable goals; (3) choose the appropriate process model and supporting methods and tools; (4) execute the processes and collect data; (5) analyze the data to evaluate current practices; and (6) package the experience. Basili then spoke briefly about how those particular steps have evolved before he launched into his historical overview.

1976—1980. The SEL started out by looking at other people's models (e.g., the Raleigh curve, TRW's 40/20/40 rule for designing, coding, and testing software) to determine whether these models applied to the SEL environment. Data collection forms were developed, existing processes were combined, and the process was studied. The "goal-question-metric" paradigm was created to enable the researchers to organize their data around particular studies. Data collection was loosely monitored, baselines built, correlations sought, and information was recorded.

The SEL learned that it needed to better understand the environment, projects, processes, and products; it needed to build its own models based on its own environment and products; it needed to understand the factors that create similarities and differences among projects so that it would know the appropriate model to apply; and the SEL needed to understand how to choose the right processes to create the desired product characteristics. Basili emphasized that a big lesson learned was that data collection has to be goal driven; you can't just collect data and then figure out what to do with them.

1981—1985. During this period, the SEL was building its own baselines and models of cost, defects, and processes. It began to set specific goals in multiple areas and began to incorporate subjective metrics into its measurement process. The SEL experimented with well-defined technologies and began experiments with potentially high-impact technologies such as Ada and object-oriented design (OOD). The SEL collected fewer data (because collection was more goal driven) and stored these data in a relational database. It shifted its emphasis to the relationship between process and product characteristics; recorded lessons learned; and began formalizing process, product, knowledge, and quality models. In other words, the SEL began to package its own experience. At this time the SEL developed its first recommended QIP: characterize, set goals, choose the process, execute, analyze, and record.

From these activities, the SEL learned many lessons. It discovered that software development follows an experimental paradigm; i.e., designing experiments is an important part of improving, and evaluation and feedback are necessary for learning. It discovered a need to experiment with new technologies and to learn about the relationships between process, product, and quality models. It learned that it could multiply its effectiveness by reusing experience in the form of processes and other forms of knowledge. And it learned that you can drown in too much data; data must be selectively collected to support well-defined goals.

1986—1990. During the late 1980s, the SEL worked on capturing experience in models; it built models that enabled it to differentiate between environments. Goals and models became the driver for measurement. The SEL built the Software Management Environment (SME), an intelligent tool that can access the SEL database and produce graphical models of various factors for dozens of projects. The SEL tailored and evolved its use of Ada and OOD based on experience. Experience and feedback were recognized as an integral part of the QIP. The SEL embedded data into the processes and began to closely monitor study projects, and it demonstrated relationships between processes and products, based on extensive analysis. The SEL made the transition from simply recording information, to packaging information in the form of focused tailored packages. The SEL began to learn how to transfer technology better through organizational structure, experimentation, and evolutionary culture change.

Lessons learned during this period include the following: experience needs to be evaluated, tailored, and packaged for reuse; there is a tradeoff between reuse and improvement (because introducing change causes a loss in experience and predictability); and software processes must be put into place to support the reuse of experience. The SEL learned that packaged experiences need to be integrated, i.e., that there is an appropriate way for processes to work together (for example, code reading is more effective when not followed by testing, as in Cleanroom methodology). As a result of these findings, the QIP was reformulated as: characterize, set goals, choose process, execute, analyze, and package. It was at this time that the SEL organization and activities became formalized under the "Experience Factory" title.

1991—1995. In this period, the SEL is continuing to build relationship models between processes and products. It is performing domain analysis studies to identify similar projects using techniques appropriate for software engineering data. The SEL plans to automate the model-based goal-question-metric as much as possible. The SEL is developing technologies tailorable to specific project needs (such as tailored reading techniques). It is building a more powerful, flexible experience base by increasing and improving the interaction between developers and experimenters to enable the capture of more effective feedback. Today, the SEL is clearly focused on local needs and goals. The SEL is learning how to run more efficient experiments and combine controlled experiments with case studies. And, through an empirical modeling technique known as optimized set reduction, the SEL is building better models for cost modeling and defective module prediction.

Basili concluded by saying that the SEL has come a long way! A great deal has been learned about software improvement, and the learning process has been continuous and evolutionary. The SEL has packaged what it has learned into its process, product, and organizational structure. This evolution is supported by a strong relationship between research and practice. Finally, Basili noted, the relationship known as the SEL continues to require patience and understanding on both sides of the research and practice continuum, and when this relationship is nurtured and protected, it pays measurable dividends.

PROCESS IMPROVEMENT AS AN INVESTMENT: MEASURING ITS WORTH

presented by Frank McGarry, NASA/Goddard

In this presentation Frank McGarry compared the Software Engineering Institute (SEI) Capability Maturity Model (CMM) to the SEL process improvement model. He looked at steps required to compute return on investment (ROI), examined information needed to determine ROI, and illustrated how investment in process improvement has affected process and products in the SEL environment.

McGarry began his comparison of the CMM and the SEL process improvement model by quickly explaining the concepts behind each and identifying what he sees as the fundamental difference between the two approaches. In the CMM, an organization evolves through five levels of maturity that reflect its capability to produce complex software. The levels are (1) ad hoc, (2) repeatable, (3) defined, (4) managed, and (5) optimizing. In the SEL model, the process evolves from understanding the processes being used and the products being developed, to assessing the impact of process changes, to packaging those processes that have a positive impact on measurable product characteristics. Although both models have the same goal of improving the product, the fundamental difference between the two paradigms, McGarry asserted, was that the CMM primarily focuses on the process level of an organization (such as the standards and techniques used), whereas SEL focuses on attributes of the product, such as error rates, productivity, and complexity.

To facilitate a consistent comparison of the two approaches, McGarry then outlined the steps that he would examine for both models in determining the ROI from process improvement: (1) defining goals, (2) producing baselines, (3) investing in change, (4) assessing change, and (5) measuring ROI.

Defining Goals. Both the CMM and SEL emphasize setting goals. The CMM approach is to set a goal to get to a new level of maturity. This general goal is the same for all organizations: to evolve to higher levels of maturity until reaching an optimizing state. The SEL goal, on the other hand, is to improve the product in some particular way that is appropriate to that product. In some organizations, for example, it may be most appropriate to decrease error rates; in another organization it may be more important to try to increase the level of reuse.

Producing Baselines. The second step in determining ROI is to produce a baseline. That is, you must determine where you are with respect to where you aspire to be. In the CMM paradigm, the approach is to assess your process using a "common yardstick" to rate how your organization performs certain key activities such as training, measurement, and defining and following standards. CMM establishes a process baseline. The SEL is also interested in standards, training, and measurement, but what sets the SEL paradigm apart from the CMM paradigm, McGarry asserted, is that the SEL emphasizes both process and product understanding. Furthermore, the measures that the SEL focuses on are specific to a particular project's goals with respect to their domain, people, and environment. For example, if your goal is to reduce error rates, then you baseline error rates; if your goal is to increase reuse, then you baseline your current level of reuse.

To further illustrate measurements characterizing process baselines, McGarry showed pie charts of effort distribution and classes of errors, and line plots of source code growth rates. To exemplify product characteristic baselines, McGarry showed bar charts of error rates, cost, and reuse. In all cases, the measurements were based on actual SEL data. McGarry emphasized that these measurements would be used as qualifiers to determine whether changing the process was having a positive change on the product.

Investing in Change. Using the CMM, the organization's assessed level drives the change. If your organization is assessed as a level 1, then you work on activities that will get you to level 2 before you work on activities to get you to level 3, 4, or 5. In the SEL paradigm, your process improvement activities are driven by your organization's unique experiences and goals. If your experience tells you that programming in Ada does not lead to improved productivity, then you don't mandate Ada. If there is evidence that Cleanroom lowers error rates, and this is important to your organization, then you develop a training program to teach Cleanroom techniques and you experiment with Cleanroom. You invest in changes that you discern will make a positive impact on process efficiency as well as product characteristics.

To exemplify how the SEL has invested in change, McGarry showed how the results of experiments in design techniques, testing approaches, Ada, OOD, Cleanroom, and other experiments have been packaged into training, software measurement guidebooks, and recommended approaches to developing software and managing software development. Again, McGarry emphasized that experience drives process change.

Assessing the Impact of Change. The fourth step in determining ROI is to assess the impact that change has had on the organization. In the CMM paradigm, success is measured by process change: Did you achieve the maturity level you set out to achieve? In the SEL paradigm, you reexamine not only the impact of the process, but also the impact that process change has had on your product. Product and process change in the SEL are very domain-dependent. As an example, McGarry offered evidence of the impact of introducing the Cleanroom technique by showing graphs of the impact of Cleanroom on effort distribution and source code growth rate with Cleanroom.

Measuring the Return on Investment. McGarry explained that the first piece of information needed is the actual cost of investing in process improvement. He began by showing a summary of the kinds of investments the SEL has made in process improvement in four categories: project overhead, including filling out forms, collecting data, and training; data processing, including quality assurance, maintaining a database, and archiving data; analysis and application, including developing processes and standards, defining experiments, and analyzing results; and developing and maintaining mission support software. McGarry noted that the total cost of SEL process improvement activities (the investment) was approximately 11% of all expenditures.

Next, McGarry discussed the return on investment. McGarry noted that an investment in product-driven goals enabled direct measurement of return. He illustrated his point by providing measurement examples, based on changes from the mid 1980s to the early 1990s, that show an increase in product quality and a decrease in product cost: reliability was improved 75%—measured as a reduction in errors per thousand source lines of code

(KSLOC) from 4.5 to 1; the average level of reuse was up from 20% to 79%; and the average system cost was down 55%, from 490 to 210 staff-months (related, in part, to reuse).

McGarry concluded that process improvement activities not only enabled the SEL to produce more functionality, for more complex systems, with higher reliability, at significantly lower costs, it also had the side benefits of focusing the SEL's research activities, integrating standards and measurements with training and technology insertion, and contributing to a culture change that recognizes developers as being valuable partners in the process improvement paradigm, resulting in a synergistic approach to process improvement and a positive return on investment.

Question: In your comparison of the CMM and the SEL approaches, it struck me that they really didn't seem to be two independent approaches, in that you could use the SEL approach to improve your CMM rating. The SEL approach could be an implementation of a process to improve your CMM, so that it wasn't a matter of comparing one approach to another, because the SEL approach seemed to be more a how to do it, and the CMM approach seems to be more what you are. Do you see that difference as opposed to just saying these are two alternatives?

McGarry: I think there is a marriage between the two approaches. There is a reason that both exist. But your observation that there should be a continuous improvement and refinement of either model—in this case the CMM—driven by the product goals—I absolutely agree with that.

RECENT SEL EXPERIMENTS AND STUDIES

presented by Rose Pajerski, NASA/Goddard

Rose Pajerski began her discussion of recent SEL experiments with an overview of projects and studies conducted by the SEL since 1976. She emphasized that SEL studies have been based on the actual development of operational flight dynamics systems in the local environment, that each project represents a potential source of data for SEL studies, and that many studies are ongoing.

Pajerski indicated that, due to a recent organizational change, the SEL now has responsibility not only for software development, but also testing and maintenance. As a result, many of the new studies being conducted by the SEL are in the areas of testing and maintenance, thus allowing the SEL to study and report on activities that span the entire software life cycle.

Pajerski then quickly presented a summary of current SEL studies both in terms of the SEL improvement model and the software life-cycle phases. She reiterated that SEL studies follow a three-step improvement paradigm of understanding, assessing, and packaging. For this presentation, Pajerski chose to focus on three studies: (1) the cost and schedule estimation study, (2) a comparison of testing approaches, and (3) the maintenance study.

Cost and Schedule Estimation Study. The goals of this study were to rebaseline the SEL environment in terms of cost (effort), schedule, and reuse—given the organizational and technological changes—and to update the SEL's established baseline cost and schedule estimation models if necessary. The study included 39 projects ranging from 20 to 300 KSLOC and looked at the impacts of reuse, language, application types, and subjective factors such as levels of experience and kinds of technologies.

The study revealed some interesting discriminators: while the level of reuse and language drove cost, the application type drove schedule in the SEL. Key findings include: it costs 50% more to reuse a line of Ada code than a line of FORTRAN code; software size growth is 15% lower for high-reuse systems; it takes about 35% longer to develop a simulator than it does to develop an attitude ground support system (due to the level of uncertainty in the requirements); and subjective factors did not have a significant impact on cost or schedule.

Experiments in Testing. The goals of this study were to assess the impact of organizational changes on the SEL processes; to compare testing approaches with respect to their impact on process measures and product measures; and to assess effort and error distributions to determine testing effectiveness. The study examined four testing approaches, covering 34 projects of various characteristics during the past 15 years.

Before presenting the results of the experiments, Pajerski explained similarities and differences among the four testing approaches. In the SEL standard testing process, developers are responsible for implementing software and for system testing the software to verify end-to-end data flow. The software is then passed to a separate acceptance organization that tests the software using a functional, requirements-based approach. The Independent Verification and Validation (IV&V) approach adds an independent team to verify and validate the operational scenarios, and to ensure that the software requirements actually meet the mission needs. The SEL Cleanroom approach mandates that developers implement the software and that a separate test team perform integration and statistical testing and functional, requirements-based testing. The independent testing approach uses two teams: the development team, which implements the software and performs unit and integration testing, and an independent test team, which performs the equivalent of system and acceptance testing.

A comparison of test effort distribution by activity (design/code vs. test) revealed an observable change in process characteristics. The SEL standard testing approach was roughly comparable to the SEL Cleanroom approach, with both approaches splitting the activity time roughly 50:50 between design/code and testing. In the independent testing approach, however, 70% of the activity time was spent in design/code, and only 30% of the activity time was spent in testing. This observable process change will be studied more in the coming year by the SEL.

Pajerski then commented on what she called the bottom-line goal of the test experiments: determining testing effectiveness by looking at product measures. In examining error rates for the various approaches, Pajerski presented preliminary evidence showing that projects using the independent test approach experience fewer errors per KSLOC during development than those using the SEL standard test approach or the SEL Cleanroom approach. Furthermore, high-reuse projects had much lower error rates than low-reuse projects. Pajerski concluded that both the testing approach and the reuse level affect error rates.

Maintenance Study. The short-term goal of this study is to build a baseline understanding of the maintenance process in terms of software characteristics, effort distribution, and error/change profiles. The long-term goal of the study is to build estimation models for maintenance. There are 105 operational systems under maintenance totaling 3.5 million SLOC, ranging in size from 10 to 250 KSLOC. Eighty-five percent of the code has been developed in FORTRAN, and 80% of the work has been done on the mainframe computer. All activities after the first operational use of the system are being examined to build this baseline understanding.

Pajerski first discussed error rates and cost of maintenance. She pointed out that single-mission systems had detectable error rates of 0.1 errors/KSLOC and that multimission spacecraft have detectable error rates of 1.5 errors/KSLOC. She noted that the cost to maintain a multimission system was about 10% of the total development cost, whereas the cost to maintain a single-mission system was only 2% of the development cost. Pajerski conjectured that the difference in error rates as well as cost was due to the fact that the multimission systems are used, updated, enhanced, and maintained more often.

Pajerski then discussed effort and change type distribution. She noted that, in the SEL environment, about 27% of the requests for change received were for software enhancements and 72% of the requests for change received were due to error conditions. The actual effort to implement these changes, however, broke down differently: about 67% of the effort was spent enhancing the code, 22% of the effort was used to correct errors, and 11% of the effort was spent adapting the code to such things as new operating systems and compilers.

Pajerski then compared the effort distribution of development projects with maintenance projects as another way of characterizing the maintenance process. The most significant difference between maintenance and development effort distributions was that in development, 30% of the effort was devoted to testing, whereas in maintenance, only 5% was spent on testing.

Pajerski concluded her presentation by showing these studies in the context of the SEL process improvement model. She pointed out that the cost and schedule study had just completed the packaging stage; testing approaches were in the assessing stage; and maintenance was in the understanding phase. Pajerski maintained that none of these studies would ever be really complete, in that each of them would repeatedly cycle through the understanding-assessing-packaging phases as part of the SEL's ongoing process improvement program. With the change in the organization of the SEL, much effort will be spent studying and experimenting with testing and maintenance processes, and in time, the insights gained from studying these processes will be packaged as models, guidebooks, and tools.

Question: I was interested in the slide that suggests that it's more expensive to reuse Ada than it is to reuse FORTRAN, and I was wondering if there was a difference in productivity that tends to compensate in terms of the actual cost and also whether or not there are factors that are special to the SEL that affect this difference.

Pajerski: The productivity numbers are not the same for FORTRAN and Ada, and I did not have them up there on the chart. They do tend to balance one another out. John Bailey's Ada talk tomorrow is going to go over FORTRAN and Ada comparisons like that in some detail.

SPECIFICATION BASED SOFTWARE SIZING: AN EMPIRICAL INVESTIGATION OF FUNCTION POINTS

presented by Ross Jeffery, University of New South Wales

Ross Jeffery started out by describing the goal of the study as an assessment of function point metrics to evaluate the sources of variation in the metric. The organization that he studied was looking for a language-independent software sizing metric that could be applied early in the life cycle. The organization was the first to be certified to the Australian quality standard (AS 3563) and thus was likely to have minimal variation in its process.

The study is based on 17 recently developed systems in a variety of application domains by one software organization. The systems are implemented in C, Powerhouse, COBOL, Windows, and Excel macros.

The first part of the study compared function point metrics across the life cycle with the goal of assessing the metric's stability. Jeffery compared function point counts taken from specification products produced early in the life cycle with counts taken from final specification products that document the completed systems. He showed scatter plots that visually supported his statistics. The function point counts from the final specifications did well in predicting the effort even after removing some large projects that heavily influenced the results. Counts based on the early specifications did not do as well. His results using the post-completion counts were consistent with other studies in the literature. He pointed out that removing the outlier data from the post-completion correlation did reduce the correlation as expected, but that the effort-predicting function did not change substantially—evidence that the organization performed consistently across project size.

Analysis showed that one or two function elements accounted for much of the variation seen in the complete function point metric. Others have reported similar results but with internal correlation for different sets of function elements. Some of the function elements showed a higher correlation with effort than the aggregate counts. With this result, it is possible to develop another definition of function points, using fewer function elements, that would simplify the estimation process in the early parts of the life cycle.

The weighting factors used to compute the level-2 weighted function points seemed to be neutral for the set of data Jeffery presented. Applying the weights did not significantly change the ability of the metric to predict effort.

In the last part of Jeffery's presentation, he discussed an analysis of the possible causes of error in the process of developing function point counts. He considered three sources of error: rater interpretation of the specification, applicability of the function point counting method to the application, and rater interpretation of the function point counting rules. His study used only one function point counting standard, thus the second and third source of error could not be separated.

Two raters were used to count function points for all the projects in the study. One was a professional rater; the other was less experienced. (Function point counts used in the earlier parts of the presentation were the average of the two raters' counts.) The results of an analysis of the differences between the raters showed an average relative difference of approximately 55% between the raters. About one-third of this difference could be ascribed to rater error (the bulk of which came from the expert, ironically, Jeffery pointed out). The remainder is attributable to the function point standard or the requirements specification.

Jeffery presented a version of the function point scatter plot that showed a distinct pattern of separation of the counts by application domain (distributed systems vs. traditional centralized mainframe systems).

In summary, Jeffery said that function points showed a strong relationship with effort based on counts taken after the project was completed. However, some of the internal counts correlate, and this amounts to counting the same thing twice. Because the counting process is manual, it is subject to variations due to human error.

He summarized future directions for the organization that he had studied. More accurate effort predictions would result from putting more effort into the early requirement specifications. Developing the metric automatically within a CASE tool would reduce the human error as a source of variation in the metric.

Question: Do CASE vendors offer function point metrics?

Jeffery: Yes. But what is that metric? Is it right for you? Generally speaking, CASE users do not understand the metric generated for them. The real risk is that it comes free with the software so people don't question it.

SOFTWARE COST FORECASTING AS IT IS REALLY DONE: A STUDY OF JPL SOFTWARE ENGINEERS

presented by Martha Ann Griesel, Jet Propulsion Laboratory

Martha Griesel began by describing the situation in which engineers make cost forecasts. Cost estimation is done at the same time as the engineering. Engineers have integrated forecasting into their development process. Thus it is important to integrate forecasting tools into the environment. It is also very important to develop tools that assist with the forecasting process that they use now; engineers will not use tools that use a different forecasting process.

The goal of the study was to discover the fundamentals of cost forecasting as it is actually practiced by engineers. Are there only a few processes? Are the individual methods used by engineers so unique that there is no hope of developing tools to help them? If there are differences, where do they arise?

A search of the literature supported the idea that there is a small number of forecasting activities but also showed that studies of the process do not produce repeatable data.

Because of the multidisciplinary nature of the study team, it was possible to borrow several useful techniques. In this study cognitive psychology provided a technique that allows verbal dialog to be scored in a consistent manner. This enabled the construction of a "costing vocabulary." Stochastic processes provided transition probability matrices as a tool for analyzing how people move from one activity to another.

The study used 28 verbal reports from software cost and size forecasters who had a great deal of experience and who were also identified as personnel with good track records in forecasting. The descriptions of forecasting activities came from early phases of development. Analysis of the verbal reports developed a high-level set of forecasting activities such as requirements identification, size estimation, and cost estimation. Each individual report was then reviewed to extract the sequence of the high-level activities.

Griesel presented the COCOMO model as an example of how the forecasting activities might be connected in sequence. (No one in the study used a model that remotely resembled this example.) In an attempt to find patterns, all 28 activity sequences were overlaid, but the result was a complete blur.

After some study, three basic forecasting approaches were identified: new/old decomposition, assessment, and size estimation. The technique used in new/old decomposition was to partition the system into parts familiar to the forecaster and those not familiar. Different estimation techniques were then used for the new and old parts. The assessment approach involved obtaining a second estimate as a sanity check. Some forecasters developed a size estimate before they developed an effort estimate. Some application domains showed a preference for a particular basic approach.

Cost forecasting is roughly divided into three phases: problem definition and analysis, cost determination, and cost assessment. Griesel presented and described composite transition diagrams for each phase of each basic forecasting approach.

In summary, the study was able to identify some fairly well-defined paths through the forecasting activities and to detect some dependence on the application domain. Most forecasters used one approach to forecasting and very few performed detailed work in more than one phase of forecasting. Forecasters use simple techniques and tend to keep the number of attributes small.

Question: It's great to have a multidisciplinary study on real tasks. Were you able to relate accuracy of estimate versus the method used?

Griesel: The described activities were based on those used most recently and were not tied to specific tasks. We would very much like to collect that sort of data.

ASSESSING EFFICIENCY OF SOFTWARE PRODUCTION FOR NASA-SEL DATA

presented by Anneliese von Mayrhauser, Colorado State University

Anneliese von Mayrhauser began her presentation with the statement that it would be a "success story of some failures." Her study used a technique prevalent in operations research, production models. This presentation used data from the NASA SEL database.

Production models are attractive for this analysis because the efficiency of software development has many drivers such as people, process, and product characteristics. An analysis based on production models would use three steps. First, the measures used as input and output to the models must be selected based on the goals of the study: what type of efficiency is to be evaluated? Second, the models are applied to projects and the efficient and inefficient projects are identified. Third, a root cause analysis is performed based on other available project descriptors such as subjective measures.

In production model analysis empirical data are used to define a production function. von Mayrhauser described a simple production model with one input and one output to demonstrate the principle behind a production function. In the example, she demonstrated the definition of efficiency as the best observed ratio of the selected input and output measures. Other (non-best) observed ratios of input to output would be some fraction of the best. The relative efficiency of a project is the ratio of its efficiency to the best. She assured the listeners that the simple example could be extended to more complex situations. (von Mayrhauser's coauthor has applied this technique to another environment.)

As she began to describe the results of one production model (the study investigated 11 models), she described the trouble in identifying data to drive the production models and her lack of faith in the rank-order nature of the subjective data she was using. She contrasted the enormous size of the SEL database to the sparse list of acceptable data to drive the production models. Incomplete data and the small number of ratio-level metrics were cited as disappointments. Specifically, she was unable to investigate quality metrics, performance metrics, and effort data by phase as production model drivers.

The study looked at factors that impact overall efficiency, those that pertained to only efficient or inefficient projects, and factors that discriminate efficient from inefficient projects.

One study result, which she labeled curious, was that certain factors correlate with inefficient projects and not with efficient projects. She specifically cited development team application experience as increasing the efficiency of inefficient projects but having no effect on efficient projects. The audience was cautioned not to place much faith in this result since the rank-order data were of unknown quality.

In conclusion von Mayrhauser stated that a consideration of how metrics are to be evaluated must be included in developing metrics. Avoid collecting lots of rank-order data; there is not much you can do with them. She assured the audience that a disciplined metrics development would bring many successes.

Question: I'm concerned about isolating individual factors. Any individual factor may show strange behavior. Take the factor you found, experience, for example. If experienced people are put on a project because negative things are happening on that project, then that will affect things. So you are not going to find a relationship between experience and efficiency because of the other factors. The problem is multivariant with tremendous interaction of those factors.

von Mayrhauser: I agree with you. This is where you have to have a rich enough set of factors for the production models and then, after efficiency is identified, other factors tell you if there is a reason for it. Then if you identify an unusual project you can just forget about it.

Follow-up: It's more complicated than that. One thing you can do is say: if the system has a high complexity level, then experience becomes important. But if it's not complex, the experience is not relevant.

Experience in isolation is not an interesting variable. There are interdependencies among the factors.

von Mayrhauser: I agree that we should not do one at a time.

Session 3: Technology Assessment

THE (MIS)USE OF SUBJECTIVE PROCESS MEASURES IN SOFTWARE ENGINEERING

presented by Jon D. Valett, NASA/Goddard

Jon Valett began his remarks by defining three categories of measurement data: quantitative, characteristic, and subjective. By subjective data Valett meant "those data that are based on the opinion of individuals." He suggested three reasons why people have tried in the past to capture subjective data: (1) to help quantify the software process; (2) to improve models of software process and product; and (3) to define software domains. As examples of subjective measures he listed team experience, management stability, quality of tool set, and product complexity. Among the previous models to make use of subjective measures were Walston and Felix, COCOMO, and various domain analysis models.

After setting the stage in this way, Valett launched into a historical summary of what the SEL has done with subjective data over the past 17 years. In 1977, the philosophy behind the SEL's original foray into collecting subjective measures was to validate the models of other researchers and, in so doing, to fully characterize the SEL environment. For each SEL project, over 300 pieces of data were collected and rated on a 0-5 scale. Upper level managers made these assessments, and the resulting data were entered into the SEL database with no validation or clarification.

Valett then briefly described two SEL research efforts, Bailey and Basili's "Meta-Model for Software Development Resource Expenditures" (1981), and Card, McGarry, and Page's "Evaluating Software Engineering Technologies" (1987). The first study attempted to develop a cost model that incorporated subjective process measures. The second attempted to answer the question "Do modern programming practices affect productivity and reliability?"

Valett drew two main lessons from these two early studies. First, having a lot of data does not mean that one can generate a lot of results. Second, beware of false correlations, because if you look hard enough, you are bound to find some correlations. To help prevent such erroneous conclusions, he recommended that researchers confirm their results over multiple similar data sets.

Following the Card study, the SEL revised its collection of subjective data. It reduced its set of data to just 36 items. This smaller set contains most of Boehm's subjective measures from his *Software Engineering Economics* (leaving out a handful not thought appropriate to the SEL environment) and includes a few additional measures thought applicable based on SEL experience. The 36 measures were rated on a 1-5 scale, were collected from the project leads (rather than upper management), and were again entered into the SEL database with no validation or clarification.

Valett next summarized the analysis of subjective measures found in the recent *SEL Cost and Schedule Estimation Study Report*, by Condon et al. (1993). Initial analysis here seemed to find some relationships between effort and subjective measures, but further analysis revealed no consistency across different subsets of SEL projects. Following this discovery, the researchers replaced the SEL subjective data with random integers (1-5) for all projects and then repeated the analysis. With random data, about the same degree and frequency of improvements were found in the accuracy of the resulting effort models as had been found with the actual SEL data.

Valett's final research study was one that he himself performed shortly before the Workshop. He tried to improve models for predicting effort, errors, and changes by including subjective measures data. Valett converted the integer-based subjective measures to a binary scale (as did Bailey and Card in their studies). Valett also assumed some dependency in the data. He found little or no consistency in his results across multiple SEL data sets and concluded that even conservative use of these data is questionable.

In summarizing the lessons he had learned from the studies, Valett cautioned the audience not to collect too much subjective data, not to blindly search for correlations, not to go beyond the validity and consistency of the data, and finally, not to rely on such data except to spot trends or to set experiment goals.

Despite Valett's negative assessments drawn from these four studies, he did not totally discount the value of subjective information. In his concluding remarks, he drew a distinction between the valuable nature of some subjective information—such as that collected in "lessons learned" documents and in project annotations—and the very questionable value of subjective measures data collected by survey forms. If there were to be any hope that survey form data could serve some useful purpose, it would depend on more rigorous local definitions of subjective measures and also on more consistent data collection methods than have been used previously by the SEL.

Question: In an experiment in which I was involved in the past year, we introduced people to N-squared charts and data flow diagrams. We then assessed them on their ability to learn, interpret, and use these tools. We found that in many cases people's subjective response at the end of a test was different from their objective performance. For example, we found that test subjects would actually perform better with one technique, but would say that they performed better or preferred the other technique. Do you have any thoughts on the kind of research that might get to a deeper level on why this happens and why in your research one person considers something a "3" and another person considers it a "5."

Valett: We have experienced similar things in the SEL. We did a comparison of testing techniques: code reading, structural testing, and functional testing. People in the experiment said that they found the most errors in functional testing, but in reality most of the errors were found in code reading. For subjective data our best hope is to come up with templates for definitions of subjective data so that we do understand across our environment what a given score means. But that is not an easy task at all.

ANALYSIS OF A SUCCESSFUL INSPECTION PROGRAM

presented by Ray Madachy, Litton Data Systems

Ray Madachy's talk presented the results of a couple of years' experience in implementing inspections at Litton. He pointed out that the metrics collection program was somewhat more recent than the peer review program.

Among the unique features of the Litton inspections program is the absence of a "reader" role. As Madachy explained, his company follows the Gilb method, not the Fagan method, of inspections, so they do no paraphrasing. The Software Engineering Process Group (SEPG) Peer Review Coordinator serves as the inspection moderator. Litton's method sets no time limit on causal analysis, and it allows no discussion of defect category during the inspection.

The inspection statistics data sheet that Madachy displayed reported the preparation time of each participant plus the number of major and minor problems asserted by each participant. (A major defect was defined as anything that caused a software trouble report; a minor defect was anything else.) It also listed the total pages inspected, the duration of the meeting, and the number of new defects found at the meeting but not caught during the preparation time. These data all went into Litton's database. More recently Litton has also collected and stored the number of rework hours spent, plus the number of major and minor defects accepted by the author.

Madachy then presented slides demonstrating several relationships and conclusions from the Litton experience. The first relationship showed a downward trend in average defects per page as one progressed through the software life-cycle phases, backing up Madachy's assertion that you "get more bang for your buck" from early inspections. A graph of inspection effort over time showed, as might be expected, that effort peaks occurred before each quarterly Technical Interface Meeting, when the customer evaluated inspected documents. Using other graphs, Madachy showed that the optimum inspection mode at Litton had the following characteristics: (1) 4 or 5 inspectors per meeting, (2) 40-50 pages inspected per hour, and (3) a ratio between 0.5 and 2.0 for preparation time versus inspection time. The number of pages per hour includes all types of documents except code. Litton recommends that 250 SLOC be inspected per hour. Inspections at Litton were typically 2 hours long.

Madachy devoted two slides to addressing the return on investment for the inspection process. To arrive at this figure, Madachy subtracted the inspection effort from the test phase effort saved. This effort saved was estimated by multiplying the number of major defects found at each inspection by the average effort to fix a defect during the test phase. (For several years preceding this inspection experiment Litton had kept data on trouble reports and the effort required to fix them; these data were used to provide the average effort to fix.) The inspection effort included the preparation time, the time spent at the actual inspection meeting, and also the time required after the meeting to fix the defects. Using these formulas, Madachy showed that 139 out of 223 inspections saved time. The average inspection savings for all inspections was 63.4 person-hours.

Madachy presented two slides showing the separate effects of preparation time (per page) and inspection time (per page) on the number of new items found (per page) at an inspection. He argued that both graphs showed that as the amount of time increased, so did the number of items found. The effect of inspections on reducing the number of trouble reports (TRs) written against a build was demonstrated by one graph which showed a 76% reduction in the TR density (TRs per KSLOC) following the introduction of inspections.

In his concluding remarks, Madachy emphasized that inspections were a worthwhile investment. In addition to the other points brought out earlier in his talk, he noted that inspectors and authors had both improved since inspections began, and that the inspection analysis provided the impetus for improving Litton's metrics tracking procedures.

In response to two questions, Madachy pointed out that Litton had tried inspections both with readers and without readers and found that, prior to coding, the reader did not add much. In addition, the defect finding rate for documents was better without a reader. Consequently Litton does not use a reader in its inspection process, although they are looking into using a reader for code, where they think it might be valuable. In Litton's process, the moderator takes over much of the reader's role, but Litton does not use paraphrasing.

LESSONS LEARNED APPLYING CASE METHODS/TOOLS TO ADA SOFTWARE DEVELOPMENT PROJECTS

presented by Maurice H. Blumberg, IBM Federal Systems Company

Maurice Blumberg began his talk by defining megaprogramming. STARS sees this as an emerging paradigm of software development that is process driven, relies on domain-specific reuse, and is supported by technology. In brief, it is "a product-line approach to building a family of systems." The STARS strategy is to demonstrate the benefits of megaprogramming on some real-world projects, not just pilot projects.

Blumberg's talk, however, was not about megaprogramming and the STARS demonstration projects, but rather about the precursor to megaprogramming. These precursor alpha test projects provided early experience and feedback in the use of the IBM STARS Software Engineering Environment (SEE) and helped define what it takes to transfer technology to a project.

The three alpha test projects were all based on a RISC System/6000 primary development platform running various CASE, publishing, and testing tools. This platform was connected via a LAN to a Rational (300C or 1000) design facility and also to Xstation and PS/2 (DOS, Windows 3.0) remote access workstations. Blumberg summarized the various tools used by each of the three projects in each of the software development life cycles.

Blumberg next presented some lessons learned applicable to all three projects. (Slides detailing lessons learned from individual projects are included in these *Proceedings* as well, but due to time limitations only the last two slides of this latter set were presented at the Workshop.)

The main impediment to change was inertia. To overcome this obstacle, Blumberg recommended enlisting early customer support, involving developers in the planning, and ensuring strong support and vision from management and technical leads. In addition, one must protect against overblown expectations arising from marketing hype and overzealous advocates of particular tools. Emphasizing realistic hopes, developing a phased implementation plan, and relying on strong management vision could help a project get over the initial setbacks involved in inserting new technology.

Blumberg saw planning as a critical part of the success of a technology transfer. There are many elements to the startup costs besides the mere purchase of software and some hardware. One must plan for wiring, installation, and checkout. One must carefully choose the number of licenses to support the planned uses, must factor in maintenance costs, and must anticipate significant adaptation and integration expenses. All tools need to be tailored, and each tool may require an administrator. Blumberg mentioned that all of the alpha projects significantly underestimated the impact of introducing multiple changes and technologies. Each project tried to do too much at once.

Despite the various difficulties encountered in each project in introducing new technologies, Blumberg noted that each project experienced a significant morale boost from the new technology and that the upgraded technology resulted in upgraded skills. The new process that evolved was more effective than the old process and involved better team communication and coordination. Higher quality products resulted, and higher productivity is anticipated in subsequent phases, now that most of the learning is over, the team is acclimated, and many of the problems encountered have been resolved.

Question: How do you discriminate between temporary hurdles and truly bad (for you) technology?

Blumberg: You have to expect some loss in productivity the first time you adopt a new tool. If the tool results in an unacceptably large drop in productivity, however, you must fix the problem or consider dropping the tool for that project.

SOFTWARE ENGINEERING WITH APPLICATION-SPECIFIC LANGUAGES

presented by David J. Campbell, Unisys Corporation

David Campbell began by observing that application-specific languages (ASLs) have the potential to dramatically reduce cost and to increase software quality and reliability. ASLs, he explained, are special-purpose languages designed to solve a specific class of problems by automatically generating source code or other work products. Much less code is required to write an application in an ASL than in a general-purpose programming language such as C or Ada. ASLs are also inexpensive to produce; an experienced team can usually implement an ASL in a few weeks or person-months.

The software generation process starts with a specification that the programmer writes in the ASL describing the requirements for the software to be produced. The specification is read by an ASL translator, which then generates source code or other work products, such as test cases or documentation. The high-order language code is then compiled to obtain the application program. One of the key features of this process is that it maintains a clean separation between what the software does and how it does it, making it easy to port software to a different system. However, Campbell pointed out, ASLs can be used only if you already know the solution to a problem. They allow you to formulate a generic solution as a set of reusable code templates; the translator instantiates the templates and produces the required software from the specification.

Not all projects are candidates for implementation using an ASL, Campbell said, although most large projects have some area that would benefit from ASL usage, such as screens, reports, or parsing input commands. ASLs should be considered for use on a project when coding tasks are repetitive, complex, or error-prone; when requirements are subject to change; or when the problem being addressed will recur on other projects.

ASL technology requires an expert in the application area to design the solution to the recurring problem, and an expert in language design/compilers to develop a language that allows the requirements to be specified in terms familiar to those working in the application area. The language expert will also develop the translator that checks the input specification for semantic errors and generates the code that satisfies the requirements.

Campbell cited two examples of ASLs that have been applied on projects at NASA/JSC: Editor Generator (Egen) and STMM (Strip Merge and Manipulate). Egen was developed for the Tethered Satellite System (TSS) and was subsequently used on two other payloads. STMM replaced 40 programs that performed specific operations on flight design data files.

For the TSS, the customer wanted a graphical-user-interface-based system for editing a database of 500 variable-field satellite commands. The typical approach to developing the 140 screens (that were needed in 3 months) would have been to divide them up and give them to five engineers to code. Since design decisions would have to be known ahead of time for this approach to work, an ASL was used instead. With Egen, all 140 screens were generated from a short input specification. In addition to the compilable source code, Egen also generated the 200-page user manual and prepared a test program that found specification errors automatically.

While ASLs are not a panacea, Campbell said, they do provide benefits when code is recurring. Not only do ASLs increase productivity (because less code is needed to develop and maintain software), they also increase reliability; once the templates are correct, the code will be correct and will conform to project standards. The main benefit of an ASL, however, is increased manageability. Updates to all 140 TSS screens can be performed in a single location in the translator. User manuals can be kept in sync with code by simply rewriting the input specification and regenerating the documentation. Requirements changes create less of an impact; the addition of 40 new TSS commands had no effect on the project cost or schedule.

Question: Some of what you are describing can be done with object-oriented programming languages. Where do you draw the line between using object-oriented programming and an ASL, where you have to maintain the translator, the inputs, and the high-order language source?

Campbell: In general, you make a cost assessment. An object-oriented language may take more lines of code to accomplish the same task. Also, in some cases, you may need to maintain only the generated output. When a tradeoff study shows that you can use an ASL because there is enough repetition, you will actually have fewer lines of code to maintain with the ASL. For example, the TSS translator [Egen] was 7 KLOC. Because it was based on compiler technology, only 4K of those lines of code were custom-written for the application. The rest were reusable components, code from parser generators, etc. Of that 4 KLOC, 1/3 were for the test program, 1/3 were for documentation, and 1/3 were for the actual application. The total generated TSS editor was 12 KLOC, of which only 4K were written. In general, there is a magnification between what you have to write and what you get generated: we get 2-1 to 10-1.

APPLYING FORMAL METHODS AND OBJECT-ORIENTED ANALYSIS TO EXISTING FLIGHT SOFTWARE

presented by Betty H. C. Cheng, Michigan State University

This project, Betty Cheng said, was sponsored by a NASA faculty fellowship and was performed at the Jet Propulsion Laboratory. The purpose of the effort was: (1) to integrate formal methods into a portion of shuttle software; (2) to construct an object-oriented view of the system, even though it was not developed using object-oriented technology; (3) to demonstrate the utility of formal methods in an industrial application; and (4) to facilitate current and future maintenance of the software, ensuring that the original functionality and safety-critical properties are preserved when features are added.

A formal method, Cheng explained, consists of a formal language with a well-defined syntax, a well-defined set of semantics, and a proof system that allows you to manipulate symbols in the language. Formal methods are used to improve the quality of a software system by uncovering incompleteness and inconsistencies, and for automatic reasoning and verification. Her project used a language called PVS (Prototype Verification

System) which was developed by SR International. It is a predicate-logic-based language with an interactive theorem proving capability.

Referring to the previous presentation, Cheng noted that object-oriented techniques, as well as ASLs, can be used to develop software specific to an application. The benefits of object-oriented techniques are abstraction, information hiding, and modularity, which facilitate understandability, maintenance, and reuse.

In this project Cheng said, they were interested in the ability of formal methods to support abstraction and to generate proof obligations. They chose the object modeling technique (OMT) developed by Rombaugh because it offers three complementary perspectives: the object model gives the architectural view of the system, allowing information to be organized pictorially and offering a view of how the whole system fits together architecturally; the functional model provides data and control flow information; and the dynamic model allows modeling of state transitions.

Although the shuttle has had an excellent record for software, computer scientists often did not participate in early requirements analysis for shuttle software. Authors were free to express requirements in the form they preferred, resulting in widely varying requirements formats, styles, conventions, and perspectives. To be able to insert new technology and features, today's requirements analysts need to know what the requirements are and whether the changes they propose will affect the original functionality of the software.

The specific project Cheng's team tackled was the Phase Plane module within the Orbit Digital Autopilot System (DAP). The Phase Plane module is a control system for monitoring the angular rotation of the shuttle. It sends information to a module that selects which jets should be fired to attain a desired shuttle position. The wiring requirements diagram for DAP that Cheng showed revealed a very complex system. Although the module has worked successfully for thousands of hours, analysts have had difficulties in understanding the requirements for the module and testing it, and they want to make changes to it in the future.

The goals of the project were to obtain high-level requirements for the module by applying reverse engineering techniques, to develop an OMT "roadmap" for the system, and to establish a linkage between the specifications and the diagrams. The team used an iterative process. First they constructed a low-level specification corresponding to the wiring diagrams and the source code. To introduce abstraction, they used data flow diagrams to model the as-built layer. They then worked upwards, preserving the critical information from one level to the next as they developed data flow diagrams and an object model for the Phase Plane.

The team learned a number of significant lessons during the project. The first was that several layers of specifications were needed to go from existing code to high-level specifications. Theorems must be constructed to describe the properties that a given layer obeys and to provide traceability from one level to the next.

The second lesson was that formal methods provide a mechanism for integrating disparate sources of project information—e.g., wiring diagrams, the crew training manual, and design notes. The third lesson was that object-oriented analysis and design techniques can be exploited for reverse engineering to help understand the original functionality of a system, its architecture, and its state transitions.

Finally, the team learned that reverse engineering is an iterative process; one level of formal specifications is constructed, followed by a level of diagrams, and the process is repeated for each higher level. The diagrams, Cheng noted, help introduce the abstraction necessary to produce the high-level requirements.

In summary, Cheng said, the project incorporated formal methods into an existing software system to facilitate maintenance tasks, to aid verification of critical system properties, and to expedite future changes by using automatic reasoning to find requirements violations. The project has also demonstrated that formal techniques are not merely academic. Currently, the team is developing mid-level specifications and constructing proofs of correctness that will trace one level of the specification to the next. They hope to integrate the formal specifications with the OMT diagrams more closely, and demonstrate how formal methods can be used to assure that critical properties are satisfied.

Question: Are you aware of the work being done by Nancy Leveson at the University of Washington? She is using a state-chart technique to reverse engineer systems for the FAA. Could you talk about the differences in the approaches?

Cheng: Again, we want to capture all aspects of the system. State charts only capture state transition information. We also want to have an architectural view of the system. The multiple views led us to use OMT.

INTEGRATING END-TO-END THREADS OF CONTROL INTO OBJECT-ORIENTED ANALYSIS AND DESIGN

presented by Janet E. McCandlish, TRW System Development Division

Janet McCandlish opened her presentation by observing that current object-oriented methodologies fall short in their representation of end-to-end system processing. With a functional decomposition approach, data flow or process dependency diagrams show how the entire system works. The focus of object-oriented technology, however is on individual objects as reusable components, not on how they tie together. With object-oriented technology, a system is represented piecemeal with multiple views, making it difficult to get a full picture of how the system operates.

In addition, the goals associated with object-oriented and distributed systems are conflicting. In real-time distributed systems, competing demands for resources are reconciled by partitioning the system into multiple processes. Object-oriented technology, on the other hand, strives to partition a system into objects, encapsulating all data and associated operations within the object.

The approach taken in the current research, McCandlish explained, is to represent threads of control and associated class/objects to better illustrate how a system operates. The researchers began by examining five different object-oriented analysis and design methodologies: Coad and Yourdon, Shlaer and Mellor, Booch, Firesmith, and Rombaugh. They then introduced a representation that overlays dynamic flow onto the static architecture. Because of the amount of information being handled, they grouped classes and objects at a higher level of abstraction in two phases: (1) logical groupings to provide a coarse-grained partitioning and (2) process groupings to extend logical groupings with process partitioning criteria.

McCandlish highlighted key aspects of object-oriented methodologies. Static architecture, she said, refers to a nontemporal representation of a system, typically depicted with entity relationship diagrams that have been enhanced to include attributes, operations specifications, and relationships. Dynamic behavior is usually reflected in a different view of the system, and contains state, data flow, and timing information. A thread of control is a path that traces the sequence of operations for a particular execution of a system. It represents a scenario for a particular test case, and can be used during analysis, design, or testing to trace through the model for completeness, and to address real-time processing requirements, timing constraints, bottlenecks, and the like.

The research team found that static and dynamic representation methods exist, but thread-of-control representations are limited. Showing a chart that compared the five different object-oriented methodologies, McCandlish noted that each is incomplete in some aspect. Firesmith's comes closest to satisfying end-to-end traceability, but information is spread over three different diagram techniques, making it difficult to assimilate.

In the team's approach, the logical view represents groupings of classes or objects that are logically related. Partitioning into logical groups is accomplished based on engineering judgment, and is designed to minimize the associations, aggregations, and generalizations between groups. This type of logical grouping helps one understand the system as a whole. It is not new; such groupings are addressed (using different terminologies) in each of the methodologies previously cited.

McCandlish showed an entity relationship diagram with two logical representations. The first showed the traditional logical partitioning of a system. The second representation showed a new, logical composite class representation, in which thread-of-control information is aggregated up to the logical grouping level and overlaid onto the logical view. Because the focus of this representation is on "boundary class/objects"—i.e., objects that communicate across the boundary lines of the logical grouping, these logical groupings may differ from those of the traditional representation. The rationale for this new representation, McCandlish reiterated, is to be able to look at the system from end-to-end. It is also the first phase of partitioning for process composite classes.

Because the object-oriented methodologies previously mentioned do not address how logical groupings may transition into allocations for processes, McCandlish's team also introduced a process view that maps the class/objects to processes (i.e., executable entities). To obtain this process view, the team took the logical composite classes and applied process partitioning criteria keyed to communication and timing. To minimize the communication among processes, classes and objects that communicate frequently are grouped into separate processes, as are class/objects that occur along a particular time-critical path or that access the database. Finally, the groupings are adjusted to ensure that total execution-time criteria are met. The result, McCandlish showed, is a process composite class representation.

To formulate these process composite classes, McCandlish said, the researchers store representations of the classes, their attributes, and all of their interrelationships in a database. They can then extract information and link it with threads of control to learn, for example, that operation x impacts attribute y. From the database, they can determine the number of dependencies amongst the threads-of-control and classes. They can then take process partitioning criteria and system constraints and apply those through allocation algorithms, such as branch-and-bound, to determine the best grouping for the process composite classes.

In summary, current object-oriented representations do not provide the viewer with a clear understanding of the end-to-end processing that defines system operation. Consequently, the research team has introduced logical and process composite classes that act as structures for representing groupings of class/objects and the threads of control through those class/objects. Further study is needed to extend these structures into a design language and to address cases where the object-oriented and distributed system partitionings are in conflict.

Question : Does the user actually see both the object view and the threads of control so he can have an idea that the system will work or not work at the design review?

McCandlish: That's the plan. You identify classes and objects and build it bottom up. Ultimately, the idea is to graphically represent this information, draw threads-of-control through it, and represent all the associated information in the database. Then you can pull out a thread-of-control and look at all the things that are impacted by it, and can assign timing information as well.

Session 5: Process

FUSING MODELING TECHNIQUES TO SUPPORT DOMAIN ANALYSIS FOR REUSE OPPORTUNITIES IDENTIFICATION

presented by Susan Main Hall, Softech, Inc.

Susan Hall's presentation described her team's experience performing a high-level domain analysis fusing functional analysis techniques with object-oriented analysis to facilitate reuse among several software development efforts. As part of the Army Reuse Center, Hall's team is chartered to identify reuse opportunities for clients who reuse the software and donors who produce software to be reused. Her group's specific assignment was to perform domain analysis of four Army systems currently under development, in 6 person-months. Each of the systems was functionally oriented and developed in Ada.

Hall's team was experienced in software development but not in object orientation. Because reuse is easier to achieve with object orientation, her group had to choose whether to use the functional models and then struggle to move to objects later or to struggle with the objects right from the beginning. After assessing their purpose, time limitation, and current skills, they decided to merge the functional and the object modeling techniques.

Hall began her discussion of the merged modeling techniques by first defining domain analysis and the difference between vertical and horizontal domains. Hall's group was tasked to examine a horizontal domain, application support layer (ASL) software. She stated that just about all system types have an ASL.

In using the modeling techniques, they took advantage of everything a functional model could offer (state transition diagrams, data flow diagrams, and flow charts) then moved to a homegrown functional hierarchical grouping which helped in transitioning to Rombaugh's object-oriented model.

Specifically, they began by reviewing the existing functional models and creating any missing data flow diagrams, state transition diagrams, and flow charts. This enabled them to capture the basic activities of the ASL. By noting commonalities and differences, they identified six components of the ASL domain:

- Perform utilities and services
- Provide user/machine interface
- Provide help information
- Provide application layer interface
- Manage ASL data
- Provide COTS interface

The homegrown hierarchical modeling technique was introduced to avoid losing important information when moving from a functional to an object model. This was needed because their first attempts to move from functional to object-oriented were unsuccessful; there wasn't enough information—not enough decisions had been made. The technique consisted of identifying functions in a hierarchical tree, grouping the lowest level functions together based on objects manipulated, and dividing functions into those in the domain and those interfacing with the domain.

Hall provided a sample of the hierarchical grouping technique they applied: ASL functions "accept user input" and "display output" were grouped as the "user-machine interface" object. The "store data" and "produce reports" functions were grouped into "database" object. The "database" object was further broken down into what was actually being manipulated—files, records, and fields. At this point it was possible to attach operations and attributes to the object-oriented model and to complete the transition from the functional model to the object model.

Hall now returned to the main purpose of the domain analysis which, as stated previously, was to facilitate reuse within one or among several software development efforts. Because they now had both functional models and object models resulting from their analyses, they were able to identify reusable functions as well as reusable objects in high-demand categories such as user-machine interface.

In summary, Hall stated that the multiple modeling approach enabled them to view the domain more clearly and to identify more substantial reuse opportunities in the process. She felt this approach was faster (completed in 6 person-months) than traditional domain analysis would have been and that it provided an effective modeling technique to be used in the future.

Question: Our group is starting to do object-oriented requirements development; we have been totally functional up to this point. Do you have additional materials that show how to move from data flow diagrams to some type of object-oriented, more detailed programming instructions?

Hall: Yes, we have all the models. The in-between models don't look like any particular modeling technique. When we started, we started with things we knew like Demarco, DFD technique. We went back to data flow charts to detail things like how the computer and user interfaced, down to studying a key and getting information back. As we did each iteration, at some point in time, the model started to look like a particular, popular technique. We ended up with a definite data set diagram and a definite Rombaugh object-oriented model.

AN EMPIRICAL COMPARISON OF A DYNAMIC SOFTWARE TESTABILITY METRIC TO STATIC CYCLOMATIC COMPLEXITY

presented by Jeffrey M. Voas, Reliable Software Technologies Corporation

Jeffrey Voas began his comparison of dynamic testability and cyclomatic complexity by first differentiating between what it means to *achieve* versus *assess* quality. He stated that it is easier to achieve quality than it is to assess or measure it. He explained that assessing software provides the extra confidence not directly available from testing. Because we can never be certain that a verification system is correct, software testability has been introduced to give us a level of confidence that the software is correct.

Voas emphasized the difference between testing and testability. Testing defines with some "authority" whether an output is correct. Testability says nothing about correctness, but rather the likelihood that errors are being hidden in the software. It takes exhaustive testing, which is generally not possible, to be certain there are no hidden errors. Testability is, then, a prediction of the probability that existing faults will be revealed during testing according to some testing scheme.

Voas' testability model or metric is based on two premises:

1) What is the likelihood that a fault will cause a failure during testing according to some testing scheme.

2) A fault that is unlikely to cause a failure will be more difficult to see during testing.

Voas explained his fault size metric using an image of urns containing black and white balls. Each ball represented one possible input to the software, with black balls representing inputs that produce failures and white balls representing inputs that do not produce failures. The occurrences of black balls are then strung or chained together to produce a "fault size." Fault size is a way of quantifying the likelihood of discovering a fault/error in the software.

For example, if five different inputs produce a failure due to one fault in the program, five black balls are strung together, giving a string length, or fault size, of 5. They are connected because those five balls all go to the same fault. When executed they cause the state to become infected and that infected state propagates to the output. With a fault size of 5, there are five chances (when pulling balls out of the urn) to pull one of the black balls in the string of five that will reveal that fault. The greater the string length, the higher the testability of the software, because there are more opportunities to discover that fault. Five faults of size 1, on the other hand, means that there is only one input in the entire urn that will reveal any one of those five faults, so the odds are lower of finding them.

Voas relayed that for years reliability/testing researchers have asked the question, "What is the probability that this program will fail?" He has rephrased the question: "What is the probability that this program can't fail even if the program is incorrect?"

Voas applied his model to source code generated by CASE tools at NASA/Langley. This software consisted of 3 to 4 KSLOC including 58 functions. He used 2,000 randomly generated inputs and ran the simulator for 55 hours. His results showed that 15 of the 58 functions were of "high testability." The functions in the low testability range could easily be exhaustively tested. He also evaluated these 58 functions using McCabe's Cyclomatic Complexity Metric. All 58 functions had complexity values of less than 10, indicating that they were not very complex. He concluded that "chaining" within the urn cannot be predicted based on the complexity measure. That chaining, though, is the key to answering his question: "What is the probability that this program can't fail even if the program is incorrect?"

Question: You mentioned doing a short study on a few thousand lines of code. I'm interested in whether you did a comparison of what your model predicts versus which modules were error prone?

Voas: We did this for NASA/Langley on software where they knew where the errors were. We wanted to test whether this technique would tell us where the errors were. Our analysis came back and told us, to the exact order of magnitude, the likelihood that those faults would affect the output. There's no way that you could come up with the exact number, but as long as you are in the ballpark, you can then convert it, using the probable correctness model, back to the number of test cases you would need to catch that fault. If NASA had tested to the level we said, they would have found the fault. This is the sort of analysis that has to be done to test the technique.

Follow-up: You said that it took 55 hours on a SPARC II to apply your technique to 3 to 4 thousand lines of code. Is it something that scales up linearly or is it a more difficult problem?

Voas: It does not scale up linearly; it is a more difficult problem. I would never tell you to brutely apply it to 1 million lines of code. You would apply this technique to the parts of your code that are most critical; namely the parts where you want to make sure there are no errors hiding.

SOFTWARE QUALITY: PROCESS OR PEOPLE

presented by Regina Palmer, Martin Marietta Astronautics

Palmer, a staff quality engineer, presented her perspective on software quality: "Is it due to process or people?" She claims that a defined process is necessary for quality, but without the right people working with that process, there may be no benefit to having it at all. She based this statement on her experience examining 8 years of data collected from six software development projects.

Palmer described the involvement of the Quality Assurance (QA) organization throughout the development life cycle in her environment: They start with the beginning of the project during process definition and develop a quality plan. At requirements time, QA reviews the requirements and ensures that they are traceable from the customer-level document and that they are complete, understandable, testable, and traceable. They follow through design, participate in design walkthroughs as an independent evaluator, and check that the code follows the design.

The projects Palmer studied were compared on the basis of

- Involvement of the developers with the process definition
- Stability of the requirements
- Thoroughness of the unit and system test
- Degree of quality oversight
- Variance from schedule
- Meeting budget and expected productivity

The first project, the only successful one of the six, was a critical software project involving the safety of the astronauts. It had a well-defined process, very stable requirements, thorough testing, and met its schedule and planned productivity level. It was developed by experienced people.

The other five projects, which Palmer defined as ranging from "unsuccessful" to "disastrous," had varying difficulties. For example, on four of the five projects the process was imposed; there was little or no involvement from the developers in defining the process. Other problem characteristics were that the requirements were only stable in two out of five projects; adequate testing was performed in one out of five projects; and quality assurance was only performed in two out of five. The impact to these projects was that delivery dates were missed and planned productivity levels were not met. Software did not meet requirements and software was delivered with known errors. According to data collected on these projects, the quality of the software produced did not correlate to the experience of the developer.

Palmer concluded with some lessons learned: Everyone has to be involved when the process is defined. Everyone has to agree with and abide by it. Get rid of people who are not cooperative. It is necessary to have the right people for the job. The right people may be experienced engineers or recent graduates.

All the metrics collected from these six projects were collected at the end of the projects using a tool called the "Software Quality Assurance Interactive Database," which was developed by Palmer's coauthor, Modenna LaBaugh. Palmer felt the tool was very effective but believes it would have been better to collect the data earlier in the projects' life cycle when there would have been a chance to correct problems.

Question: In defining your process, do you have a standard process that you tailor for use on each of the programs, or did you start from ground zero each time?

Palmer: We have a standard process that we tailor. We have standards and procedures that give you the minimum requirements. We have one for programs that are small, medium, and large. We pick the one based on size and tailor it for that program.

Follow-up: And, how do you get everyone involved in the process?

Palmer: To get everyone involved in the process we have a "tabletop" to discuss and write the first draft of the process document and pass it out to be reviewed by all program members. That document is done by a few people, Software Lead, Quality, Configuration Management, and Test. They do the initial draft and include the generic things that have to be there. Then we have another tabletop, attended by all members of the group, to discuss the process and get agreement on the process.

Session 6: Software Engineering Issues in NASA

PROFILE OF NASA SOFTWARE ENGINEERING: LESSONS LEARNED FROM BUILDING THE BASELINE

presented by Dana Hall, Science Applications International Corporation

Dana Hall presented his experiences and highlights of the data he gathered while building the baseline of software engineering within NASA. This work is being sponsored by the NASA Software Engineering Program as the first essential step in establishing a long-term evolutionary improvement program for software engineering organizations within NASA.

Hall explained that obtaining a baseline understanding of the current software products and software engineering practices is a mandatory first step of any process improvement program. The goal is to understand; not to judge right or wrong. This measured understanding is then used to identify and define potential process improvements and to later measure improvement progress. Although his presentation mainly focused on the initial baselining effort, he pointed out that the understanding part of the process improvement paradigm is ongoing; in a continuously improving organization, baselining should be done periodically to measure change.

He cited four categories of data of interest: Product data that provide end item characteristics; process data that describe how the end item is developed and maintained; environment information that describes how the process is supported by tools and infrastructure; and application domain information that describes the type of work being done, providing an essential context for interpreting the other data. He then presented several examples from the baseline of NASA/Goddard. For example, a surprisingly large amount, 33%, of the 12,000 people who work at Goddard work on software. He estimates that Goddard presently has about 43 million SLOC in operational use. He also presented information regarding language usage and effort distribution across software engineering activities.

Hall pointed out that data availability is often an indicator of process maturity of an organization. It is often very difficult to capture the data that you want. His experience indicates that you can capture information on languages, budgets, and amounts of software with an accuracy of +/-25%, but that less tangible data such as effort distribution by phase, error statistics, productivity, investment in overhead functions, and software longevity can be captured with an accuracy of only +/-50%.

Hall used a combination of four methods to gather the information: administered surveys, informal roundtable discussions, data and documentation review, and one-on-one interviews. He shared several key insights that he gained from this experience. He stressed the need to prototype the survey instrument and to make the responses quantities or checkmarks. He also stressed the importance of using a small team to gather the data (1 or 2 people who are familiar with the organization, e.g., NASA). One team member met with each respondent, listened to their responses and indicated the proper response on the survey form. This ensured consistency in the data and reduced misinterpretations of the questions or answers. Directed sampling produced the best results, starting with key senior managers and then sampling the "software pockets" within the organization to cross-verify the results. Hall typically samples 10% of the software people.

Hall pointed out that baselining is not free, but that it is not terribly expensive either; 18 staff-months were spent over a 12-month period to produce the Goddard baseline. Activities included survey development and testing, data gathering, data archiving, data analysis and information extraction, and reporting the results.

In summary, Hall reviewed several lessons learned. It's important to be objective at all times; learn, don't qualify. Be sure to gather the perspectives of those in different roles within the organization, such as managers, developers, testers; this provides cross-verification of the data. Layer the baselining, starting at the top and working down through the organization; only go as deep as you need. When you are finished, give the organization the opportunity to review your findings, but don't compromise them. Organizations don't like to be surprised, and the review will provide one last check for oversights. Finally, use a combination of methods to gather the information and data; it's the only way to successfully gather relatively accurate baseline data and information.

IMPACT OF ADA IN THE FLIGHT DYNAMICS DIVISION: EXCITEMENT AND FRUSTRATION

presented by John Bailey, Software Metrics, Inc.

John Bailey reported the results of an independent assessment that he has conducted over the past year to determine the future of Ada in the Flight Dynamics Division (FDD) at NASA/Goddard. The FDD began investigating Ada in 1985 with the expectation that they would fully transition to Ada within 10 years. But today, 9 years later, only 15% of the new code is being written in Ada. Bailey was to determine why and whether the FDD should abandon or continue to pursue Ada.

Bailey reflected that the FDD originally pursued Ada because it was expected to be "more that just another language;" it was expected to drive an integrated well-defined software engineering process and lead to a major culture change; it would help build better products, specifically by reducing life-cycle cost and schedule and by reducing errors.

Over the past 9 years, the FDD has delivered approximately 1 million SLOC in Ada; they delivered 11 systems, all satellite data simulation systems developed and operated on VAX computers. The organization's goals with regard to Ada gradually changed over this time period, beginning with familiarization and moving on to reuse, cost, generalized systems, and process.

Bailey reported that measurements taken on the projects show promise. Process measurements show a maturing use of Ada language features and evidence that a true process change has occurred. Product measures indicate significantly increased reuse, lower error rates, shorter project durations, and lower delivery cost when comparing recent Ada projects to the 1985 FDD FORTRAN baseline. However, FORTRAN systems show nearly the same degree of improvement over this same time period. The data also show that it costs more to develop a new statement in Ada than in FORTRAN, but that the cost to deliver a statement of Ada is lower due to high reuse.

Bailey was quick to point out, however, that although both Ada and FORTRAN systems were reaping similar benefits from high reuse which is rooted in object orientation, the implementation approaches were distinctly different. In the Ada systems, generalized reusable components were implemented as Ada generics facilitating reuse through parameterized instantiation within the reusing system. Conversely, in the FORTRAN systems, generalized components were written to handle all foreseen cases and isolated in large independent modules that are linked with system-specific code to form the reusing system. Although both approaches have led to faster, better, and cheaper development of new systems, the FORTRAN reusable components have required more maintenance.

With such promising results, why didn't Ada flourish in the Division? Bailey explained that other unanticipated factors had derailed progress. System performance of the Ada systems turned out to be a major issue. Little attention was paid to performance while building the early Ada simulators, because performance had never before been an issue for the FDD. But the early Ada simulators ran much slower than their FORTRAN counterparts, giving users a very bad first impression of Ada. Ada still suffers from this bad reputation today, even though the problems have been corrected and current simulators outperform most previous FORTRAN simulators.

In addition, limited vendor support for Ada development environments hampered efforts to expand use of Ada. The FDD began developing systems using a DEC Ada environment in 1985 with the hopes of expanding within 5 years to the IBM mainframe environment (where most of the large FDD systems are built and operated). DEC Ada provided good tools and adequate performance; but several in-house evaluations of compilers and tools for the mainframe proved them to have a very limited set of immature tools that were hard to use. Thus, it was impossible to begin to transition Ada to the mainframes as planned in 1990, and even today, Ada cannot be used in the FDD (mainframe) operational environment.

With VAX Ada systems reaping the benefits of high reuse and with the inability to expand to the mainframe environment, the amount of new code written in Ada declined dramatically, leveling off at 15% with about 25-30% of the staff having had exposure to Ada. When asked about their language of choice, developers cited existing reusable code as the main driver for language choice in general; some regarded Ada as just another language, while others declared Ada to have clear advantages, but noting that Ada requires more tool support than other languages.

Looking at all of the evidence, Bailey concluded that Ada had a major impact on the FDD. Even though the language itself had not been widely adopted, Ada concepts (e.g., generalization, object-oriented design, domain analysis, information hiding) laid the foundation for broad process improvements. A healthy competition between FORTRAN and Ada developers stimulated the infusion of Ada concepts into the FORTRAN projects resulting in across-the-board improvements. Bailey attributed this phenomenon to the strong emphasis on process and continuous improvement in the organization's culture.

In conclusion, Bailey recommended that Ada should not be mandated in the FDD, because there is no pressing need for a common language as there is in DoD, for example. Also, the FDD builds systqems that are fairly small (200 KSLOC) compared to the 1 million-line systems that Ada was designed to support. But Ada should not be abandoned either; it should be used as any other method or tool when appropriate. Over the coming year, Bailey will be developing guidelines for when Ada should be used within the FDD. He concluded by pointing out that one of the expected major benefits of Ada, lower maintenance costs over the long-term, has not been tested in the FDD because, due to the mainframe situation, Ada is used primarily for the throwaway simulators rather than the longer-lived ground support systems.

Question: How were the FORTRAN projects able to apply Ada concepts, since FORTRAN doesn't support them?

Bailey: FORTRAN doesn't enforce good software engineering practices as Ada does, but FORTRAN does allow them—there is nothing in the language that prevents the use of information hiding, for example. Parnas was doing information hiding in FORTRAN in the 1970s. Process was the key to this. The evidence here seems to indicate that when an organization has a strong process, it can afford a wider choice of languages.

SOFTWARE ENGINEERING TECHNOLOGY TRANSFER: UNDERSTANDING THE PROCESS

presented by Marvin V. Zelkowitz, University of Maryland

Marv Zelkowitz summarized his findings thus far from a study of technology transfer practices and policies within NASA. This work is being sponsored by the NASA Software Engineering Program in an effort to characterize how software engineering technology is transferred within the Agency.

His study addressed two fundamental issues. First, to determine how NASA technology transfer is *intended to* occur, i.e., what are the official mechanisms and organizations that have been established to facilitate technology transfer. Second, to determine how technology transfer *actually* occurs, by examining instances of successful technology transfer within NASA. The study considered all technologies that are used to build software, but its primary focus was on tools and processes developed specifically to support software engineering activities.

Zelkowitz surveyed a broad-based group of software engineering professionals to identify software engineering technologies of interest. When asked to list the top five technologies that have had the greatest impact on their jobs since 1980, they most often cited workstations and PCs, object-oriented methods, GUIs, process models, and networks. NASA software engineers identified the same set, but also included measurement. Zelkowitz noted that of this group of responses only object-oriented methods, process models, and software measurement are limited to the software engineering field.

Zelkowitz reported that NASA views technology transfer as a critical part of its mission—to transfer to industry useful technologies that are developed through space research. Thus, NASA's official technology transfer mechanisms focus on the transfer of aerospace and engineering technologies out of the Agency. Little attention is paid to the infusion of technology into the Agency; it's left up to the individual project personnel to stay aware of and use the best processes and technologies available to do their jobs. However, with shrinking budgets and NASA's current "faster, better, cheaper" emphasis, technology infusion will become a critical element of the improvement programs that will achieve these goals.

In addition, he pointed out that most technologies transferred in the engineering disciplines are product oriented, meaning that the process is packaged as part of the product. But in software engineering, processes that describe the actions to take are as important as the tools that are used; for example, inspections, object-oriented design, or Cleanroom methodology. Thus, a successful technology transfer model for software engineering technologies must address a process as well as a product.

To understand the process of transferring software engineering technologies, he used a directed study within NASA to identify instances of successful technology transfer within, into, or out of NASA. He reported on the first stage of this work, where he focused on studying technology infusion into NASA. His preliminary results, based on four example software engineering technologies (Ada, object-oriented design, Cleanroom methodology, and formal inspections) that have been successfully transferred into specific NASA organizations, indicate that there are two distinct stages in the infusion process: understanding and transition.

The understanding stage is when the consumer organization is learning about the technology, by experimenting and conducting pilot projects; this stage usually takes about 2.5 years. The transition stage involves phasing the new technology into full use in the organization for suitable projects; this stage lasts at least as long as the understanding stage, but can last much longer depending on the degree of change to the software development process in practice in the organization. In all cases, people-contact seemed to be the main transfer agent of change. Forward-thinking individuals within the organizations became aware of the technologies through professional papers, journals, or conferences and introduced the technology to their organization. In cases where personal contact was made with the technology developer and they were involved in the transfer process, the understanding time was shorter.

Zelkowitz closed by cautioning that his results are still preliminary and that he expects to learn much more as he continues this study during the coming year.

Session 1: The Software Engineering Laboratory

Victor Basili, University of Maryland

Frank McGarry, NASA/Goddard

Rose Pajerski, NASA/Goddard

The Maturing of the Quality Improvement Paradigm in the SEL

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

The Software Engineering laboratory uses a paradigm for improving the software process and product, called the Quality Improvement Paradigm [Ba85, BaRo88]. But this paradigm has evolved over the past 18 years, along with our software development processes and product. Since 1976, when we first began the SEL, we have learned a great deal about improving the software process and product, making a great many mistakes along the way. For example, we tried to assess the quality of our processes and products before we understood what they were. When trying to understand, we were data driven rather than goal and model driven. We tried to use other people's models to explain our environment rather than recognizing we had to build models of our own environment before we could compare it with others.

The learning process has been more evolutionary than revolutionary. We have generated lessons learned that have been packaged into our processes, products and organizational structure over the years. We have used the SEL as a laboratory to build models, test hypotheses. We have used the University to test high risk ideas and develop technologies, methods and theories when necessary. We have learned what worked and didn't work, applied ideas when applicable and kept the business going with an aim at continually improving and learning.

This paper offers a personal perspective on how our approach to quality improvement has evolved over time and where I think we are evolving. I will try to carry you through various phases of our evolutionary learning process, arbitrarily breaking the learning into five year periods. showing you some of the things we did wrong and what caused us to change our ideas. I will use the Quality Improvement Paradigm steps themselves, as it presently stands, as a guidelines to how our thinking evolved based upon experiences in the SEL.

But first, let me give you the Quality Improvement Paradigm, as it is currently defined. In its full version, it can be broken up into six steps:

1. **Characterize the** current **project and** its **environment** with respect to the appropriate models and metrics.

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

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

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

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

6. **Package the experience** 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 to be reused on future projects.

We often use a shortened version of the paradigm which is defined as three steps: understand, assess, and package. These steps can be mapped onto the six steps by noting that understand is step 1, assess is steps 2 through 5 and package is step 6.

Each of these steps changed over time, either in how we defined them or how we implemented them. Characterization went from collecting metrics to defining baselines to building models. Goal setting started out as simply data collection, evolved to being goal driven and finally goal and model driven, i.e., data collected based upon goals and quantifiable models. The processes, methods and technologies available in the process selection step evolved from combinations of heuristic methods, to well-defined technologies, to high impact, combinations of integrated technologies, methods, and life cycle models, to the evolving and tailoring processes to the situation. During process execution, we moved from loosely monitored projects to closely monitored projects with well defined feedback loops. In the beginning we collected too much data, independent of the process. Later data became embedded in the process. The types of analysis we performed in the beginning were correlations and regressions, and we have evolved to other forms of model building, based upon the nature of the software engineering data, and to the use of qualitative analysis. Packaging began as recording and generating lessons learned but evolved to focused tailored packages that were integrated into the development processes. We started by packaging defect and resource baselines and product characteristics and have been evolving to seeking the relationship between process and product characteristics.

1976 - 1980

What we did

We began the SEL in 1976. At that time, the paradigm looked like:

1. ~~Characterize/Understand~~ Apply Models
2. ~~Set Goals~~ Measure
3. ~~Select Process~~ Study Process
4. Execute Process
5. Analyze Data Only
6. ~~Package~~ Record

We tried to characterize and understand by using other people's models. For example we spent a great deal of time trying to apply such models as the Rayleigh curve model of resource allocation, reliability growth models, etc. without asking ourselves if they were appropriate for our particular environment.

We decided on measurement as an abstraction mechanism and developed data collection forms and measurement tools. We collected data from half a dozen projects for a simple data base and we defined the GQM as in informal mechanism to help us organize the data around the study of defects [BaWe84].

It had not really occurred to us to select process as we did not yet understand that process was a variable that needed to be selected and tailored to the environment. This was because we had not yet understood our environment sufficiently. So we started to study process, applied heuristically defined combinations of existing processes and began to run controlled experiments at the university with students.

During development, data collection was an add-on activity and was loosely monitored. We analyzed data only and began to build baselines and looked for correlations. We recorded what we found, built defect baselines and resource models and measured project characteristics.

What we Learned

During this period we learned that we needed to better understand the environment, projects, processes, products, etc. We needed to build our own models to understand and characterize our environment, we could not just use other people's models. Those models were built for their environments and could not be generalized easily.

We learned that we needed to understand what factors create similarities and differences among projects so we know the appropriate model to apply. This included the need to understand how to choose the right processes in order to create the desired product characteristics.

We realized that evaluation and feedback are necessary for project control and that data collection has to be goal driven; we could not just collect data and then figure out what to do with it.

From our perspective, the major improvement technology that emerged from this period was the Goal/Question/Metric Paradigm, even though it was still quite primitive.

An Example

As an example of what we learned, we tried to apply the 40/20/40 rule in SEL. It had been reported by Boehm [Bo73] that approximately 40% of project resources were expended in analysis and design, 20% in code, and 40% in checkout and test. Shortly thereafter, Walston and Felix reported that in IBM/FSD, 35% of the resources were expended in analysis and design, 30% in code, 25% in checkout and test and 10% in other, which clearly violated the 40/20/40 rule [WaFe77]. But in the SEL, we were collecting two types of resource data, phase data and activity data. The phase data represented milestone data. That is, analysis and design data represented the resources expended up to the design review milestone (CDR). The activity data represented what a developer did each week, e.g., 20 hours designing, 10 hours coding, 5 hours in training, 5 hours in travel. Using the phase data, we found that 20% of the resources were expended in analysis and design, 45% in code, 28% in checkout and test and 5% in other, while using the activity data, we found that 21% of the resources were expended in analysis and design, 28% in code, 23% in checkout and test and 27% in other.

	TRW	**IBM**	**SEL**	
			Phase	**Activity**
Analysis/Design	40%	35%	20%	21%
Code	20	30	45	28
Checkout/Test	40	25	28	23
Other		10	5	27

Table 1. Resource Allocation Data

It became clear that the data from the other environments represented phase data rather than activity data since they did not collect activity data. It also was clear that each of the organizations defined their milestones and phases differently, so each organization has a different model for resource allocation and it is hard to compare them. Phase data is highly dependent on how an organization defines its milestones. Since phase data and activity data represent two entirely different things, it is not clear what the activity data look like in these other organizations. It should be noted that this example represents an argument why it would be very difficult to build a national data base across environments and share and compare data.

1981 - 1985

What we did

In the early eighties, the paradigm had evolved to look more like:

1. Characterize/Understand
2. Set Goals
3. Select Process
4. Execute Process
5. Analyze
6. ~~Package~~ Record

To characterize and understand the environment we built our own baselines/models of cost, defects, process, etc. We began to set goals for all data collected and expanded our definition of the GQM to perform studies across multiple areas and projects. We began to incorporate subjective metrics into our measurement process. To help us select process we experimented with well defined technologies and began experiments with high impact technology sets, e.g., Ada & OOD. During project execution, we collected less data than we had before and moved the data from a file system to a commercial, relational data base. We began to understand how to combine some of our off-line controlled experiments with the case studies in the SEL. We shifted the analysis emphasis to the process and its relation to product characteristics. We recorded lessons learned, and began formalizing processes, products, knowledge and quality models.

What we Learned

During this period we learned that software development follows an experimental paradigm, i.e., you need to set your goals up front and check that you are achieving those goals. The design of experiments is an important part of improvement and evaluation and feedback are necessary for learning. We also learned that we needed to better understand relationships between various kinds of experiences, e.g., the relationship between processes and the set of product characteristics it evokes or the resources required to perform it, the relationship between component size and complexity and defect rate. To do this process, product, and quality models need to be better defined, experimentally tested, and improved.

We learned that reusing experience in the form of processes, products, and other forms of knowledge is essential for improvement. We need to learn what works and what does not work and what needs to be modified and what needs to be thrown out. At the same time we need to experiment with new technologies, motivated by our experiences.

By this time, we had more data than we knew what to do with them, but we did not have the data we needed to help us interpret what was happening. We learned that you can drown in too much data, especially if you don't have goals. Besides having a good data base, you need to store your models as well as your data .

An Example

As an example of demonstrating that we need to understand the relationship between variables, consider the study in the SEL where we compared fault rate with component size and complexity. In a study in the early eighties, we found that the simple minded view that defect rate increases with size did not hold in the SEL environment. In fact, we found the opposite for the actual data we had available for study [BaPe84]. We believe this relationship is due to the fact that interface defects dominate the problem of the complexity of the individual component, when components are small.

On the other hand, we have hypothesized that as the size grows beyond the developer's ability to cope with its size and complexity, the complexity of the individual component will dominate the complexity of the interface and fault rate will again grow.

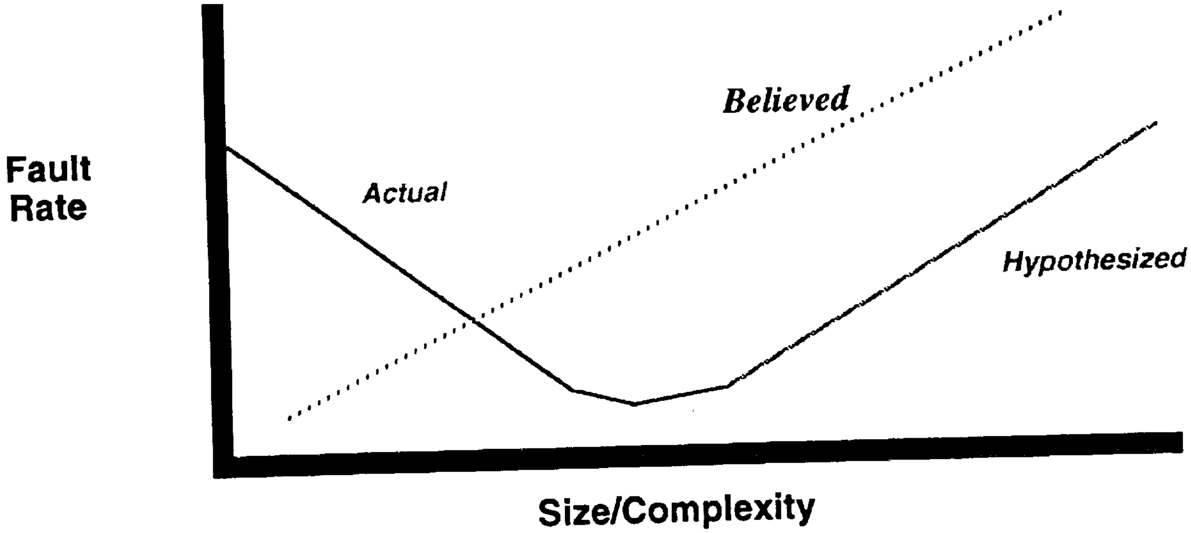


Figure 1. Relationship between Fault Rate and Size or Complexity

We have since found support for the first statement, i.e., fault rate decrease with size and complexity in data from several companies. This result was a surprise at the time since most people believed that smaller components were better. However. the relationship between size and fault rate appears not to be that simple.

1986 - 1990

What we did

It was in this period that the QIP took its current form, recording being changed to packaging.

1. Characterize/Understand

2. Set Goals
3. Select/Tailor Process
4. Execute Process
5. Analyze
6. Package

To characterize and understand we worked on capturing experience through models. Goals and models became the commonplace driver of measurement and we built SME [Va87], a model-based experience base with dozens of projects. We began to tailor and evolve high impact technologies based on experience, e.g., Cleanroom, and experimentation and feedback became an integral part of the QIP. During process execution, we embedded the data collection process into the development processes and more closely monitored projects, especially those where we were experimenting with new approaches.We began to demonstrate various (process, product) relationships, e.g., the effect of a particular method on defect reduction. We developed focused tailored packages, e.g., generic code components, and learned to transfer technology better through organizational structure, experimentation, and evolutionary culture change.

What we Learned

We learned that experience needs to be evaluated, tailored, and packaged for reuse. That is, you just cannot write lessons learned documents, you have to analyze and synthesize what has been learned and integrate it into the existing knowledge so that it is usable by future projects. This requires organizational support and resources.

A variety of experiences can be reused, e.g., process, product, resource, defect and quality models. But processes must be put in place to support the reuse of experience and the development process must be modified to take advantage of reusable experiences. Experiences can be packaged in a variety of ways, e.g., equations, histograms, algorithms.

Packaged experiences need to be integrated. When introducing a new process, an organization needs to make sure it fits and is supported by the other processes being used, that is, it needs to understand the relationship between various changes in the parameters in one model and the effect on another model. If I modify my reading technology, what will be the effect on the class of defects I find, the resources allocated for rework, etc.

There is a tradeoff between reuse and improvement. Evolution is slow as I cannot introduce too much change at one time. When I do introduce change, I loose experience and predictability. On the other hand, processes have to be changed to cope with the continuously growing need for quality.

During this period we evolved the GQM to include templates and models [BaRo88] and formalized the organization via the Experience Factory Organization [Ba89].

An Example

To demonstrate that how a technology is packaged and integrated has a strong effect on its effectiveness, consider our experiences with evaluating and integrating reading technology.
We ran a controlled experiment comparing equivalence partitioning testing, structural testing, and reading by step-wise abstraction[BaSe87]. Reading was found to be more effective and efficient than testing in uncovering defects. Based upon these results, we put reading into practice as a technology in the SEL. But we found that reading had little effect on defects. This appeared to be because the readers did not read well because they knew they were going to test and believed that,

in spite of the experimental results, testing was better. Our belief that reading is more effective when not followed by developer testing motivated our use of the Cleanroom approach [SeBaBa87]. When embedded in the Cleanroom approach, reading did demonstrate a substantial lowering of defect rates.

1991 - 1995

What we are doing

This bring us up to the current time. The current evolution of the QIP appears to be aimed at instantiating the steps, making them more specific, providing details, and developing support technologies.

To characterize and understand the project and environment, we are building a repository of (process,product) relationship models that characterize the SEL environment. We are working on automating the GQM in order to support the setting of goals. We are studying what experience is exportable to other environments in help other organizations take advantage of our process experience We are working on building models to measure process conformance and domain understanding.

During execution of the processes, we are working to capture the details of experience by providing more interaction between developers and experimenters and more effective feedback mechanism. This will help us to evolve processes that are more focused and detailed for our local needs and goals.

We are building qualitative analysis approaches to extract our experiences and provide input to the data models. We continue to evolve SME and we continue the evolution and packaging of the Experience Factory Organization.

Many of the current, specific SEL activities are covered in this workshop proceedings. However, there are more global SEL activities aimed at evolving the application of the QIP to other organizations. These activities concern packaging the SEL organizational experience for other groups in NASA, understanding whether and how to move activities to common use, and better integrating reuse into the development process

The research activities are based upon instantiating the steps of the Quality Improvement Paradigm by providing support technologies and automation, and integrating the various activities.

Where the research is going

The table below shows some of our current research interests aimed at instantiating the Quality Improvement Paradigm.

Step	Studies / Research Projects
Characterize	Perform domain analysis to identify similar projects using techniques appropriate for SE data
Set goals	Automate the model-based GQM as much as possible

Choose process	Develop technologies tailorable to the specific project needs
Execute processes	Build a more powerful, flexible experience base
Analyze data	Learn how to run more efficient experiments and combine controlled experiments with case studies
Package experience	Build better models and modeling notations

Table 2: Instantiating the Quality Improvement Paradigm

Example research projects

To give some specific examples of research projects, let us consider three: the work on domain analysis, reading technologies, and empirical modeling.

Domain Analysis

Problem Addressed:
How do you recognize which projects are most like yours in order to use the experiences from these projects to allow you to build models, choose similar process, etc.?

Current Status:
We have established procedures to identify and analyze software domains within and across organizations so that opportunities for reuse of experiences may be identified [Lionel Briand]. This has entailed defining both an experience-based procedure taking advantage of intuition and expert knowledge as well as a data-based procedure for when data is available.

Validation Strategy:
We are using both procedures to identify domains within NASA, and have analyzed data within the SEL data base to determine whether or not our assumptions are supported locally.

Focused Tailored Reading Techniques

Problem Addressed:
How do you tailor a process to the project goals and local organizational characteristics?

Current Status:
Have developed scenario-based technologies for reading various documents that are tailorable and can be focused for the particular environment. As an example, we have developed several model-based scenarios that take advantage of local knowledge and technical models to define a technology for reading. For example, defect-based reading is based upon the different defect classes, e.g., missing functionality, data type inconsistencies, in a requirements document that have been found in requirements [BaWe81].

Validation:
We have run a couple of controlled experiments that show that defect-Based reading is significantly more effective that ad hoc reading or checklists [PoVo94].

Empirical Modeling: Optimized Set Reduction

Problem Addressed:
How do you build empirical models that allow you to define interpretable, accurate, easy to use and automate modeling procedures that take into account the specific constraints of software engineering data?

Current Status:
OSR has been developed based on pattern matching; searching for similar experiences in the data set and the use of non-parametric statistics. There are no functional assumptions made; the approach handles interactions and inter dependencies among variables, and no "learning" parameters need to be tuned before hand.

Validation:
We have shown OSR to be easier to interpret and more accurate than regression and tree-based approaches for cost modeling and defective module prediction [BrBaTh92, BrBaHe93]. A prototype tool exists and a commercial tool is under development.

Conclusion

Over the past 18 years we have learned a great deal about software improvement. Our learning process has been continuous and evolutionary like the evolution of the software development process itself. We have packaged what we have learned into our process, product and organizational structure. This evolution is supported by the symbiotic relationship between research and practice. It is based upon a belief that software engineering is a laboratory science. As such it involves the interaction of research and application, experimentation and development. It is a relationship that requires patience and understanding on both sides, but when nurtured, really pays dividends!

References

[Ba85]
V. R. Basili, "Quantitative Evaluation of Software Engineering Methodology," Proc. of the First Pan Pacific Computer Conference, Melbourne, Australia, September 1985 [also available as Technical Report, TR-1519, Dept. of Computer Science, University of Maryland, College Park, July 1985].

[Ba89]
V. R. Basili, "Software Development: A Paradigm for the Future", Proceedings, 13th Annual International Computer Software & Applications Conference (COMPSAC), Keynote Address, Orlando, FL, September 1989

[Ba90]
V. R. Basili, "Software Modeling and Measurement: The Goal/Question/Metric Paradigm," University of Maryland Technical Report, CS-TR-2956, UMIACS-TR-92-96, September 1992.

[BaRo88]
V. R. Basili, H. D. Rombach "The TAME Project: Towards Improvement-Oriented

Software Environments," IEEE Transactions on Software Engineering, vol. SE-14, no. 6, June 1988, pp. 758-773.

[BaPe84]
V. R. Basili, B. Perricone, "Software Errors and Complexity: An Empirical Investigation," ACM Communications, vol. 27, no. 1, January 1984, pp. 45-52.

[BaSe87]
Victor R. Basili, R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," IEEE Transactions on Software Engineering, Vol. SE-13, No. 12, December 1987, pp. 1278-1296.

[BaWe84]
V. R. Basili, D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, vol. SE-10, no.6, November 1984, pp. 728-738.

[BaWe81]
V. R. Basili, D. M. Weiss, "Evaluation of a Software Requirements Document by Analysis of Change Data," Proceedings of the Fifth International Conference on Software Engineering, San Diego, USA, March 1981, pp. 314-323.

[Bo73]
B. W. Boehm, "Software and its Impact: A Quantitative Assessment," Datamation 19, No.5 48-59 (My 1973).

[BrBaTh92]
Lionel C. Briand, Victor R. Basili, and William M. Thomas, "A Pattern Recognition Approach for Software Engineering Data Analysis," IEEE Transactions of Software Engineering, Vol. 18, No. 11, pp. 931-942, November 1992.

[BrBaHe93]
Lionel C. Briand, Victor R. Basili, and Christopher J. Hetmanski, "Developing Interpretable Models for Identifying High Risk Software Components," IEEE Transactions on Software Engineering, November 1993.

[PoVo94]
Adam Porter, Larry Votta, "An Experiment to Assess different Defect Methods for Software Requirements Inspections," Proceedings of the 16th ICSE, Sorrento, Italy, May 1994.

[SeBaBa87]
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.

[Va87]
J. D. Valett, "The Dynamic Management Information Tool (DYNAMITE):Analysis of the Prototype, Requirements and Operational Scenarios," M.Sc. Thesis, University of Maryland, 1987.

[WaFe77]
C. E. Walston and C. P. Felix, "A Method of Programming Measurement and Estimation," IBM Systems Journal, Vol. 16, No. 1, 1977, pp.54-73.

The Maturing of the Quality Improvement Paradigm in the SEL

Victor R. Basili

Institute for Advanced Computer Studies

Department of Computer Science

University of Maryland

December 1-2, 1993

Maturing the Improvement Paradigm Since 1976

In 18 years have **learned a great deal**, e.g.,
- tried to assess, before understanding
- were data driven rather than goal and model driven
- tried to use other people's models to explain our environment

Learning process has been more **evolutionary** than revolutionary

Generated **lessons learned** that have been **packaged** into our process, product and organizational structure

Used the **SEL as a laboratory** to build models, test hypotheses,

Used the University to **test high risk ideas**

Developed technologies, methods and theories when necessary

Learned what worked and didn't work, applied ideas when applicable

Kept the business going with an aim at improvement, learning

Talk offers my perspective on
- how we have evolved
- and where we are going

Maturing the Improvement Paradigm
Quality Improvement Paradigm

Characterize the current project and its environment with respect to the appropriate models and metrics.

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, validate and analyze the data 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 to be reused on future projects.

Maturing the Improvement Paradigm
Major Activity Evolution

Characterize
metrics ----> baselines ----> models

Set Goals
data driven ----> goal driven ----> goal/model driven

Select Process
heuristic ----> defined ----> high impact ----> evolving
combinations technologies combinations processes

Execute Process
add-on data collection ----> less data ----> data embedded in process
loosely monitored ----> closely monitored/feedback

Analyze
correlations ----> regressions ----> model ----> qualitative analysis

Package
recording ----> lessons learned ----> focussed tailored packages
defect ----> resources ----> product ----> process x product
baselines models characteristics relationships

Maturing the Improvement Paradigm
1976 - 1980

~~Characterize/Understand~~ Apply Models
Looked at other people's models, e.g., Raleigh curve, MTTF models

~~Set Goals~~ Measurement
Decided on measurement as an abstraction mechanism
Developed data collection forms and measurement tool
Collected data from half a dozen projects for a simple data base
Defined the GQM to help us organize the data around a particular study

~~Select Process~~ Study Process
Used heuristically defined combinations of existing processes
Ran controlled experiments at the University

Execute Process
Data collection was an add-on activity and was loosely monitored

Analyze Data Only
Mostly build baselines and looked for correlations

~~Package~~ Record
Recorded what we found, build defect baselines and resource models

Maturing the Improvement Paradigm
1976 - 1980

Learned

Need to better **understand** environment, projects, processes, products, etc.

Need to build our own models to understand and characterize
- can't just use other people's models

Need to understand the factors that create similarities and differences among projects so we know the appropriate model to apply

Need to understand how to choose the right processes to create the desired product characteristics

Evaluation and feedback are necessary for project control

Data collection has to be goal driven
- can't just collect data and then figure out what to do with it

...

Developed the Goal/Question/Metric Paradigm

Maturing the Improvement Paradigm
1976 - 1980

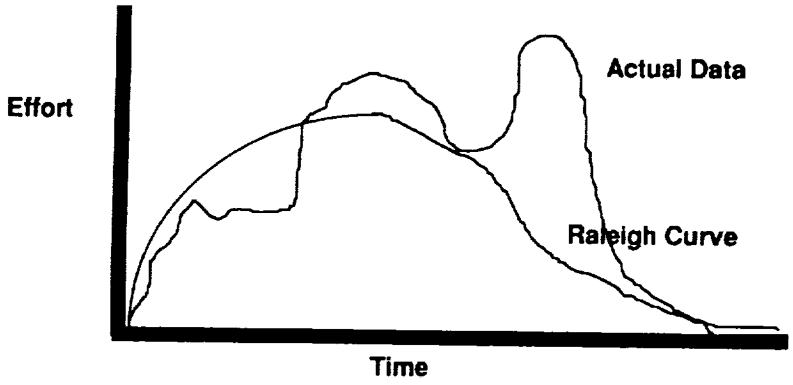
Trying to Apply the 40/20/40 Rule in SEL

	TRW	IBM	SEL	
			Phase	Activity
Design	40%	35%	20%	21%
Code	20	30	45	28
Checkout/Test	40	25	28	23
Other		10	5	27

The 40/20/40 rule does not apply to us

The rule does not imply what you may think

Maturing the Improvement Paradigm
19~~81~~76-19~~88~~80
Applying a resource Allocation Model



Need to understand the local context
Local context makes a big difference

Maturing the Improvement Paradigm
1981 - 1985

~~Characterize~~/Understand
Built our own baselines/models of cost, defects, process, etc.

Set Goals
Began to set goals and defined the GQM to study multiple areas
Began to incorporate subjective metrics into our measurement process

Select Process
Experimented with well defined technologies
Began experiments with high impact technology sets, e.g., Ada & OOD

Execute Process
Began to understand how to combine experiments and case studies
Collected less data and stored it in a relational data base

Analyze
Shifted emphasis to process and its relation to product characteristics

~~Package~~ Record
Recorded lessons learned
Began formalizing process, product, knowledge and quality models

Maturing the Improvement Paradigm
1981 - 1985

Learned

Software development follows an **experimental paradigm**, i.e.,
Design of experiments is an important part of improvement
Evaluation and feedback are necessary for learning

Need to experiment with new technologies

Need to learn about relationships
- process, product, and quality models need to be better defined

Reusing experience in the form of processes, products, and other forms of knowledge is essential for improvement

Can drown in too much data, especially if you don't have goals

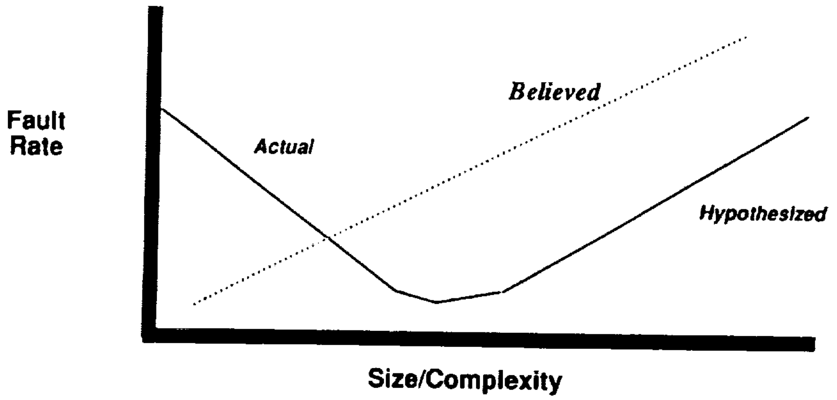
Need a data base and you need to store your models as well as your data

...

Developed the QIP as:
Characterize, Set goals, Choose process, Execute, Analyze, and **Record**

Maturing the Improvement Paradigm 1981-1985

Measuring Fault Rate against Size and Complexity



We need to understand the relationship among variables
The relationship between fault rate and size is non-linear

Maturing the Improvement Paradigm 1986 - 1990

Characterize/Understand
Worked on capturing experience in models

Set Goals
Goals and Models became the commonplace driver of measurement
Built SME, a model-based experience base with dozens of projects

Select Process
Tailored and evolved the high impact technologies based on experience
Experimentation and feedback became and integral part of the QIP

Execute Process
Embedded data into the processes and closely monitored study projects

Analyze
Demonstrated various (process, product) relationships

Package
Developed focussed tailored packages, e.g., generic code components
Learned to transfer technology better through organizational structure, experimentation, and evolutionary culture change

Maturing the Improvement Paradigm 1986 - 1990

Learned

Experience needs to be **evaluated, tailored,** and **packaged** for reuse

There is a **tradeoff between reuse and improvement**

Software processes must be put in place to support the reuse of experience

A variety of experiences can be reused, e.g., process, product, resource, defect and quality models

Experiences can be packaged in a variety of ways, e.g., equations, histograms, algorithms

Packaged experiences need to be integrated

...

Reformulated QIP as:
Characterize, Set goals, Choose process, Execute, Analyze, and **Package**

Evolved GQM to include templates and models

Formalized the organization via the Experience Factory Organization

Maturing the Improvement Paradigm 1986 - 1990

Evaluating and Integrating Reading

Testing vs. Reading experiment
Reading more effective and efficient than testing

Reading in Practice
Reading had little effect

Reading as part of Cleanroom at the University
Reading had a high impact

Reading as part of Cleanroom in the SEL
Reading had a high impact

How a technology is packaged and integrated has a strong effect
Reading more effective when not followed by testing

Maturing the Improvement Paradigm 1990 - 1995

Characterize
- Building (process,product) relationship models

Set Goals
- Automating the GQM

Select Process
- Study what experience is exportable
- Study process conformance and domain understanding

Execute Process
- Capture the details of experience - more interaction between developers and experimenters - more effective feedback
- More focused and detailed on our local needs and goals

Analyze
- Qualitative analysis to extract experiences

Package
- Continuing to evolve SME
- Evolving and packaging the Experience Factory Organization

Maturing the Improvement Paradigm 1991 - 1995

Many specific activities in the SEL will be covered in this workshop

SEL activities aimed at evolving the application of the QIP concern
- packaging the SEL **organizational experience** for NASA
- understanding whether and how to **move activities** to common use
- better **integrating reuse** into the development process

Research activities aimed at evolving the QIP are mostly based upon
- **instantiating the steps** of the Quality Improvement Paradigm
- **providing support technologies** and automation, and
- **integrating** the various **activities**.

Maturing the Improvement Paradigm 1991 - 1995

Instantiating the Quality Improvement Paradigm

Step	Studies / Research Projects
Characterize	Perform domain analysis to identify similar projects using techniques appropriate for SE data
Set goals	Automate the model-based GQM as much as possible
Choose process	Develop technologies tailorable to the specific project needs
Execute processes	Build a more powerful, flexible experience base
Analyze data	Learn how to run more efficient experiments and combine controlled experiments with case studies
Package experience	Build better models and modeling notations

Maturing the Improvement Paradigm Domain Analysis

Problem Addressed:

How do you recognize which projects are most like yours to build models, choose process, etc.?

Current Status:

Establishing procedures to
identify and analyze software domains within and across organizations so that opportunities for reuse of experiences may be identified.

We have defined
- a data-based procedure
- an experience-based procedure

Validation Strategy:

Identify domains within NASA, analyze data to determine whether or not our assumptions are supported.

Lionel Briand

Maturing the Improvement Paradigm
Focused Tailored Reading Techniques

Problem Addressed:

How do you tailor a process to the project goals and local organizational characteristics?

Current Status:

Have developed scenario-based technologies for reading various documents that are tailorable and can be focused for the particular environment

Example: Defect-based reading is based upon the different defect classes, e.g., missing functionality, data type inconsistencies, in a requirements document

Validation:

Defect-Based reading has been shown to be significantly much more effective that ad hoc reading or checklists

Adam Porter, Larry Votta

Maturing the Improvement Paradigm
Empirical Modeling: Optimized Set Reduction

Problem Addressed:

How do you build empirical models that allow you to define interpretable, accurate, easy to use and automate modeling procedures that take into account the specific constraints of software engineering data?

Current Status:

OSR has been developed based on
- pattern matching; searching for similar experiences in the data set
- non-parametric statistics
- no functional assumptions, handles interactions and interdependencies
- no "learning" parameters to be tuned before hand

Validation:

Shown easier to interpret and more accurate than
regression and tree-based approaches
for cost modeling and defective module prediction
Prototype tool exists; commercial tool under development

Lionel Briand, Chet Hetmanski, Bill Thomas

Maturing the Improvement Paradigm Conclusion

Over 18 years we have **learned a great deal** about software improvement

Our learning process has been **continuous and evolutionary** like the evolution of the software development process itself

We have **packaged** what we have learned into our process, product and organizational structure

The evolution is supported by the **symbiotic relationship between research and practice**

It is a relationship that requires **patience and understanding** on both sides, but when nurtured, really pays dividends!

PROCESS IMPROVEMENT AS AN INVESTMENT: MEASURING ITS WORTH

Frank McGarry
Kellyann Jeletic

SOFTWARE ENGINEERING BRANCH
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771
(301) 286-6347
(301) 286-7698

ABSTRACT

This paper discusses return on investment (ROI) generated from software process improvement programs. It details the steps needed to compute ROI and compares these steps from the perspective of two process improvement approaches: the widely known Software Engineering Institute's Capability Maturity Model and the approach employed by the National Aeronautics and Space Administration's (NASA's) Software Engineering Laboratory (SEL). The paper then describes the specific investments made in the SEL over the past 18 years and discusses the improvements gained from this investment by the production organization in the SEL.

INTRODUCTION

For many years, various organizations have put forth significant efforts toward the improvement of software process and product. In recent years, the development of the Software Engineering Institute's (SEI) Capability Maturity Model (CMM) has significantly accelerated interest in the overall improvement process for software. With the development of this model, software development organizations have a relatively clear definition of recommended approaches for attaining better and better levels of software process that, in turn, is expected to result in better and better software products. After six years of experience with the application of the CMM concept, there still is a shortage of empirical evidence quantifying the impact of investments in software process improvement. In general, there has been significant uncertainty in the return on investment stemming from process improvement activities. As organizations invest resources in software process improvement efforts, they need to understand what they are getting for their money and determine whether there has been any benefit from this investment.

This paper details the steps needed to compute return on investment (ROI) and compares these steps from the perspective of two process improvement approaches: the widely known Software Engineering Institute's

Capability Maturity Model and the approach employed by the National Aeronautics and Space Administration's (NASA) Software Engineering Laboratory (SEL). It then describes the specific investments made by the SEL over the past 18 years and discusses the benefits gained from this investment by this production organization.

SEL OVERVIEW

The SEL is an organization sponsored by NASA's Goddard Space Flight Center (GSFC) which was created to investigate the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was established in 1976 and has three primary organizational members: NASA/GSFC's Software Engineering Branch, the University of Maryland's Department of Computer Science, and the Computer Sciences Corporation's (CSC's) Software Engineering 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 apply successful development practices.

Within the SEL, over 100 production projects have been monitored and studied over an 18 year period to assess the impact that process change has on the developed software products. These production projects result in software that is used for ground support for GSFC missions and is typically used to carry out all Flight Dynamics functions at the GSFC. These software projects range in size from 4 or 5 thousand (K) source lines of code (SLOC) to over 1 million SLOC, with a typical size of 100-300 KSLOC.

In carrying out these 100 'experiments' with software process, the SEL has accumulated detailed information on specific processes used for each project as well as the resultant product characteristics such as cost, error rates, cycle time, rework required, etc. With this information, some insight can be gained into the ROI that is attained with the usage of particular process changes within the environment.

CHARACTERISTICS OF TWO SOFTWARE PROCESS IMPROVEMENT PARADIGMS (CMM AND SEL)

Although the paradigm used by the SEL differs from the SEI's Capability Maturity Model (CMM), both approaches share the underlying principle of continuous, sustained software process improvement. The CMM focuses on improving an organization's software process by evolving through a series of maturity levels to attain the ultimate goal, becoming a continuously improving organization ('level 5'). At each level, the organization must meet a set of well-defined criteria to advance to the next level or beyond.

Within the CMM, an organization strives to mature to a continuously improving process. To do so, the organization must advance through the following maturity levels [Reference 1] where the organization's software process is defined as:

Level 1 -	an ad hoc process
Level 2 -	a repeatable and more disciplined process
Level 3 -	a standard, consistent, and defined process
Level 4 -	a predictable and manageable process
Level 5 -	an optimizing and continuously improving process

The SEL's process improvement paradigm consists of a three step iterative process driven by the specific goals of an organization (e.g., to decrease average error rates) and the experience gained from earlier development efforts (e.g., most errors are interface errors). These three steps include:

1) *Understanding* - a baseline of an organization's software process and product is developed. How is the organization's software business done? What is the lifecycle process? What standards are used? What are the characteristics of its software

product (e.g., cost, error rates, productivity)?

2) *Assessing* - based on the goals of an organization (e.g., reduce error rates), some change is introduced to the process and the subsequent result of that change is assessed.

3) *Packaging* - once improvements have been identified and verified, they are packaged in some tangible form (e.g., training, standards) and infused back into the organization's process.

These three steps are performed iteratively and continuously over time.

The two process improvement paradigms, the CMM and SEL, are depicted in Figures 1 and 2, respectively.

HOW IS ROI COMPUTED?

With any process improvement approach, an organization is eager to determine what it has gained from its investment. There are five steps necessary to determine the benefits gained from investing in software process improvement. These are:

(1) Define goals. The organization must set goals for what is to be improved.

(2) Produce a baseline. The organization must establish a basic understanding of its current software process and product.

(3) Invest in change. To improve anything, change must first be made. An investment in this change must be made.

(4) Assess change. Once a change has been made, its effects must be measured to determine if any improvement has been achieved.

(5) Measure ROI. Has the investment in process improvement been a success? What has the investment been and what has been gained from this investment? The ROI must be measured by (a) determining what resources have been expended for software process improvement, (b) establishing what improvements, both quantitative and qualitative, have been achieved, and (c) determining the difference between the investment made and the benefits obtained. Has the investment been worthwhile?

How are these steps achieved within the framework of the CMM and SEL process improvement approaches? Each step is addressed below from the perspective of both approaches.

Throughout this paper, 'process' refers to the characteristics of how an organization develops and maintains software. 'Process' includes the organization's tools, standards, policies, life cycle, management approaches, etc. It also includes all measures reflecting these items such as effort distribution, error distribution, profile of software change and growth rate, etc. 'Product' refers to the characteristics of the resultant software including productivity, reuse levels, error rates, cycle time to produce, etc.

1 - DEFINING GOALS:

Each organization must set goals for what is to be improved. With the CMM, the goal is generalized, i.e., to improve the software process. With the SEL, goals are product-driven and vary from organization to organization.

CMM: There is a generalized, domain-independent goal that focuses on process. Every organization strives to improve the software process and, ultimately, evolve to a continuously improving, optimizing process (maturity level 5). Organizations A and B both try to improve their processes and become level 5 organizations, thereby minimizing any risk incurred because of software development.

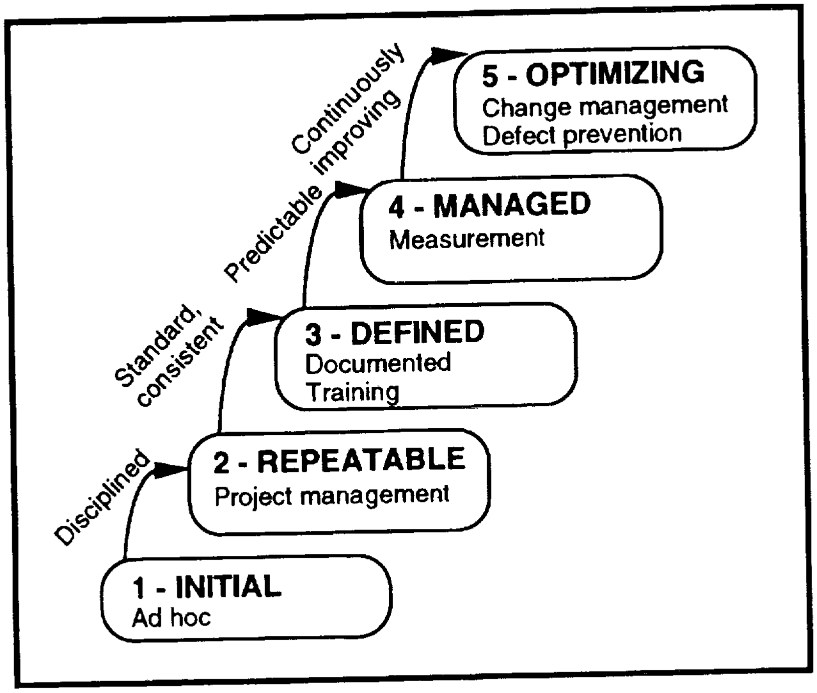


Figure 1. CMM Process Improvement Paradigm

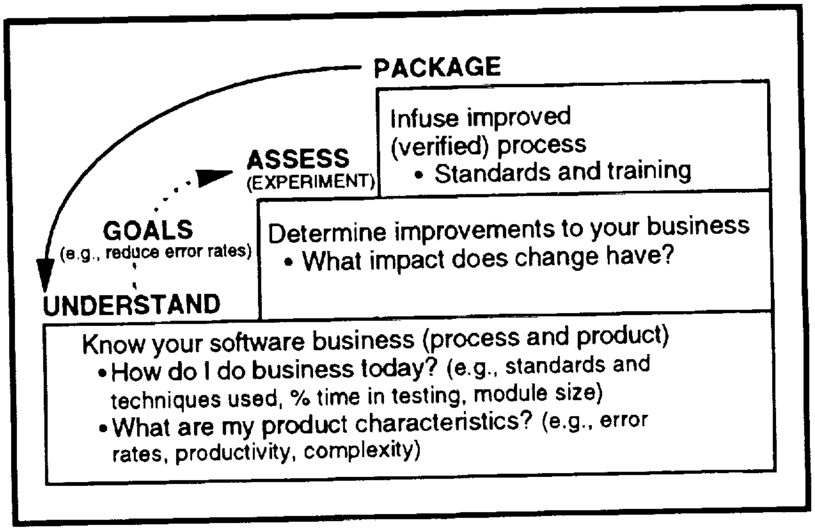

PACKAGE
Infuse improved
(verified) process
• Standards and training
ASSESS
(EXPERIMENT)
Determine improvements to your business
• What impact does change have?
GOALS
(e.g., reduce error rates)
UNDERSTAND
Know your software business (process and product)
• How do I do business today? (e.g., standards and techniques used, % time in testing, module size)
• What are my product characteristics? (e.g., error rates, productivity, complexity)

Figure 2. SEL Process Improvement Paradigm

SEL: The emphasis of the SEL approach is to improve the organization's software product. Goals vary from organization to organization and are driven by product, not process characteristics. Organization A may attempt to improve reliability by decreasing error rates. Organization B may strive to decrease development cycle time. Goals are domain-dependent.

2 - PRODUCE BASELINE

Each organization must establish a basic understanding (baseline) of its current software process and product. The CMM baseline is process-based and established against a 'common yardstick.' The SEL baseline is domain-dependent and is both process- and product-based.

CMM: Baselining within the CMM is achieved by performing an assessment of the organization's process. This assessment is made against well-established criteria defined by the SEI [Reference 1] and the organization is baselined at some maturity level. These criteria enable comparisons across domains since every organization is assessed against the same criteria, a 'common yardstick.' The same elements are examined for every organization: does it have good standards, what is its training program like, how is its measurement program, etc. Based on the examination of these criteria, the organization is baselined at some maturity level.

SEL: Baselining involves understanding the process and product of each individual organization. This baseline is organization-dependent (or domain-dependent). Unlike the CMM, there is no common yardstick enabling comparison across organizations. Some factors need to be characterized (baselined) by all organizations, e.g., how much software exists, what process is followed, what standards are used, what is the distribution of effort across lifecycle phases, etc. Other factors of interest depend on the goals of the organization. Organization A, for example, would want to baseline its error rate, while Organization B needs to determine its development cycle time.

The SEL process improvement approach emphasizes introducing change to attain process improvement. The effects of changes to process can only be measured by comparing them to the existing baseline. Understanding is a critical and continually needed element of the SEL approach.

Figures 3 and 4 are examples of the SEL's baseline measures. They represent data from Flight Dynamics projects as specified on the individual figures. Figure 3 depicts baseline values pertaining to process. It shows the SEL's typical effort distribution and classes of error. Figure 4 depicts baseline values associated with product. It shows the SEL's typical error rates, cost, and level of code reuse.

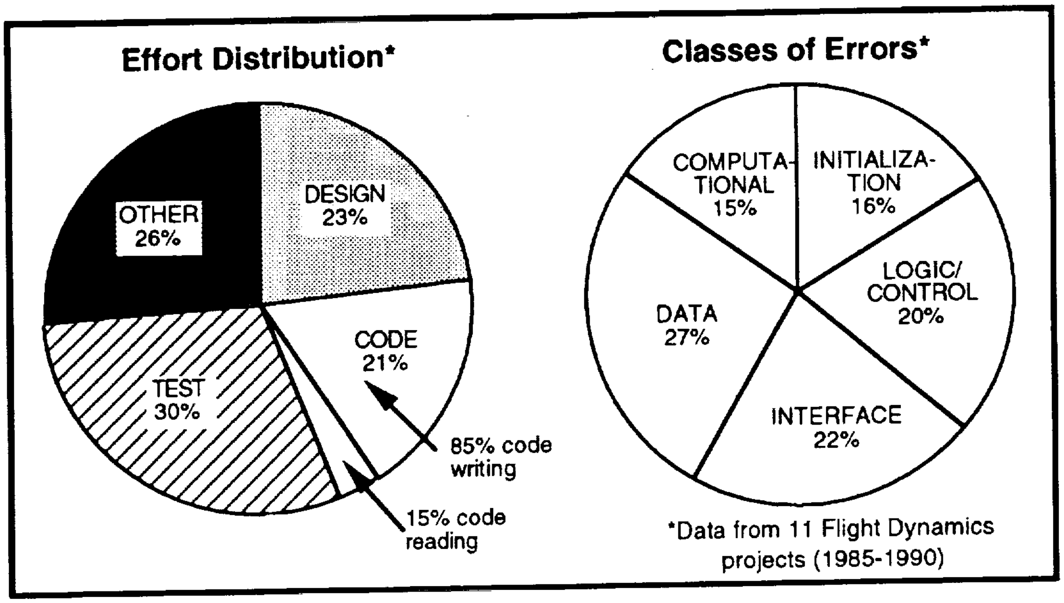
These examples represent some elements that may be characterized by an organization baselining its process and product.

3 - INVEST IN CHANGE

Organizations striving for software process improvement must invest in change. Within the CMM, the common yardstick drives change. Within the SEL, organizational goals and experiences drive change.

CMM: The CMM's common yardstick drives change. That is, the elements by which the CMM assesses maturity levels drive change. If an organization is baselined at some level, it will change elements necessary to get to the next maturity level. If an improved measurement program is needed to advance to another maturity level, the organization will focus on changing its measurement program to meet the CMM criteria. This common yardstick enables a common roadmap to success -- continuous improvement.

SEL: The goals and experiences of individual organizations drive changes. Changes to the process are made in an attempt to improve the product. An organization interested in increasing its level of reuse will invest in changes that focus on that improvement goal. For instance, they might decide to experiment



*Data from 11 Flight Dynamics projects (1985-1990)

Figure 3. Sample SEL Process Baseline

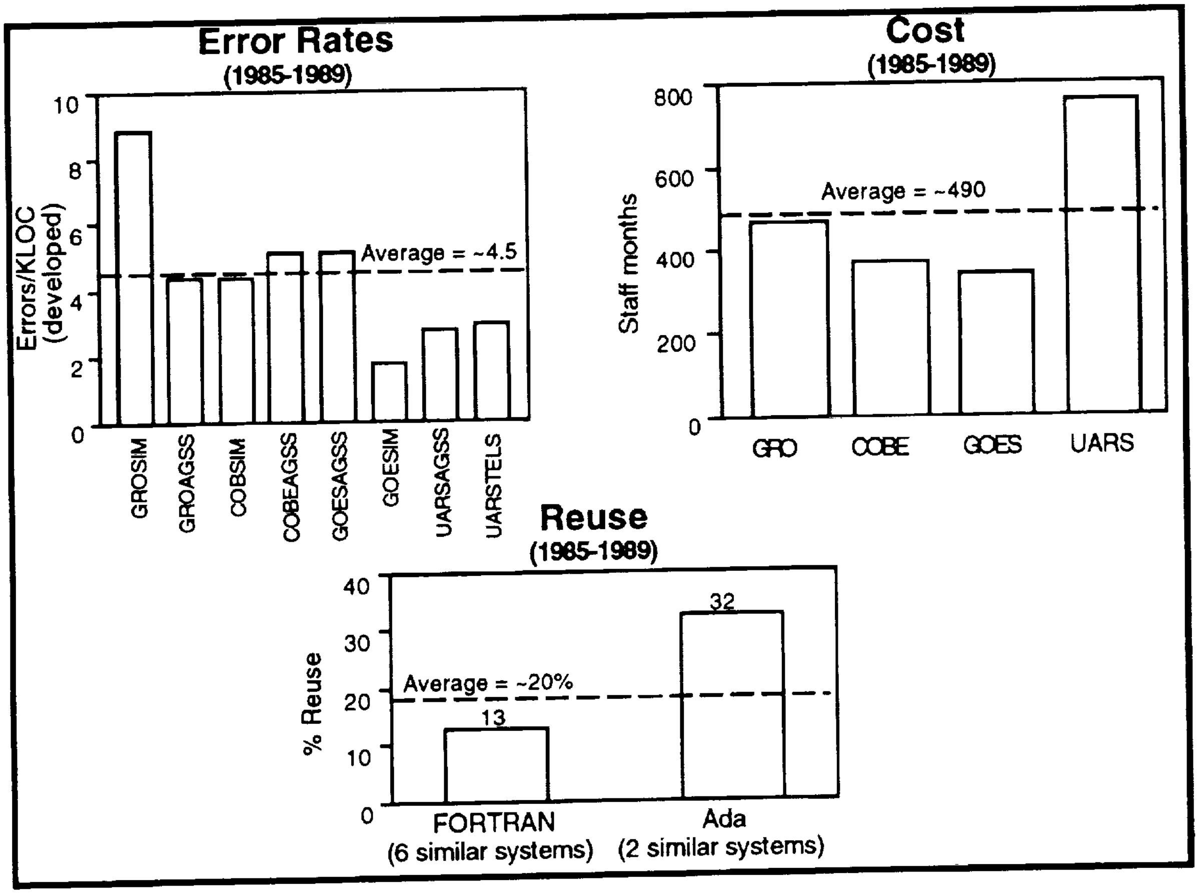


Figure 4. Sample SEL Product Baseline

with object-oriented design (OOD) to improve reuse. The organization interested in reducing error rates might decide to experiment with the Cleanroom methodology [Reference 2]. Each organization (or domain) must identify the most appropriate process changes to achieve its product goals.

The CMM is an excellent model of potential process changes that could be selected. Various elements of the model (e.g., key process areas (KPAs)) have emphasis on specific product improvements that can help in selecting potential changes in the SEL model.

4 - ASSESS CHANGE

Each organization must introduce change to make some improvement. An assessment of the changes must be made to determine if there has been improvement. The CMM assesses change by reassessing the process. The SEL assessment of change is domain-dependent and focuses on both process and product.

CMM: With the CMM, assessment of change is accomplished by reassessing the process. An organization is baselined at one level, makes changes to try to attain a higher level, and is then reassessed to determine if it has progressed to another level. Success is measured by process change. The ultimate success is changing the process until it is a continuously improving process. The organization achieves the highest maturity level rating, that is, advancing to level 5. The measure of success is domain-independent, since all organizations are measured against the same criteria, a common yardstick.

SEL: Assessment of change is domain-dependent. An improvement goal is set, change to the process made, change to the process and product examined and verified, and the effect of change evaluated against the original goal. Success is measured by product change and is determined based on the goals of the individual organization. The organization attempting to improve its reliability would institute a change, e.g., the Cleanroom methodology, to try to reduce its error rates. It would then assess the result of the Cleanroom experiment based on its original goals. What were the baseline error rates? What were the error rates resulting from the Cleanroom experiment? Did Cleanroom reduce error rates? The organization attempting to attain higher levels of reuse would make a change, e.g., OOD. Similarly, it needs to determine the level of reuse achieved using OOD and compare these reuse levels with the original baseline. The SEL examines both changes to process data and changes to product data.

Figures 5 and 6 show some sample assessments from the SEL representing process and product data. They represent data from actual Flight Dynamics projects as specified on the individual figures. Figure 5 depicts a process assessment showing the impact of a technology (Cleanroom) on the SEL's baseline effort distribution. Figure 6 shows an assessment of SEL products for the period 1990-1993. The error rates, cost, and level of reuse are reexamined to determine if there was any change from the early baseline (1985-1989) shown previously in Figure 4.

These examples also reemphasize the need for baselining of both process and product. Without the basic understanding provided by the baseline, no change can be assessed.

5 - MEASURE ROI

Goals have been set. Baselines have been established. Investment in change has been made. Changes have been introduced and their effect assessed based on the original goals and the baseline values. Organizations must now determine if the results of change have been successful. Once 'success' has been determined, then they can attempt to answer the question, "Has the investment been worth it?"

CMM: The CMM measure of success is domain-independent and is the same as its generalized goal. An organization is successful if its process becomes mature and it becomes a continuously improving,

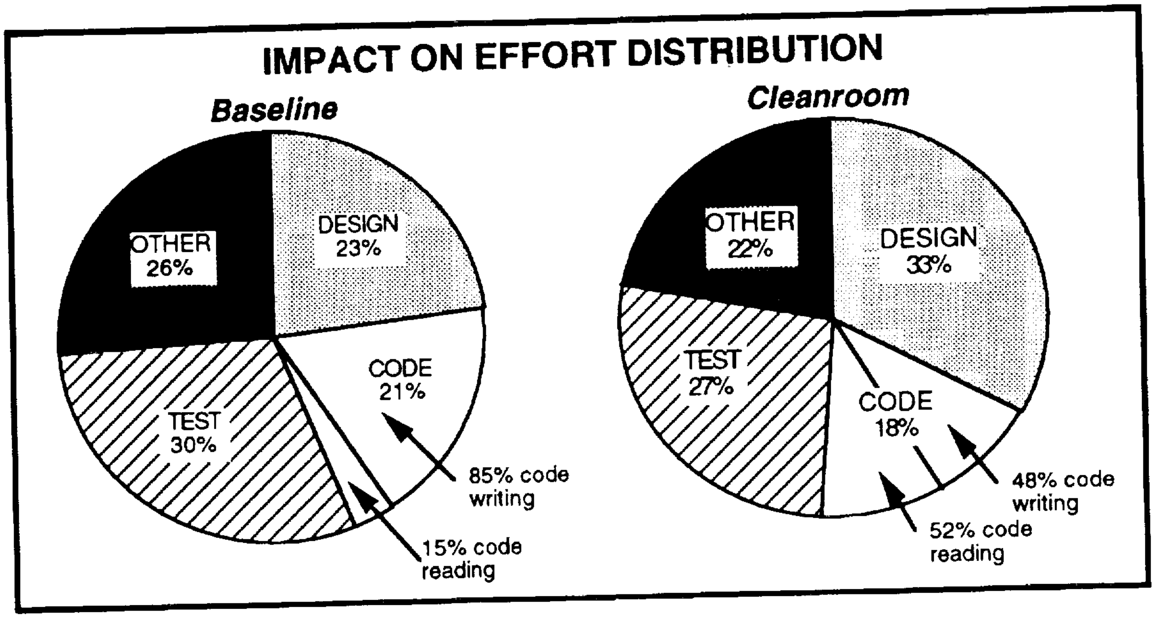


Figure 5. Sample SEL Process Assessment

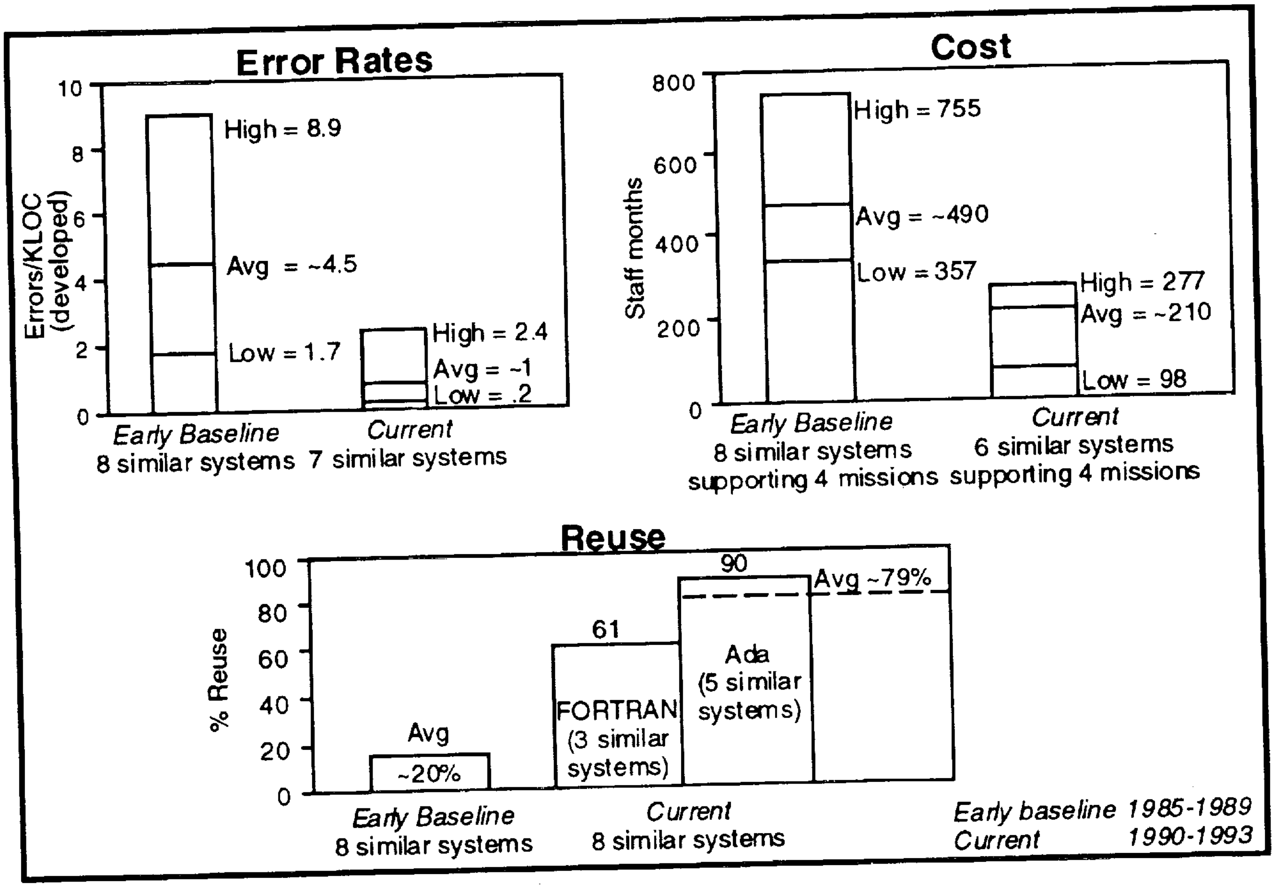


Figure 6. Sample SEL Product Assessment

optimizing process, a maturity level 5. Organizations progress to higher levels and, in doing so, expect to reduce risk and generate better products. Success can easily be determined, but how is ROI determined? After six years of experience with the application of the CMM, there is still no clear, accepted mechanism for determining the value of return for the investment required to implement software process improvement programs.

SEL: The SEL measure of success will vary from organization to organization. Success depends on the goals set by the individual organization. Success for Organization A is decreased error rates; success for Organization B is decreased cycle time. How is ROI determined? The baseline of both process and product and the assessment of change to the baseline result in quantified measures that can be examined to determine the ROI in process improvement. The remainder of this section discusses the ROI for the SEL's process improvement paradigm.

THE ROI FOR THE SEL

GSFC's Flight Dynamics Division (FDD) is the production organization with which the SEL is associated. FDD software development is driven and guided by the SEL process improvement paradigm. Over the past 18 years, the FDD has invested approximately 210 million dollars ($M) in software development activities. Of this amount, the FDD has invested approximately 11%, roughly $24M, in software process improvement. Figure 7 shows the breakdown of this investment. About 1.5% of the total investment (<$3M) is attributed to project overhead including form completion and collection, training, and other similar activities. Another 3% ($6M) was spent processing data: archiving data, maintaining the data base, quality assuring the data, etc. The largest part of this investment has been in analyzing the data. About 7% ($15M) has been spent defining experiments, analyzing results of SEL experiments, developing the SEL models and processes, producing software policies and standards (e.g., References 3 and 4), developing training material, carrying out training in changing processes, and other activities associated with improving the FDD's software products and process.

Has the FDD's 11% investment in software process improvement been worth it? In comparing projects developed in the mid 1980s to those developed in the early 1990s, several significant benefits have been achieved. Figure 6 depicted some of these results. The average level of reuse has increased by 300%, from ~20% to nearly 79% for similar classes of software. Reliability (errors/KSLOC) has improved significantly as the error rate has decreased by 75% from 4.5 to 1 error/KSLOC. The cost of developing Flight Dynamics software has also decreased significantly. The average cost of software per mission has decreased by 55% from ~490 staff months (SM) to ~210 SM.

These quantifiable improvements are complemented by more subjective ones. The SEL's process improvement activities have resulted in many impacts to the software production organization. First, the SEL integrates and focuses activities that were previously disparate. Training, standards, policies, technology insertion, and measurement have gradually become integrated as a result of the SEL's process improvement approach. Figure 8 depicts these items with respect to the three steps of the SEL improvement approach.

Second, there has been a cultural change within the production organization. The developers have become an integral part of process change. In fact, their experiences are the basis for process change. Developers have become more intimately involved with the SEL process improvement approach. For instance, developers on early Cleanroom experiments drove the development of a Cleanroom process handbook for use on later Cleanroom projects. By doing so, they packaged their experiences for future use. Another cultural change lies with the software being produced. Software development within the FDD is now process-driven and much less people-driven. The process is so

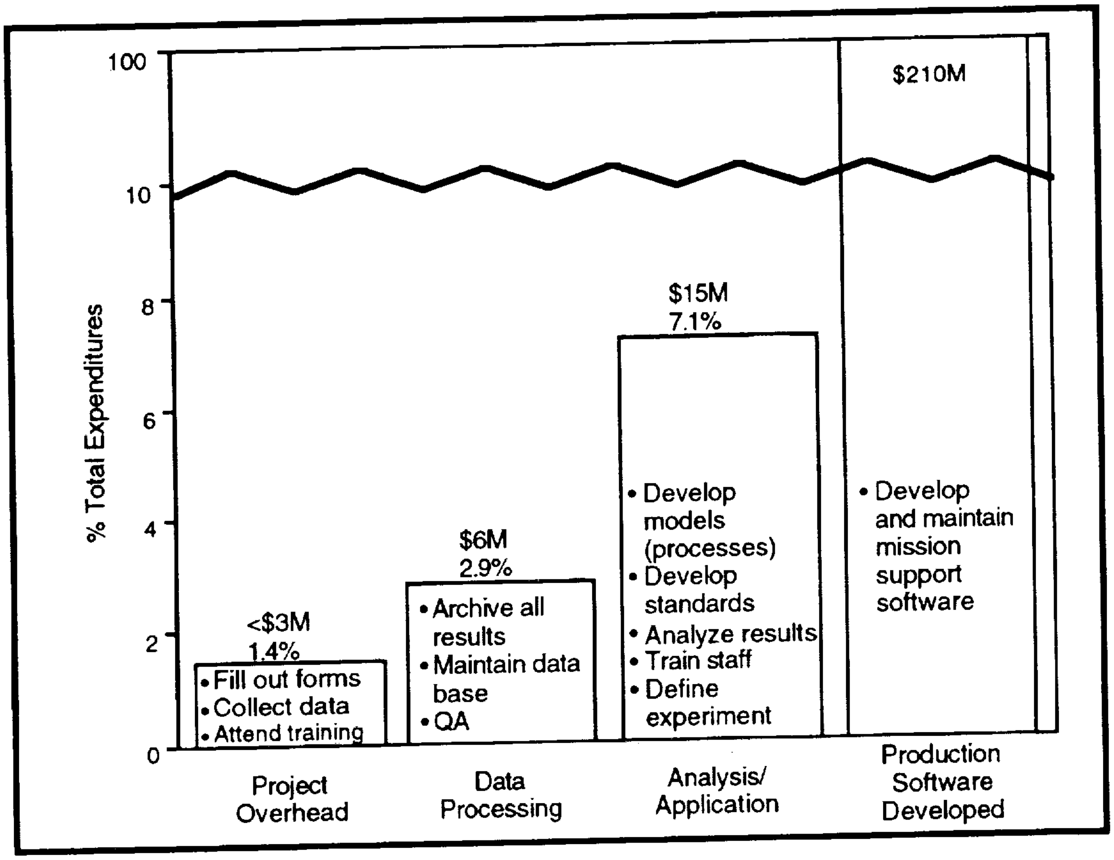


Figure 7. FDD Investment in the SEL and Software Process Improvement

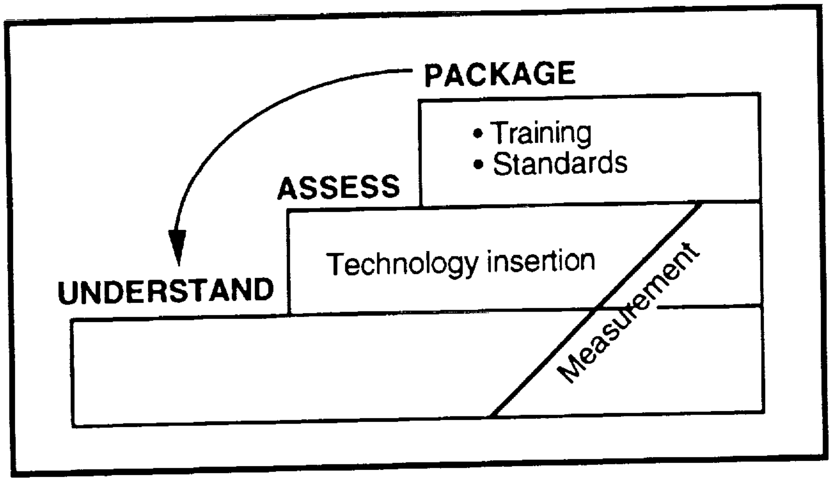


Figure 8. Integration of Software Activities via the SEL Process Improvement Paradigm

well-defined that 'heroes' are not necessary for the well being of a software project.

Finally, there is now a focused role of software engineering research. This research has become goal and product-driven rather than being performed in an ad hoc fashion. There is also a well-established mechanism by which experimentation, assessment, and adoption of technologies are performed. Within the SEL, process improvement has driven organizational evolution and optimized the allocation of software-related resources. While not quantifiable, these have been significant benefits achieved from the 11% investment in software process improvement.

Although there has been significant improvement in the software products in the SEL, there is no way of determining how much of this improvement is attributable to software process improvements and how much is attributable to normal improvements in technology. There have been significant changes to technology such as available tools, support environments, better operating systems, better trained personnel, work environments, faster machines, etc., but there has been no attempt by the SEL to distinguish between improvements driven by the technology maturation vs. software process maturation.

SUMMARY

As already discussed, there have been substantial benefits gained from the investment made in the SEL process improvement activities in the areas of level of reuse, reliability, and cost. While these benefits were being attained, the software being produced was also increasing in complexity (Figure 9). As a result of the SEL, the FDD was able to produce more complex software with more functionality while improving reliability and reducing cost.

The FDD's $24M investment over the past two decades has resulted in substantial benefits for the Division itself and many other organizations. As NASA focuses more on technology transfer, the latter may become a more significant factor in evaluating the ROI for the SEL. Not only has the SEL improved the software process and products of its own production organization, GSFC's Flight Dynamics Division, but it has shared these experiences with many other software organizations both within and outside the Agency. The impact on other organizations cannot be measured, but it certainly is a factor to be considered when determining the value added by the SEL and its process improvement paradigm.

REFERENCES

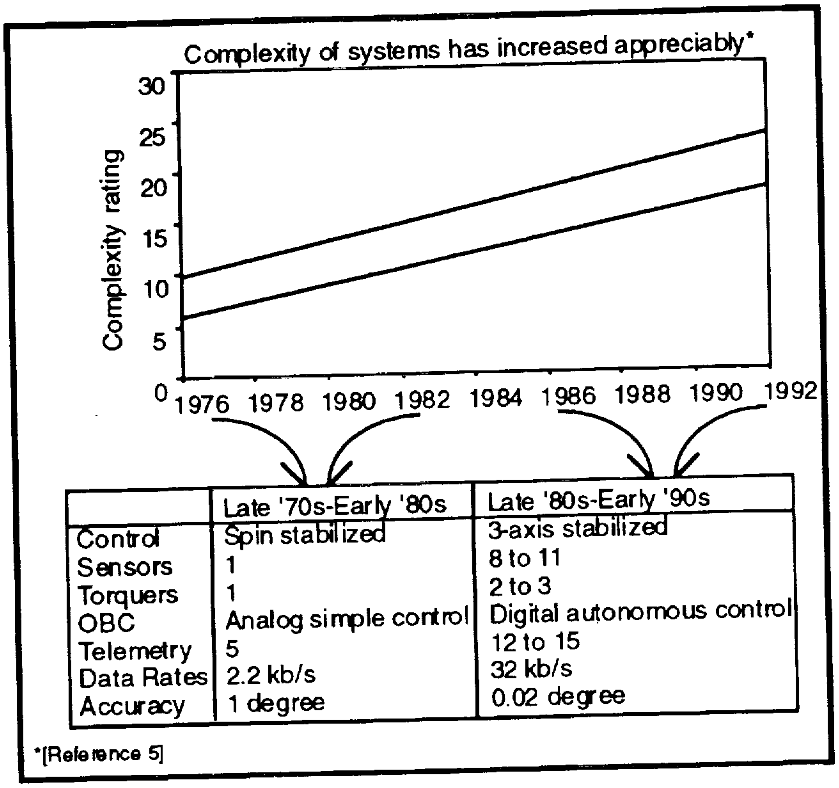
1. Paulk, M., B. Curtis, M. Chrissis, and C. Weber. *Capability Maturity Model for Software, Version 1.1,* Software Engineering Institute, Carnegie Mellon University, CMU/SEI-93-TR-24, February 1993.

2. *Software Engineering Laboratory (SEL) Cleanroom Process Model,* S. Green, NASA Goddard Space Flight Center, Software Engineering Laboratory, SEL-91-004, November 1991.

3. *Manager's Handbook for Software Development (Revision 1),* L. Landis, F. McGarry, R. Pajerski, S. Waligora, et al. NASA Goddard Space Flight Center, Software Engineering Laboratory, SEL-84-101, November 1990.

4. *Recommended Approach to Software Development (Revision 3),* L. Landis, F. McGarry, R. Pajerski, S. Waligora, et al. NASA Goddard Space Flight Center, Software Engineering Laboratory, SEL-81-305, June 1992.

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



	Late '70s-Early '80s	Late '80s-Early '90s
Control	Spin stabilized	3-axis stabilized
Sensors	1	8 to 11
Torquers	1	2 to 3
OBC	Analog simple control	Digital autonomous control
Telemetry	5	12 to 15
Data Rates	2.2 kb/s	32 kb/s
Accuracy	1 degree	0.02 degree

*[Reference 5]

Figure 9. FDD Software Complexity

PROCESS IMPROVEMENT AS AN INVESTMENT:
MEASURING ITS WORTH

Frank E. McGarry NASA/GSFC
Kellyann F. Jeletic NASA/GSFC

A846 001

MEASURING ROI* FOR PROCESS IMPROVEMENT

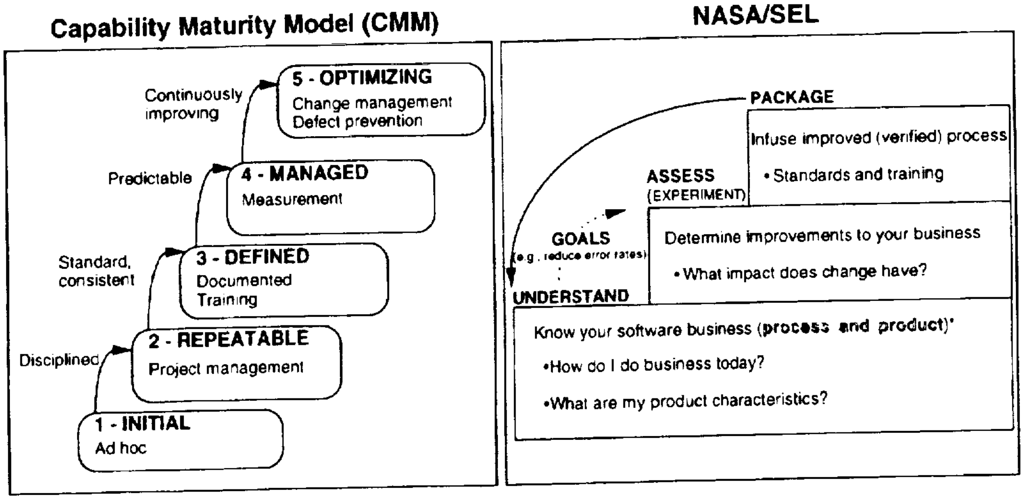
TOPICS OF DISCUSSION

1. CHARACTERISTICS OF TWO PROCESS IMPROVEMENT PARADIGMS
2. INFORMATION NEEDED TO DETERMINE ROI
3. MEASURING ROI IN NASA/SEL

*ROI = RETURN ON INVESTMENT

A846 002

TWO PARADIGMS FOR PROCESS IMPROVEMENT



Capability Maturity Model (CMM)

5 - OPTIMIZING
Change management
Defect prevention

Continuously improving

4 - MANAGED
Measurement

Predictable

3 - DEFINED
Documented
Training

Standard, consistent

2 - REPEATABLE
Project management

Disciplined

1 - INITIAL
Ad hoc

NASA/SEL

PACKAGE
Infuse improved (verified) process
• Standards and training

ASSESS
(EXPERIMENT)
Determine improvements to your business
• What impact does change have?

GOALS
(e.g. reduce error rates)

UNDERSTAND
Know your software business (process and product)*
•How do I do business today?
•What are my product characteristics?

* Process:
- How do you do business (e.g., standards and techniques used)
- Associated measurements (e.g., % time in testing, module size)

Product:
- End item attributes (e.g., error rates, productivity, complexity)

93M11U1.003

TWO PARADIGM FOR PROCESS IMPROVEMENT

STEPS	CMM EMPHASIS	NASA/SEL EMPHASIS
1. DEFINE GOALS	IMPROVE PROCESS (GET TO HIGHER LEVEL)	IMPROVE PRODUCT
2. PRODUCE BASELINE	PROCESS "ASSESSMENT" (AGAINST ONE YARDSTICK)	PROCESS AND PRODUCT UNDERSTANDING
3. INVEST IN CHANGE	COMMON YARDSTICK DRIVES CHANGE	EXPERIENCES AND GOALS DRIVE CHANGE
4. ASSESS CHANGE	REASSESS PROCESS SUCCESS - HIGHER LEVEL	REEXAMINE PROCESS AND PRODUCT SUCCESS - BETTER PRODUCT
5. MEASURE ROI		TODAY'S TOPIC

A846 006

SOFTWARE PROCESS IMPROVEMENT

STEP 1 - DEFINE GOALS

	CMM PARADIGM	NASA/SEL PARADIGM
	EMPHASIS -- IMPROVE PROCESS	EMPHASIS -- IMPROVE PRODUCT
ORGANIZATION 1	"GET TO LEVEL 5"	INCREASE **LEVEL OF REUSE**
ORGANIZATION 2	"GET TO LEVEL 5"	DECREASE **ERROR RATES**

SEL: IMPROVEMENT GOALS MAY VARY ACROSS DOMAINS

A846 007

SOFTWARE PROCESS IMPROVEMENT

STEP 2 - PRODUCE BASELINE

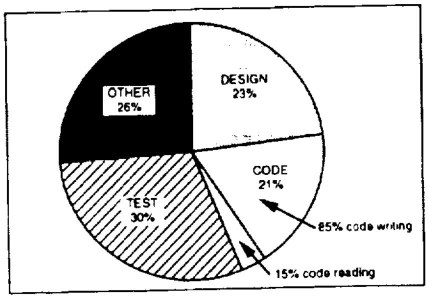
	CMM PARADIGM	NASA/SEL PARADIGM
	EMPHASIS -- PROCESS ASSESSMENT	EMPHASIS -- PROCESS AND PRODUCT UNDERSTANDING
ORG 1	STANDARDS - GOOD TRAINING - WEAK MEASUREMENT - WEAK ➧ LEVEL 1	WHAT IS YOUR DOMAIN? WHAT STANDARDS DO YOU USE? WHAT IS YOUR **LEVEL OF REUSE?**
ORG 2	STANDARDS - WEAK TRAINING - GOOD MEASUREMENT - GOOD ➧ LEVEL 3	WHAT IS YOUR DOMAIN? WHAT STANDARDS DO YOU USE? WHAT IS YOUR **ERROR RATE?**

SEL: MEASURES ARE DOMAIN-DEPENDENT
NO COMPARATIVE MEASURE ACROSS DOMAINS
CMM: HAS COMMON "YARDSTICK", CAN COMPARE ACROSS DOMAINS

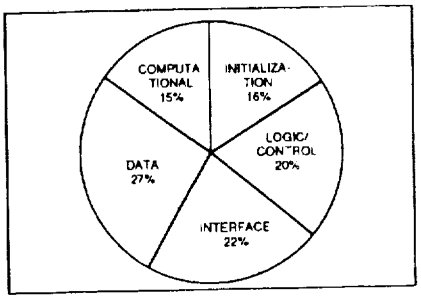
A846 008

NASA/SEL PROCESS BASELINE EXAMPLE

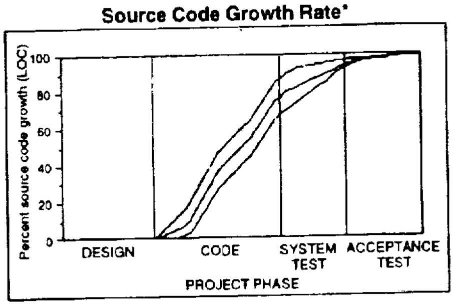
Effort Distribution*



Classes of Errors*



Source Code Growth Rate*

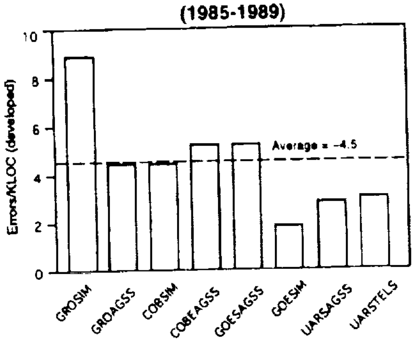


*Data from 11 Flight Dynamics projects (mid 1980s)

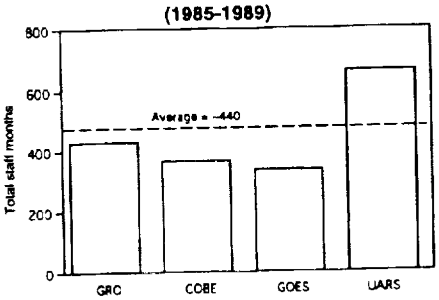
93M11U1.007

NASA/SEL PRODUCT BASELINE EXAMPLE

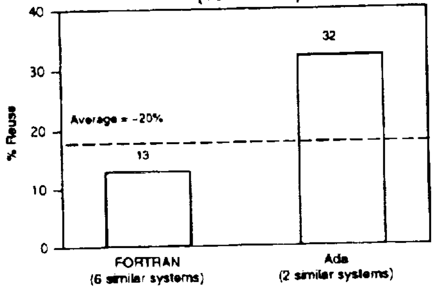
Error Rates (1985-1989)



Cost (1985-1989)



Reuse (1985-1989)



93M11U1.008

SOFTWARE PROCESS IMPROVEMENT

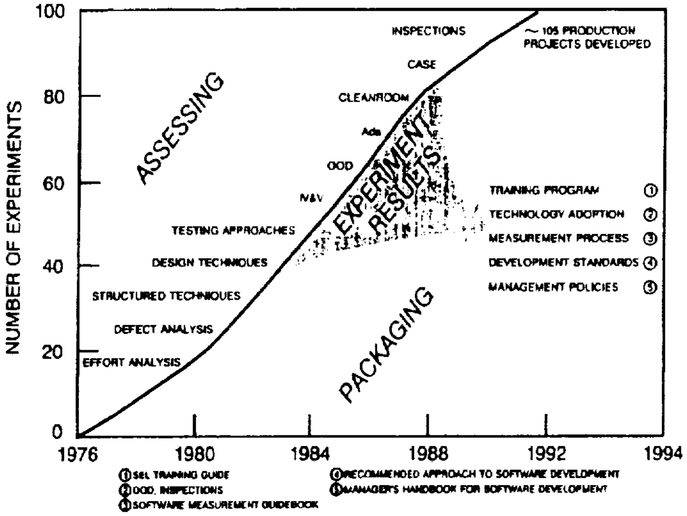
STEP 3 - INVEST IN CHANGE

	CMM PARADIGM	NASA/SEL PARADIGM
	EMPHASIS -- COMMON YARDSTICK DRIVES CHANGE	EMPHASIS -- EXPERIENCES AND GOALS DRIVE CHANGE
ORGANIZATION 1	MAINTAIN GOOD STANDARDS IMPROVE TRAINING IMPROVE MEASUREMENT	EXPERIMENT WITH OOD TO IMPROVE **REUSE** - TRAIN IN OOD
ORGANIZATION 2	WRITE BETTER STANDARDS MAINTAIN GOOD TRAINING MAINTAIN GOOD MEASUREMENT	EXPERIMENT WITH CLEANROOM FOR LOWER **ERROR RATES** - DEVELOP CLEANROOM PROCESS HANDBOOK

SEL: EACH DOMAIN MUST IDENTIFY MOST APPROPRIATE PROCESS CHANGE

A846.011

SOFTWARE PROCESS IMPROVEMENT NASA/SEL INVESTMENT IN CHANGE



EXPERIENCE DRIVES PROCESS CHANGE

A846.012

SOFTWARE PROCESS IMPROVEMENT

STEP 4 - ASSESS CHANGE

	CMM PARADIGM		NASA/SEL PARADIGM
	EMPHASIS -- REASSESS PROCESS		EMPHASIS -- REEXAMINE PROCESS AND PRODUCT
ORG 1	MAINTAINED GOOD STANDARDS IMPROVED TRAINING IMPROVED MEASUREMENT	➡ LEVEL 5*	VERIFY OOD IS USED VERIFY REUSE IS HIGHER**
ORG 2	IMPROVED STANDARDS MAINTAINED GOOD TRAINING MAINTAINED GOOD MEASUREMENT	➡ LEVEL 5*	VERIFY CLEANROOM IS USED VERIFY ERROR RATES LOWER**

*HOPEFULLY LEADS TO LOWER RISK **POSSIBLY LEADS TO LEVEL 5

SEL: SUCCESS IS MEASURED BY PRODUCT CHANGE (DOMAIN-DEPENDENT)
CMM: SUCCESS IS MEASURED BY PROCESS CHANGE (DOMAIN INDEPENDENT)

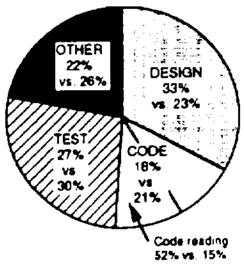
A846.013

ASSESSMENT OF CHANGES HAS PROCESS CHANGED?

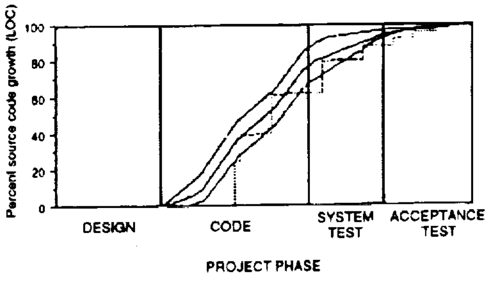
Effect of Cleanroom

IMPACT ON EFFORT DISTRIBUTION

Cleanroom vs. baseline



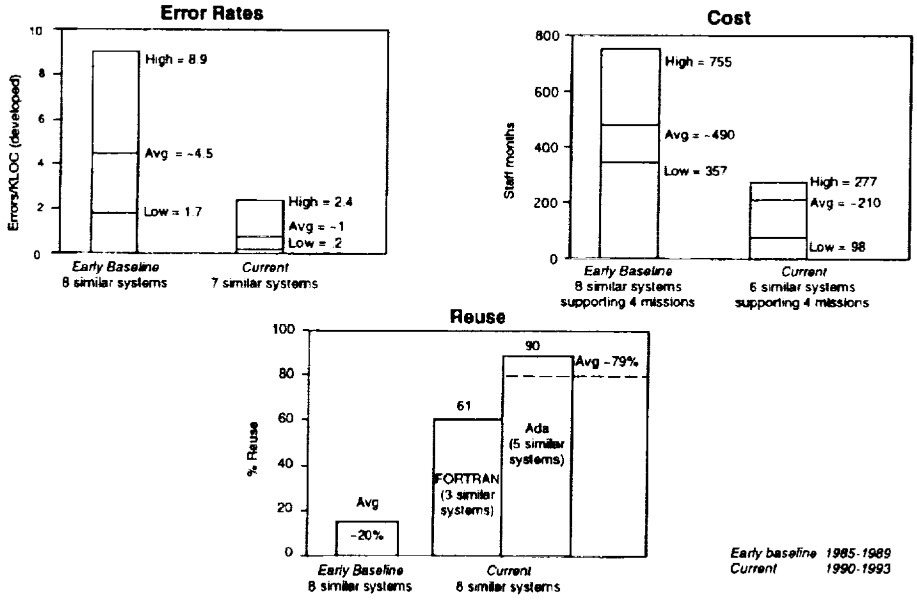
IMPACT ON SOURCE CODE (LOC) GROWTH RATE



Impact of changes are verified with process data vs. baseline

93M11U1.012

ASSESSMENT OF CHANGES
HAS PRODUCT IMPROVED?

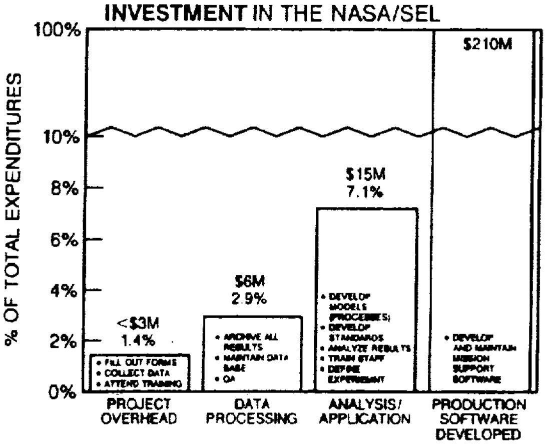


Improvement is measured against the goals of an organization

93M11U.013

SOFTWARE PROCESS IMPROVEMENT

STEP 5 - MEASURE ROI



COST OF SEL PROCESS IMPROVEMENT ACTIVITIES TOTAL ~11% OF ALL EXPENDITURES

A846 016

MEASURING RETURN ON INVESTMENT IN THE SEL
(BASED ON CHANGES FROM MID 80s TO EARLY 90s)

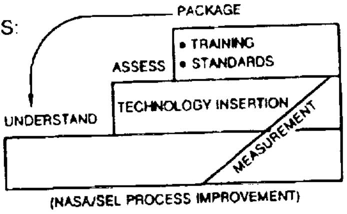
• RELIABILITY	Errors/KSLOC	down by 75%	(from 4.5 to 1)
• REUSE	Average level of code reuse	increased by 300%	(from ~20% to ~80%)
• DEVELOPMENT COST	Average mission cost*	down ~55%	(from 490 to 210 staff mos)

Investment in product-driven goals enables direct measurement of return

*Reflects reuse change

93M11U.015

MEASURING ROI FOR PROCESS IMPROVEMENT OBSERVED ORGANIZATIONAL IMPACTS



(NASA/SEL PROCESS IMPROVEMENT)

- DRIVES INTEGRATION OF PREVIOUSLY DISPARATE ACTIVITIES:
  - TRAINING
  - STANDARDS, POLICIES
  - TECHNOLOGY INSERTION
  - MEASUREMENT

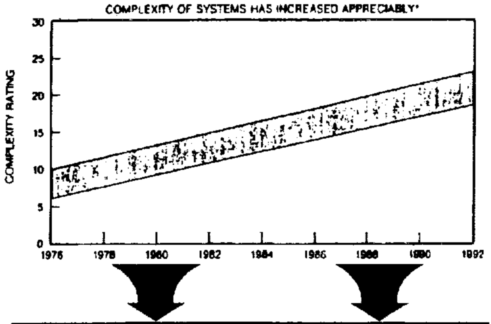
- ENHANCES ROLE OF ORGANIZATIONAL ELEMENTS (CULTURAL CHANGE)
  - DEVELOPERS -- BECOME AN INTEGRAL PART OF PROCESS CHANGE
  - SOFTWARE -- PROCESS DRIVEN (LESS PEOPLE DRIVEN)

- FOCUSES ROLE OF SOFTWARE ENGINEERING RESEARCH
  - BECOMES GOAL/PROBLEM DRIVEN
  - EXISTS MEANS TO EXPERIMENT, ASSESS, ADOPT

PROCESS IMPROVEMENT WILL DRIVE ORGANIZATIONAL EVOLUTION AND OPTIMIZE ALLOCATION OF SOFTWARE-RELATED RESOURCES

A846.17A

MEASURING ROI FOR PROCESS IMPROVEMENT



	LATE '70s-EARLY '80s	LATE '80s-EARLY '90s
CONTROL	SPIN STABILIZED	3 AXIS STABILIZED
SENSORS	1	8 TO 11
TORQUERS	1	2 TO 3
OBC	ANALOG SIMPLE CONTROL	DIGITAL AUTONOMOUS CONTROL
TELEMETRY	5	12 TO 15
DATA RATES	2.2 kb/s	32 kb/s
ACCURACY	1 degree	0.02 degree

THE NASA/SEL IS PRODUCING MORE FUNCTIONALITY, FOR MORE COMPLEX SYSTEMS, WITH HIGHER RELIABILITY, AT SIGNIFICANTLY LOWER COST.

*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

A846 018

N94- 36487

RECENT SEL EXPERIMENTS AND STUDIES

53-61
12685
p. 14

Rose Pajerski
Donald Smith

SOFTWARE ENGINEERING BRANCH
Code 552
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771

INTRODUCTION

Since 1976, the Software Engineering Laboratory (SEL) has been dedicated to understanding and improving the way in which its organization, the Flight Dynamics Division (FDD) of NASA/Goddard Space Flight Center, develops, maintains, and manages complex flight dynamics software systems. During the past 17 years, the SEL has collected and archived data from over 100 software development projects in the organization. From these data, the SEL has derived models of the development process and product, and conducted studies on the impact of new technologies.

One of the SEL's overall goals is to treat each development effort as a SEL experiment that examines a specific technology or builds a model of interest. The SEL has undertaken many technology studies while developing operational support systems for numerous NASA spacecraft missions. Viewgraph 2 is a rough mapping of spacecraft missions and support software systems (on the top) to SEL studies (on the bottom) over the past 17 years. Measures and experiences from these development projects have been saved and used to understand, characterize, and improve the development environment.

The SEL's basic approach to software process improvement is first to understand and characterize the process and product as they exist to establish a local baseline. Only then can new technologies be introduced and assessed (phase two) with regard to both process changes and product impacts. The third phase synthesizes the results of the first two phases into various packages such as process and product models, training materials, and tools and guidebooks. These products are then fed back into the development environment for subsequent projects to use and benefit from. Viewgraph 4 illustrates the SEL three-phase process improvement model.

The SEL organization consists of three functional areas: software project personnel, database support personnel, and software engineering analysts. The largest part of the SEL is the 250-plus software personnel who are responsible for the development and maintenance of over 4 million source lines of code (SLOC) that provide orbit and attitude ground support for all GSFC missions. Since the SEL was founded, software project personnel have provided software measurement data on over 100 projects. These data have been collected by the database support personnel to be stored in the SEL database for use by both the software project personnel and the software engineering analysts. The database support staff enter measurement data into the SEL database, quality assure the data, and maintain the database and its reports. The software engineering analysts define the experiments and studies, analyze the data, and produce reports which now number over 300. These reports affect such things as project standards, development procedures, and project management guidelines.

The SEL has been a fairly stable organization; however one significant change has occurred recently. About a year ago, the FDD reorganized and the SEL became responsible for the maintenance of all operational software in the environment. Along with consolidating development and maintenance into one organization, all acceptance testing was added. Now one organization, the SEL, is responsible for a significant portion of the development life cycle, from requirements analysis through maintenance.

The SEL's new areas of responsibility, maintenance and testing, have become focal points for current studies which correspond to the three phases of the SEL improvement model: (1) understanding maintenance processes and products, (2) assessing the effectiveness of different testing approaches, and (3) analyzing our locally derived cost and schedule estimation models to determine if they need updating. These studies will be discussed in reverse sequence, however, going from the most complete data to the most preliminary findings.

SEL COST AND SCHEDULE ESTIMATION MODELS

Over the past several years, organizational and technology changes have occurred in the FDD software development environment that may have affected many of the SEL's baseline models that characterize some of the products and processes of the FDD. The purpose of the cost and schedule estimation study[1]was to update the SEL cost and schedule estimation models. Thirty-nine FDD development projects measured over 15 years were examined. Specific factors which may have an impact on cost and schedule were analyzed. These factors included code reuse, language, application type, and subjective data such as team experience level and technology usage.

Cost Estimation Factors

The SEL baseline cost model is based on software size, productivity, and a weighting term, the growth factor:

Effort = size / productivity x
growth factor

In this environment, managers compare current system requirements with historical experience on earlier systems to estimate initial software size in terms of both new SLOC and reused SLOC. Productivity values are also based on data from previous representative development efforts. While size and productivity are commonly used terms, the definition and usage of the growth factor in the above expression is unique to our environment. Previous experience had shown that the size of a software system will grow by 40% from the time the requirements are baselined at the Software Requirements Review to when the system becomes operational. This growth is due to early uncertainties in operational support scenarios as well as changes in the spacecraft hardware which result in software requirements changes.

The current study results (viewgraph 7) indicate that the basic effort model is still valid but that accuracy can be improved by including new weighting factors based on language and reuse levels. The language selected, FORTRAN or Ada, has an impact on both the cost of reusing code and the productivity values to be applied. For FORTRAN reuse, a 20% cost multiplier is used while the Ada reuse multiplier is 30%. The productivity values for FORTRAN and Ada are 3.5 and 5.0, respectively. The software growth factor is also affected by the level of reuse. The 40% value still holds for low-reuse systems but decreases by 15% for high-reuse systems.

Schedule Estimation Factors

The study concluded that the only quantifiable factor affecting schedule estimates was the application type. There are two application types in the FDD environment, operational ground systems and simulators. The simulators, which usually begin development with significant requirements undefined, take about 35% longer to develop than the operational ground systems. Neither the cost nor schedule models contain any subjective factors, although the study did look at the effects of subjective data on these models. (Jon Valett's report in Session 3 discusses the results of that part of the study.)

SEL TESTING STUDIES

Over the years, several testing methods have been used in the FDD environment. The goal of the ongoing SEL testing study is to determine the relative effectiveness of each method by examining key product and process measures such as effort and error rates. The testing approaches include:

- SEL Standard Test Process
- Independent Verification and Validation (IV&V)
- SEL Cleanroom
- Independent Test Team

The organizational boundaries and key process elements of each approach are described below and are tabulated in viewgraph 10.

- The SEL standard test process involves two separate organizations, the development group and an acceptance test (AT) group. The developers implement and system test the software by integrating and verifying the end-to-end system flow. Then the final build of the software is turned over to a separate AT group that performs functional acceptance testing based on the requirements.
- The IV&V approach added an independent test group to the standard process. That group worked in parallel with the development and AT groups to completely test the software as well as to verify the requirements and operational scenarios for the system.
- The SEL Cleanroom methodology has separate development and test teams that develop and test, respectively, in builds. The test team generates the test cases using a statistical method based on the frequency of activities of various operational scenarios. The final build of the software is then passed to a different organization for acceptance testing.
- The most recent approach to be applied uses an independent test team and is a direct consequence of the organizational changes within the FDD. This approach has separate development and test teams under the same organization. The test team handles all the build testing of the software using functional test case selection.

The IV&V approach will be discussed briefly because of its relevance to NASA programs today. The other three approaches will be contrasted and compared with one another in terms of some key process and product measures. The process measure that will be used is *effort distribution by activity* and the product measure is the *error rate* through development and testing.

IV&V Test Study

A recent National Research Council (NRC) report[2] recommends that IV&V be part of the standard NASA testing process. This recommendation may be appropriate for the Space Shuttle software that was studied in the NRC report; however it is certainly not appropriate for all NASA software. Results of SEL IV&V experiments[3] conducted in the early 1980s in the FDD environment were not positive. Although one of the expectations for IV&V within the environment had been increased software reliability, the study found that the error rates were not favorably impacted and that total development cost of the software increased significantly (between 30% and 60%). Consequently, it was determined that IV&V was not appropriate for adoption in the FDD.

Process Measures Comparison

Using effort distributions to compare test approaches is an effective way to identify process differences. The point of interest is to see if there is any apparent impact or observable change when compared to the SEL standard process. Viewgraph 12 shows the distribution of effort involved in design and code versus test for the three test methods. There is little difference between the SEL Standard and Cleanroom effort distributions; however, the independent test team approach does display a very different effort distribution. Simply using process measures will not determine which of these approaches is "best" but it does highlight process differences.

Product Measures Comparison

Examining the average error rates as recorded from the design phase through the acceptance test phase is a good way to determine the effectiveness of testing methods. The example in viewgraph 13 shows two ways of viewing error rate data on the same set of project data.

The chart on the left is the average error detection rates grouped by testing approach. Two points of interest are noted:

1) Cleanroom stands out as having a higher error rate than the other two methods. However, this is somewhat misleading because the Cleanroom process includes different types of errors in the error statistics. Previous studies[4] have shown that the Cleanroom process actually produces error rates that are lower than the FDD baseline for certain classes of projects.
2) The independent test team approach shows lower error rates, indicating that this approach shows some promise and deserves further study.

The chart on the right groups the data by low- and high-reuse projects. There is indication that factors such as the level of reuse may affect error rates.

The testing effectiveness study is not yet complete because other process and product measures need to be assessed, including an evaluation of data collected during the maintenance phase.

SEL MAINTENANCE STUDIES

The first area of focus by the SEL in the maintenance arena is to build a baseline understanding of maintenance products and processes such as software characteristics, effort distributions, and change and error profiles. Understanding these elements will enable cost and schedule estimation models to be built, which is one of the future goals of studying maintenance. The number of systems and the size of the software being maintained varies: 105 systems ranging in size from 10,000 SLOC to 250,000 SLOC totaling 3.5 M lines of code. A high percentage cf these systems are FORTRAN mainframe systems so they are the first ones to be analyzed in the study. Information learned so far is based on data from a handful of these systems, so the maintenance baseline presented (in viewgraphs 14-16) is considered a preliminary characterization of the maintenance process and products.

FDD is currently maintaining two types of systems: multimission systems, which support many spacecraft and have a software lifetime of 10-30 years, and single-mission support systems, which run as long as the spacecraft is operational (typically 2-7 years).

Error and Change Characteristics

Preliminary studies reveal that although the software sizes are similar, these two types of systems show very different error characteristics after 5 years of operations (viewgraph 15). For multimission systems, the error rate is an order of magnitude higher than the rate for single-mission systems. There are many possible explanations for this. The multimission systems are used more and they are also updated with more frequent enhancements. If the difference in error rates shown here is confirmed by further analysis, the reasons for it will need to be examined and evaluated.

Also shown in the viewgraph are two pie charts depicting two types of change distributions:

- Change type (right chart) as determined by the requestor on the change request forms: about 25% of the changes are enhancements and 75% are error corrections with less than 1% being adaptations (changes due to operating system or compiler upgrades).
- Effort distribution (left chart) as determined by the maintainers in satisfying the change requests: about 66% of the maintainers' time is spent on implementing enhancements and the remainder is spent correcting errors and adapting software.

Effort Characteristics

Cost (effort) is another important element for understanding software maintenance. Viewgraph 16 displays several cost characteristics, again divided into multimission and single-mission systems.

The cost to initially develop these systems is about the same for both types of system; however, the cost of maintenance varies. Maintenance costs on the multimission systems are running about 3 staff years per year, or about 10% of the development cost per year. (The 10% figure has been used as a rule of thumb in the FDD for many years in calculating maintenance costs, so it is interesting to see it confirmed with some recent data). The yearly maintenance cost for single-mission systems is currently running at about 2% of development cost, which is probably due to these systems being enhanced less often.

Another way of understanding an unfamiliar process such as maintenance is to compare it with an established process that has been baselined. The pie charts on viewgraph 15 compare the maintenance and development processes in terms of effort distribution. The biggest difference is the relative effort spent in testing activities: 30% for development and 5% for maintenance. This difference certainly must be probed further in defining a baseline understanding of maintenance processes and products in the FDD environment.

SUMMARY

The studies discussed in this paper are all examples of activities that are performed as part of the SEL's process improvement model. Using this model, the SEL starts by understanding the product and process, then assesses the impact of new technologies, and finally packages what was learned. The preliminary examination of maintenance effort, error, and change profiles to establish a maintenance baseline exemplifies understanding-phase activities. The ongoing testing study that is examining the effects of various testing approaches on process and product measures is an example of typical assessing efforts. Finally, the derivation of cost and schedule estimation models from locally driven factors such as reuse level, application type, and language is an example of experience packaging.

In the SEL, no study is ever really completed. Studies will be repeated and iterated upon in the future as part of the ongoing software improvement process.

REFERENCES

1. Condon, S., et al., *Cost and Schedule Estimation Study Report,* SEL-93-002, November 1993

2. *An Assessment of Space Shuttle Flight Software Development Process,* National Research Council, 1993

3. Page J., "Methodology Evaluation: Effects of Independent Verification and Integration On One Class of Application," *Proceedings of the Sixth Annual Software Engineering Workshop,* SEL-81-013, December 1981

4. Green, S. E., and R. Pajerski, Cleanroom Process Evolution in the SEL," *Proceedings of the Sixteenth Annual Software Engineering Workshop,* SEL-91-006, December 1991

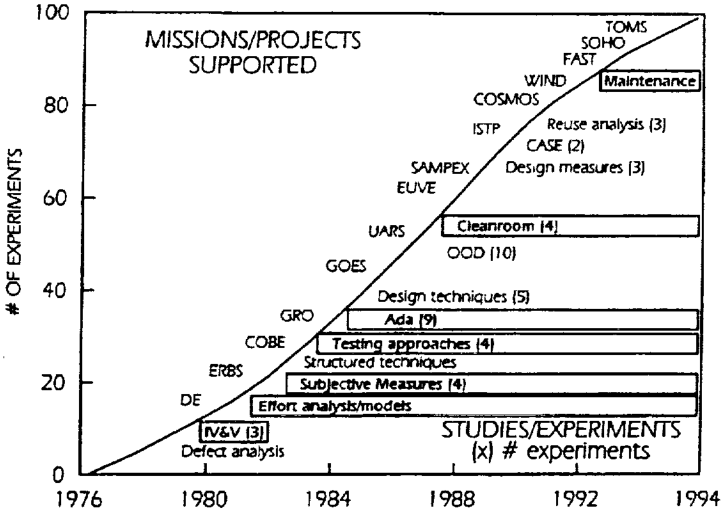
RECENT SEL EXPERIMENTS AND STUDIES

18TH ANNUAL
SOFTWARE ENGINEERING
WORKSHOP

ROSE PAJERSKI
DONALD SMITH
NASA/GSFC

B649.001

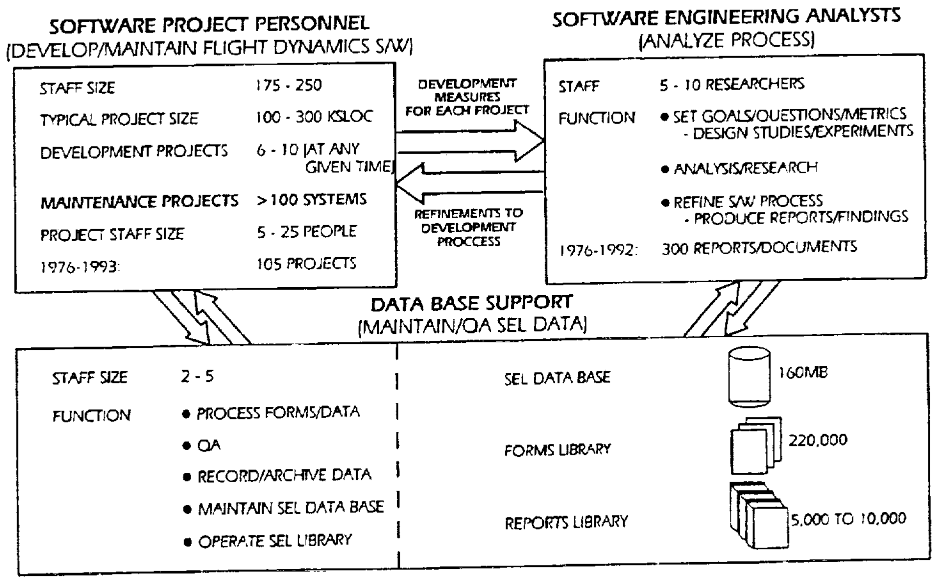
STUDIES ON OVER 100 PRODUCTION PROJECTS



EVERY DEVELOPMENT PROJECT CONTRIBUTES TO SEL RESULTS

B649.002

SEL ORGANIZATIONAL STRUCTURE



SOFTWARE PROJECT PERSONNEL (DEVELOP/MAINTAIN FLIGHT DYNAMICS S/W)

STAFF SIZE	175 - 250
TYPICAL PROJECT SIZE	100 - 300 KSLOC
DEVELOPMENT PROJECTS	6 - 10 (AT ANY GIVEN TIME)
MAINTENANCE PROJECTS	>100 SYSTEMS
PROJECT STAFF SIZE	5 - 25 PEOPLE
1976-1993:	105 PROJECTS

DEVELOPMENT MEASURES FOR EACH PROJECT

REFINEMENTS TO DEVELOPMENT PROCCESS

SOFTWARE ENGINEERING ANALYSTS (ANALYZE PROCESS)

STAFF	5 - 10 RESEARCHERS
FUNCTION	• SET GOALS/QUESTIONS/METRICS - DESIGN STUDIES/EXPERIMENTS • ANALYSIS/RESEARCH • REFINE S/W PROCESS - PRODUCE REPORTS/FINDINGS
1976-1992:	300 REPORTS/DOCUMENTS

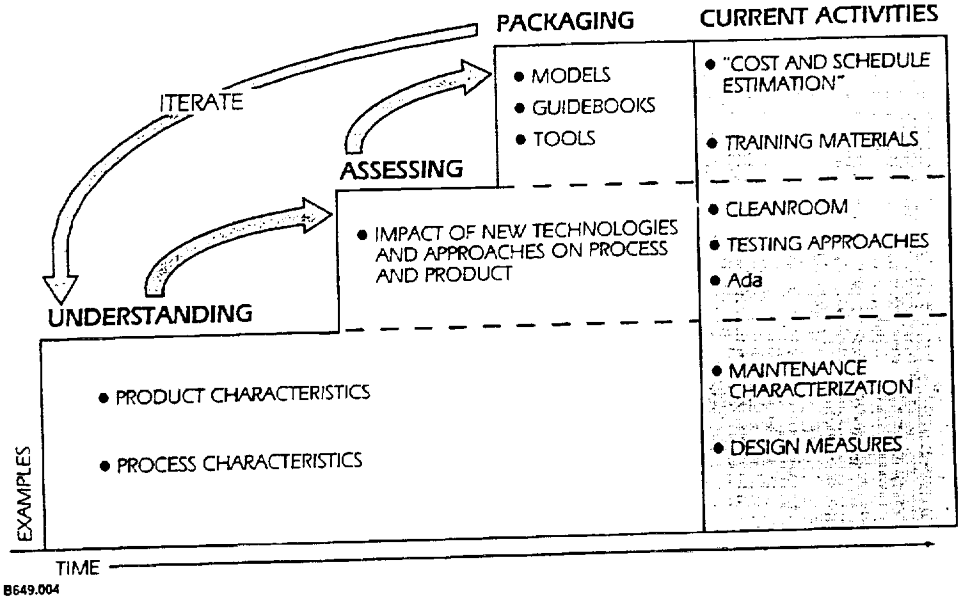
DATA BASE SUPPORT (MAINTAIN/QA SEL DATA)

STAFF SIZE	2 - 5
FUNCTION	• PROCESS FORMS/DATA • QA • RECORD/ARCHIVE DATA • MAINTAIN SEL DATA BASE • OPERATE SEL LIBRARY

SEL DATA BASE	160MB
FORMS LIBRARY	220,000
REPORTS LIBRARY	5,000 TO 10,000

B649.003

SEL IMPROVEMENT MODEL - CURRENT ACTIVITIES



ITERATE

PACKAGING
- MODELS
- GUIDEBOOKS
- TOOLS

ASSESSING
- IMPACT OF NEW TECHNOLOGIES AND APPROACHES ON PROCESS AND PRODUCT

UNDERSTANDING
- PRODUCT CHARACTERISTICS
- PROCESS CHARACTERISTICS

CURRENT ACTIVITIES
- "COST AND SCHEDULE ESTIMATION"
- TRAINING MATERIALS
- CLEANROOM
- TESTING APPROACHES
- Ada
- MAINTENANCE CHARACTERIZATION
- DESIGN MEASURES

EXAMPLES

TIME

B649.004

SUMMARY OF RECENT SEL STUDIES

PLAN	• COST AND SCHEDULE ESTIMATION STUDY ①
	• SUBJECTIVE FACTORS EVALUATION • NASA DOMAIN PROFILES • TECHNOLOGY TRANSFER
TRAINING	• RECOMMENDED APPROACH TO SOFTWARE DEVELOPMENT • GOVERNMENT/CONTRACTOR TASK MANAGEMENT • FLIGHT DYNAMICS APPLICATIONS
IMPLEMENT	• DESIGN MEASURES • CASE • Ada • OOD
TEST	• COMPARISON OF APPROCHES - INDEPENDENT VERIFICATION AND VALIDATION SEL STANDARD ② CLEANROOM INDEPENDENT TEST TEAM
MAINTENANCE	• PROCESS UNDERSTANDING • PRODUCT CHARACTERIZATION ③

B649.005

① COST AND SCHEDULE STUDY*

GOAL	UPDATE SEL BASELINE	EFFORT ESTIMATION MODEL SCHEDULE ESTIMATION MODEL
	DETERMINE IMPACTS OF	REUSE (CODE) LANGUAGE APPLICATION TYPE SUBJECTIVE FACTORS (EXPERIENCE, TECHNOLOGY)

STUDY PARAMETERS

39	PROJECTS (1977-1992)
2	LANGUAGES (FORTRAN, Ada)
2	APPLICATION TYPES
20K-300K SOURCE LINES OF CODE	

* "SOFTWARE ENGINEERING LABORATORY COST AND SCHEDULE ESTIMATION STUDY REPORT," S. CONDON, M. REGARDIE, S.WALIGORA, SEPTEMBER 1993

B649.006

① WHAT IMPACTS COST?

EFFORT = SIZE/PRODUCTIVITY x GROWTH FACTOR

$$\text{EFFORT (Ada)} = \frac{\text{(NEW SLOC + 30\% REUSED SLOC)}}{\text{PRODUCTIVITY (Ada)}} \times \begin{pmatrix} \text{1.4 (REUSE} < \text{70\%)} \\ \text{1.2 (REUSE} \geq \text{70\%)} \end{pmatrix}$$

$$\text{EFFORT (FORTRAN)} = \frac{\text{(NEW SLOC + 20\% REUSED SLOC)}}{\text{PRODUCTIVITY (FORTRAN)}} \times \begin{pmatrix} \text{1.4 (REUSE} < \text{70\%)} \\ \text{1.2 (REUSE} \geq \text{70\%)} \end{pmatrix}$$

COSTS 50% MORE TO REUSE A LINE OF Ada CODE THAN A LINE OF FORTRAN

SOFTWARE SIZE GROWTH IS 15% LOWER FOR HIGH REUSE SYSTEMS

B649.007

① WHAT IMPACTS SCHEDULE?

$$\text{SCHEDULE} = \text{COEFF} \times \text{(EFFORT)}^{0.3}$$

$$\text{SCHEDULE (GROUND SYSTEMS)} = 5.0 \times \text{(EFFORT)}^{0.3}$$

$$\text{SCHEDULE (SIMULATORS)} = 6.7 \times \text{(EFFORT)}^{0.3}$$

SCHEDULE IMPACTED BY APPLICATION TYPE, NOT BY LANGUAGE OR REUSE LEVEL

B649.008

② EXPERIMENTS IN TESTING

GOAL

- ● ASSESS THE IMPACT OF ORGANIZATIONAL CHANGES ON SEL PROCESSES

- ● COMPARE TEST APPROACHES

- INDEPENDENT VERIFICATION AND VALIDATION	1982 STUDY
- SEL STANDARD TEST PROCESS	1978 - CURRENT
- SEL CLEANROOM	1986 - CURRENT
- INDEPENDENT TEST TEAM	1992 - CURRENT

- ● ASSESS EFFORT AND ERROR DISTRIBUTIONS TO DETERMINE TESTING EFFECTIVENESS

B649.009

② OVERVIEW OF FOUR TEST APPROACHES

SEL STANDARD PROCESS (24 PROJECTS)

REQUIREMENTS	DEVELOPMENT	SYSTEM TEST	ACCEPTANCE TEST
	CODE READING	END-TO-END FLOW	FUNCTIONAL, REQUIREMENTS BASED

IV & V (3 PROJECTS)

REQUIREMENTS	DEVELOPMENT	SYSTEM TEST	ACCEPTANCE TEST
	IV & V TEST TEAM		

SEL CLEANROOM (4 PROJECTS)

REQUIREMENTS	DEVELOPMENT/TEST	ACCEPTANCE TEST
	STATISTICAL CASES, CODE READING	FUNCTIONAL, REQUIREMENTS BASED

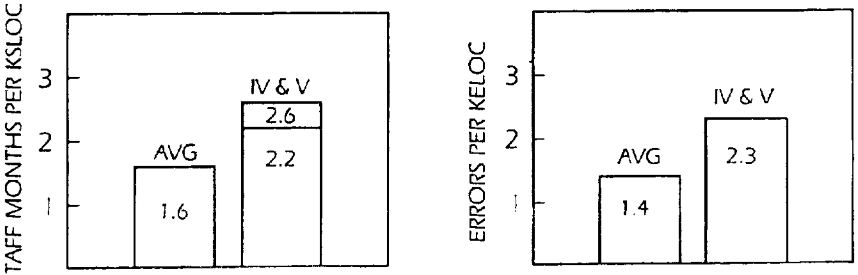
INDEPENDENT TEST TEAM (3 PROJECTS)

REQUIREMENTS	DEVELOPMENT/TEST	ACCEPTANCE TEST
	FUNCTIONAL CASES BY BUILD REQUIREMENTS BASED	

B649.010

C-2.

② IV & V TEST APPROACH



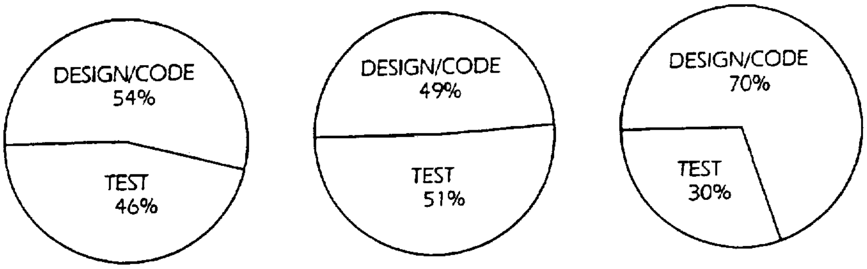
COST INCREASED BY 30-60%

RELIABILITY DIDN'T INCREASE

DOMAIN AND PRODUCT GOALS ARE KEY DRIVERS FOR PROCESS CHANGES

B649.011

② TEST EFFORT DISTRIBUTION (BY ACTIVITY)



SEL STANDARD

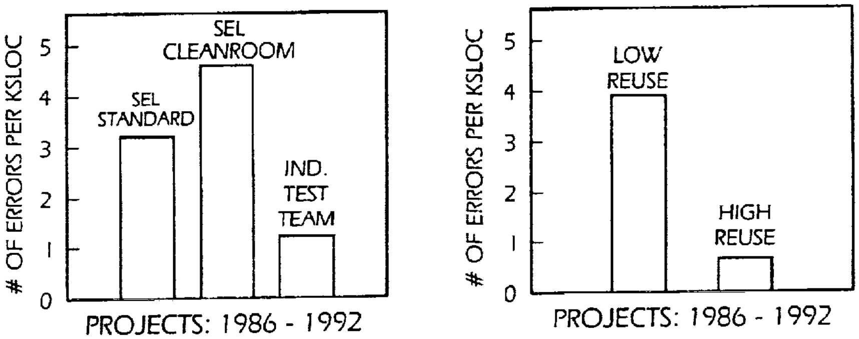
SEL CLEANROOM

IND. TEST TEAM

INDEPENDENT TEST TEAM APPROACH IMPACTING TEST PROCESS

B649.012

② ERROR DETECTION RATES - TWO VIEWS
(DESIGN THROUGH ACCEPTANCE TEST)



TEST APPROACH AND REUSE LEVEL
IMPACT ERROR RATES

B649.013

③ MAINTENANCE STUDY

CURRENT FOCUS	BUILD BASELINE UNDERSTANDING	SOFTWARE CHARACTERISTICS EFFORT DISTRIBUTIONS ERROR/CHANGE PROFILES ESTIMATION MODELS

OPERATIONAL SYSTEMS UNDER MAINTENANCE*

105 SYSTEMS	RANGE FROM 10 KSLOC TO 250 KSLOC TOTALS 3.5 MILLION SLOC		
LANGUAGES	85% FORTRAN	10% Ada	5% OTHER
PROCESSORS	80% MAINFRAME	5% OTHER	10% PC/W/KSTN

MAINTENANCE ACTIVITY PRIMARILY SUPPORTS
FORTRAN MAINFRAME OPERATIONAL ENVIRONMENT

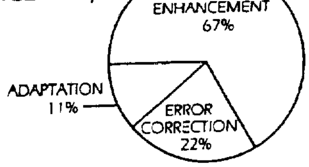
* MAINTENANCE INCLUDES ALL ACTIVITY AFTER OPERATIONAL START

B649.014

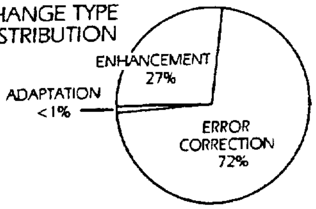
③ MAINTENANCE CHANGE/ERROR PROFILES

SYSTEM TYPE (SIZE IN KSLOC)	ERRORS DETECTED (5 YEARS OF OPERATIONS)	# DAILY USES
MULTI-MISSION (~200 KSLOC)	1.5 ERRORS/KSLOC	20-40
SINGLE MISSION (~150 KSLOC)	0.1 ERRORS/KSLOC	1-5

EFFORT DISTRIBUTION (BY CHANGE TYPE)



CHANGE TYPE DISTRIBUTION



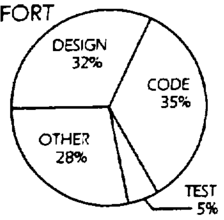
CHANGE EFFORT DISTRIBUTION NOT PROPORTIONAL TO NUMBER OF CHANGES

B649.016

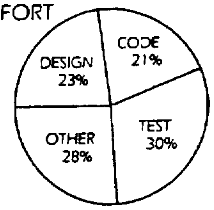
③ MAINTENANCE EFFORT

SYSTEM TYPE (SIZE IN KSLOC)	COST TO DEVELOP (STAFF YEARS(SY))	COST TO MAINTAIN (SY/YEAR)
MULTI-MISSION (~200 KSLOC)	30-40	3.0
SINGLE MISSION (~150 KSLOC)	25-35	0.5

MAINTENANCE EFFORT DISTRIBUTION



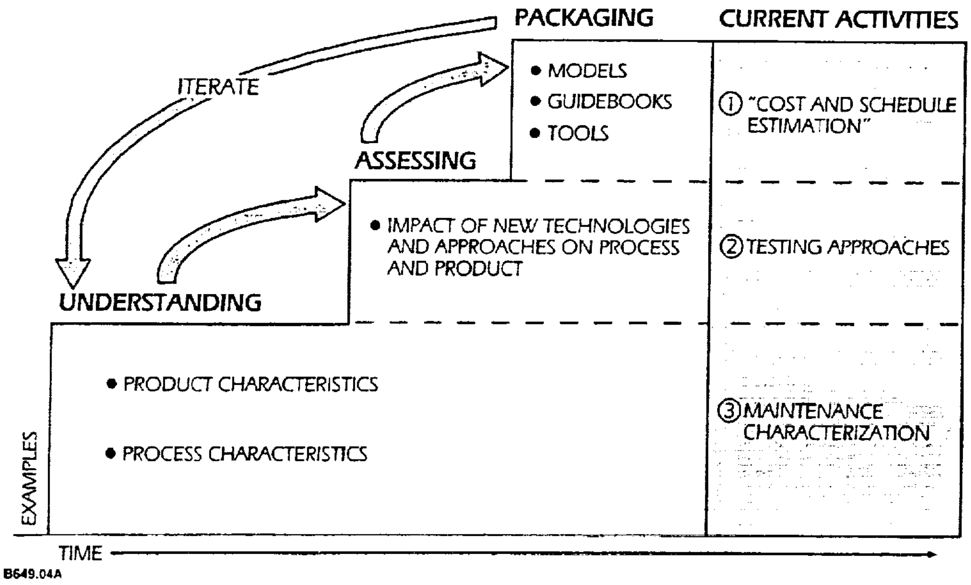
DEVELOPMENT EFFORT DISTRIBUTION



MAINTENANCE COST SIGNIFICANTLY DRIVEN BY SYSTEM TYPE

B649.015

SEL IMPROVEMENT -- AN ONGOING PROCESS



- MODELS
- GUIDEBOOKS
- TOOLS

- IMPACT OF NEW TECHNOLOGIES AND APPROACHES ON PROCESS AND PRODUCT

- PRODUCT CHARACTERISTICS
- PROCESS CHARACTERISTICS

① "COST AND SCHEDULE ESTIMATION"

② TESTING APPROACHES

③ MAINTENANCE CHARACTERIZATION

Session 2: Measurement

omit

Ross Jeffery, University of New South Wales

Martha Ann Griesel, Jet Propulsion Laboratory/
California Institute of Technology

Anneliese von Mayrhauser, Colorado State University

Specification-Based Software Sizing: An Empirical Investigation of Function Metrics

Ross Jeffery & John Stathis
School of Information Systems
University of New South Wales
P.O. Box 1, Kensington, NSW, 2033
AUSTRALIA

1. Introduction:

For some time the software industry has espoused the need for improved specification-based software size metrics (see Evanco et. al. 1992). During the 1980's significant resources have been applied to the development and use of metrics such as function points [Albrecht79], function weights [DeMarco82], feature points [Jones 1988] and other metrics. Earlier research [Jeffery&Low93] has established the similarity of these metrics. These metrics are used as one of the bases for cost estimation, software development management, software maintenance management, software value measurement, and so on. The proliferation of the use of the metrics and the tools now developed to support the measurement process to provide these measures, suggests that they fill an established need within the software industry. However the empirical research into these metrics has been sparse and generally not particularly favourable. Once again we see industry seeking problem solutions in the absence of experimental findings which support the solutions on offer.

This paper reports on a study of nineteen recently developed systems in a variety of application domains. The systems were developed by a single software services corporation using a variety of languages. The study investigated the following metric characteristics and questions:

Using both early and late lifecycle system documents as input to the counting process, what variation occurs in counts produced for the same system, and what gives rise to that variation? The research methodology adopted was to perform multiple independent counts of the system function size for the systems using the IFPUG Standard version 3.4. For each system this resulted in two measured function counts. The difference between these counts was analyzed both for its magnitude and the reasons for the variation. The internal validity of the function point metric was also studied and the appropriateness of the metric to the application portfolio of the organization.

This paper presents the results of this study. It is shown that:

1. Earlier research [Kitchenham 93] into inter-item correlation within the overall function count is partially supported
2. A priori function counts, in themself, do not explain the majority of the effort variation in software development in the organization studied.
3. Documentation quality is critical to accurate function identification
4. Rater error is substantial in manual function counting.

The implication of these findings for organizations using function based metrics are explored.

2. The Data Set:

The source of data for this project was an Australian software development organisation, MEGATEC Proprietary Limited, a company with approximately 50 employees that develop and distribute a range of computer software products in Australia and overseas. This organisation was selected as a test site for this work because it was one of the first software companies in Australia to gain certification to Australian Standard AS3563 for Software Quality Management. The commitment to quality in this organisation meant that managers were highly motivated to provide good quality data and there was a well established research ethic within the organisation. The 19 projects in the data set are drawn from a variety of applications. In total 17 recently completed projects were eventually included in the project database as two of the nineteen projects were not completed at the time of data analysis. A summary of the data is given in Table 1. The projects were developed during the period August 1990 to May 1993 and a high consistency in the quality staff in the use of methodology was expected in the database. The systems were written in a variety of languages including COBOL, Powerhouse, C and MS Windows, Excel Macro, SQL windows and combinations of these. It was decided that for the initial study tests would be carried out using the Albrecht Function Point counting technique as embodied in the International Function Point User Group standards as the basis for research.

TABLE I
PROJECT SIZE AND DEVELOPMENT EFFORT DATA

	Project Size (UFP)			Development Effort (Hours)		
No. Projects	Mean	Std Dev	Range	Mean	Std Dev	Range
17	551	923	38 - 3656	2093	3266	262 - 13905

Function Point were counted from documentation provided by the corporation. Each system was counted by two independent raters with experience in the IFPUG standard. One of the counters was an external consultant and the other was one of the researchers in the current study. Where we are studying the relationship between FP and other project phenomena we use the mean FP value. Data was available to derive the unadjusted function point count and also the fourteen complexity factors. In order to validate the data, structured interviews were held with all of the project managers. These interviews were used to validate the function point count, the effort data and to search for any reason behind abnormal results. There were three basic research questions which were being explored.

Firstly, we were interested in exploring in this organisational setting the relationship between development effort and function points. This question has had some considerable research over recent years, generally showing a consistent and significant relationship between the size measure and effort.

The second research question concerned replicating some of the work carried out by Kitchenham and Kansala (1993) concerning the relationships between constituent elements of the function point metric.

Thirdly, we were concerned with investigating the consistency of function point counting. There had been no study in which multiple systems were counted by multiple raters and yet it seemed that this is one of the critical elements given the current manual basis of function counting.

3. Results:

3.1 Effort Relationships

An initial Kolmogorov-Smirnov test indicated that the unadjusted and unweighted function count(UUFC), as well as the unadjusted function point (UFP) and effort data belonged to normal distributions. The results are shown in Table 2. That allowed us to proceed with a range of parametric statistical tests.

TABLE 2
KOLMOGOROV-SMIRNOV TEST

No of Projects	UUFC	UFP	Effort
	p	p	p
17	0.012	0.015	0.05

Figure 1 shows an initial plot of project size against effort for the full data set. The Project Sizing Figure 1 was unadjusted function points counted from early life-cycle documentation of a systems requirements. In this plot we can see that two of the projects are significant outliers in terms of effort and the other in terms of project size. We also note the scatter of points which has been typical in prior data when comparing size against function points. The R^2 for this data set is relatively poor showing a value of 0.228 ($p \leq 0.05$) for a linear regression of size against function points.

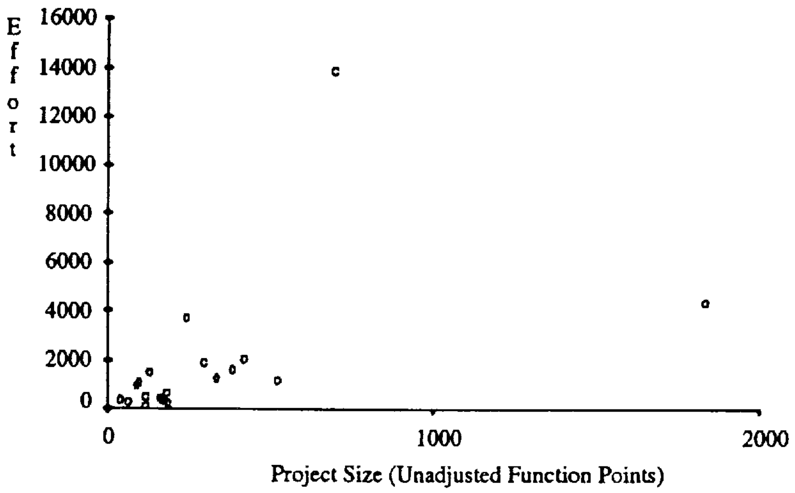


Figure 1. Scatter plot of A priori UFP against Effort

In the project manager interviews it became apparent that for some of the measured systems in the database, the project data which we show in Figure 1 was not a fair representation of the systems implemented. Taking this into account, the function point count and effort count was carried out again in order to correct any identified errors in the effort recorded or in the function point count. For example, it was found that for some of these systems the functionality had changed significantly during development and that it would be expected that a better relationship between size and effort would be found using the implemented function point count. Figure 2 shows a scatter plot for the seventeen data points after the validation of the data. The R^2 for this data set was 0.95 ($p < 0.001$). It is interesting to note the enormous difference between the data set derived at systems requirements specification stage versus the data set at implementation. This suggests that in this corporation considerable work will need to be invested to ensure requirements stability in the future if they are to gain control over predicted effort distribution.

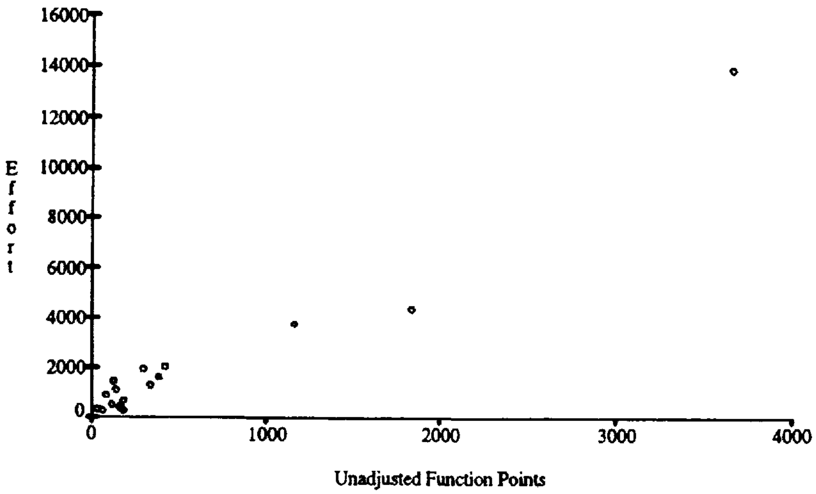


Figure 2. Scatter plot of A posteriori UFP against effort

Further analysis of the data revealed that three of the projects could be considered outliers and in line with conservative statistical analysis. Table 3 shows the regression results for the complete and the reduced data set where the outliers have been removed. Notice the reduction in the R^2 and that the effort-size relationship as expressed through the regression equation has not changed significantly suggesting that the outliers were in fact normal for this organisation.

TABLE 3
COMPARISON OF REDUCED AND FULL DATASET

	Full Dataset	Reduced Dataset
No. Projects	17	14
Equation	Effort = 192.31 + 3.45 * UFP	Effort = 187 + 4.03 * UFP
R^2	0.95 ($p<0.001$)	0.58 ($p<0.01$)

3.2 Internal Consistency of Function Points:

Table 3 shows the Pearson correlation coefficient between all pairs of function point elements using the reduced data set for conservatism. The results shows that three of the five function elements are significantly correlated. These are external inputs, external enquires and logical internal files.

TABLE 4
PEARSON CORRELATION COEFFICIENTS BETWEEN UFP ELEMENTS

Fn Point Element	**Total Unadjusted Function Point**	**EI**	**EO**	**Ext Inquiry**	**Extnl Int File**
External Input	**0.90** **($p<0.001$)**				
External Output	**0.14** **(n.s.)**	**-0.07** **(n.s.)**			
External Inquiry	**0.93** **($p<0.001$)**	**0.91** **($p<0.001$)**	**-0.17** **(n.s.)**		
External Interface File	**-0.33** **(n.s.)**	**-0.46** **(n.s.)**	**0.22** **(n.s.)**	**-0.45** **(n.s.)**	
Logical Internal File	**0.92** **($p<0.001$)**	**0.74** **($p<0.01$)**	**-0.06** **(n.s.)**	**0.90** **($p<0.001$)**	**-0.33** **(n.s.)**

Kitchenham and Kansala's study used Kendall's *t* as a robust measure of correlation. In their study they found significant correlations between three pairs of function elements not reported as significant in our study. These were outputs and inputs, outputs and enquiries and outputs and internal logical files.

The results of both of these studies shows that the function elements are not independent and therefore it is possible that there may be a better effort relationship between constituent elements an effort than there is between function points. The Pearson correlation between each function point element and actual development produced the results in Table 6. These show that internal logical files and external enquiries had a higher correlation with effort than the total unadjusted function point count. This suggests that an effort estimation model derived on the internal logical file count may in fact perform better than function point for this organization.

TABLE 6
PEARSON CORRELATION RESULTS
FUNCTION ELEMENTS AGAINST EFFORT

Function Element	R^2	p
Logical Internal File	0.73	< 0.001
External Inquiry	0.63	< 0.001
External Input	0.37	< 0.001
External Output	0.03	n.s.
External Interface File	0.005	n.s.
Sum of Function Elements (UFP)	0.58	< 0.01

These results are somewhat different to Kitchenham and Kansala who found that a combination of external inputs and outputs provided a better effort predictor than unadjusted function points.

A further analysis was carried out was to compare the extent to which the complexity adjustments in the function point model add to the value of the model in explaining effort. Table 7 shows the regression results for the unadjusted and unweighted function count versus the unadjusted function point count. It can be seen from this table that once again the function point metric as a measure of size when used in its relationship with effort, appears to be performing less well than some of the constituent elements of that count.

TABLE 7
PEARSON CORRELATION RESULTS
FUNCTION ELEMENTS (UUFC & UFP) AGAINST EFFORT

	Level 1	Level 2		
	UUFC	UFP		
Function Element	**R^2**	**p**	**R^2**	**p**
Logical Internal File	**0.75**	**< 0.001**	**0.73**	**< 0.001**
External Inquiry	**0.65**	**< 0.001**	**0.63**	**< 0.001**
External Input	**0.37**	**< 0.001**	**0.37**	**< 0.001**
External Output	**0.04**	**n.s.**	**0.03**	**n.s.**
External Interface File	**0.002**	**n.s.**	**0.005**	**n.s.**
Sum of Function Elements	**0.56**	**< 0.01**	**0.58**	**< 0.01**

3.3 Rater Consistency:

The model used in this study to investigate rater consistency is shown in Figure 3 in which we see that three elements which can contribute to inconsistency. These have been identified as the system specification, the function point counting method and the rater. For example, inconsistency can be derived from the fact that the raters themselves may simply introduce errors into the function point process. It can also be that the specification can be ambiguous or at an inappropriate level of granularity such that the function point is difficult to determine, or else it could be that the function point method could be ambiguous or incomplete with respect to the function counting process that is at hand.

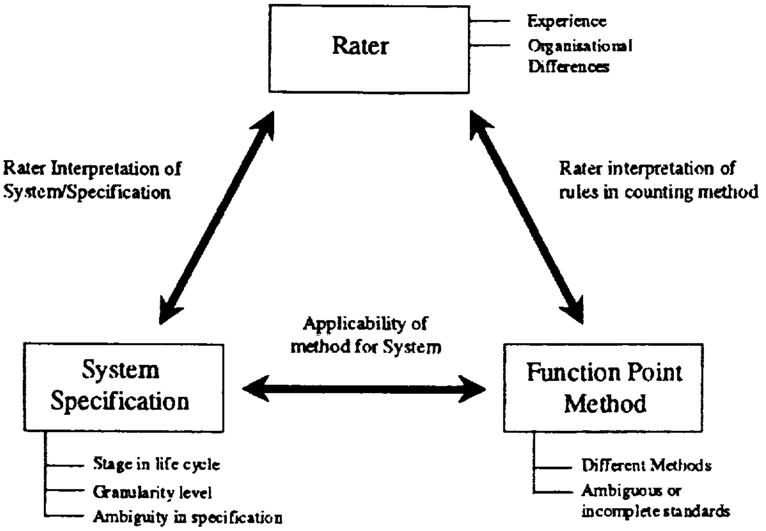

Rater
Experience
Organisational Differences
Rater Interpretation of System/Specification
Rater interpretation of rules in counting method
Applicability of method for System
System Specification
Function Point Method
Stage in life cycle
Granularity level
Ambiguity in specification
Different Methods
Ambiguous or incomplete standards

Figure 3 - A Model of the Factors Affecting Function Point Reliability

In our research we had two raters count the same systems and used variations on absolute relative difference between counts as the measure for analysis. We define the magnitude of the difference in counts between rater A and rater B as shown in equation 1 where the absolute relative is a normalised difference between the two raters normalised by average system size. We further refined this metric to the weighted absolute relative difference WARD, where we separate out the effect of each of the internal components of the function count so that errors in inputs for example, are not washed away by errors in outputs which happens if they move in opposite directions.

$$\mathrm{ARD}_{\mathrm{UFP\,(Rater\ A;\ Rater\ B)}} = \frac{|\mathrm{Rater\ A}_{\mathrm{UFP}} - \mathrm{Rater\ B}_{\mathrm{UFP}}|}{(\mathrm{Rater\ A}_{\mathrm{UFP}} + \mathrm{Rater\ B}_{\mathrm{UFP}})\ /\ 2}$$

$$\mathrm{WARD}_{\mathrm{(EI.EO.INQ.LIF.EI;\ Rater\ A,\ Rater\ B))}} = \mathrm{ARD}_{\mathrm{EI}} \times \frac{\overline{\mathrm{EI}}_{\mathrm{(Rater\ A,\ Rater\ B)}}}{\overline{\overline{\mathrm{UFP}}}_{\mathrm{(Rater\ A,\ Rater\ B)}}} + \ldots + \mathrm{ARD}_{\mathrm{EIF}} \times \frac{\overline{\mathrm{EIF}}_{\mathrm{(Rater\ A,\ Rater\ B)}}}{\overline{\overline{\mathrm{UFP}}}_{\mathrm{(Rater\ A,\ Rater\ B)}}}$$

Table 8 shows the analysis results for this and in this we see that the mean WARD for these two raters is 55%. This suggests that the counting practice is relatively unstable when looked from this perspective.

Project Number	Rater A UFP	Rater B UFP	ARD	WARD	Effort	Hours Per Function Point (A, B)
Mean	**302.8**	**337.1**	**0.31**	**0.55**	**1947**	**(7.50, 6.52)**

Further analysis of this data revealed that 68% of the variation between the two counters could be attributed to rater interpretation of the specification or the application of the counting standard to that specification. Some 32% of the difference could be attributed to a simple error on the part of the rater.

4. Conclusions:

The following can be concluded from this study:

1. In a pragmatic sense the relationship between a posteriori function points and a posteriori effort is very strong for this organisation with an R^2 of .95 for the full data set or .58 for the reduced data set. This suggests that function points could be used effectively as a basis for software management in this organization.

2. From a scientific perspective it appears clear that the function point metric has some significant limitations. There is reason for concern about the function point metric. The structure of the metric is such that the components are not orthogonal which introduces issues concerning the structure of the metric. It is also of concern that the addition of the function component complexity ratings does not add to the effort relationship or the power of the effort explanation of the model. As this is counter-intuitive it warrants further investigation.

3. Inconsistency which has been observed in this study between the raters' function point counts (58%) and the high component of that difference (68%) which can be ascribed to either the function points standard or the requirements specification, suggests that the function point counting or at least the base function counting needs to be automated.

4. Given the results concerning the strong relationships between the number of internal logical files or data entities and effort, may well be possible that given further research, that if a consistent relationship holds between data entities and effort than automated size counting from data models may well be a fruitful area for further investigation.

5. References:

Albrecht,A.J. "Measuring Application Development Productivity", Share/Guide Application Development Symposium, Oct, 1979.83-92.

DeMarco, T. (1982) *Controlling Software Projects: Management, Measurement, and Estimation*, Yourdon Press, New York.

Evanco, W.M., Thomas, W.M. & Agresti W.W. "Estimating Ada System Size During Development", *Rome Laboratory Technical Report*, RL-TR-92-318, New York, December,1992.

Jeffery,D.R Low, G.& Barnes,M. "A Comparison of Function Point Counting Techniques", *IEEE Trans. on S'ware Eng.*, May 1993.

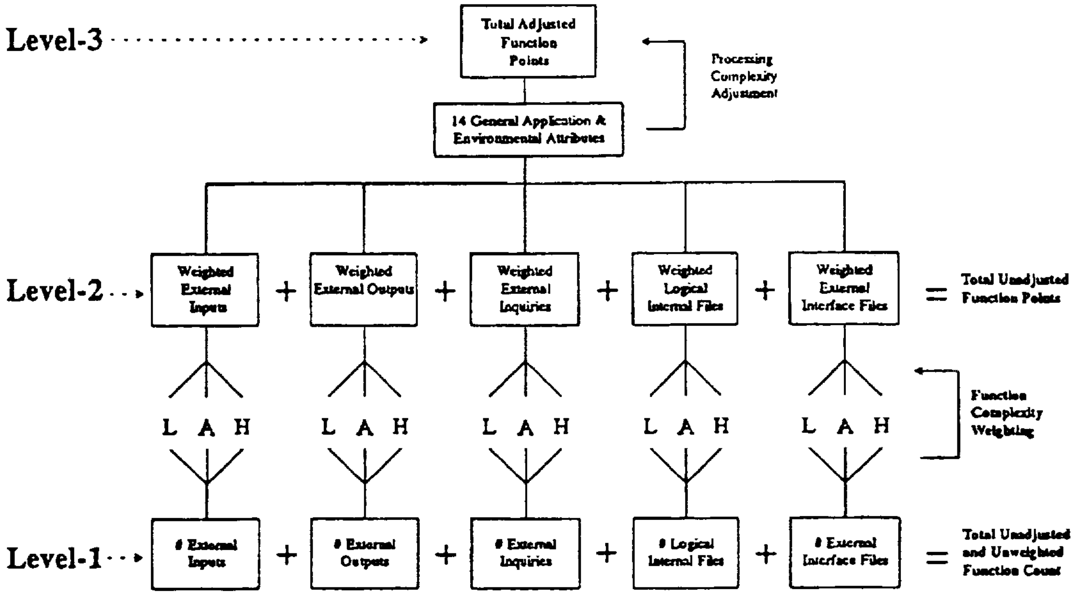
Jones,T.C. "A Short History of Function Points and Feature Points", Software Productivity Research, 1988.

Kitchenham, B & Kansala, K. "Inter Item Correlations Among Function Points", *Proc First International Software Metrics Symposium*, IEEE computer Society, Baltimore, May, 1993. 11-15.

Specification Based Software Sizing:
An Empirical Investigation of Function Metrics

Ross Jeffery & John Stathis
University of New South Wales
P.OBox 1, Kensington, NSW 2033
Australia

NASA SEL Workshop 1993



Level-3

Total Adjusted Function Points

Processing Complexity Adjustment

14 General Application & Environmental Attributes

Level-2

Weighted External Inputs + Weighted External Outputs + Weighted External Inquiries + Weighted Logical Internal Files + Weighted External Interface Files = Total Unadjusted Function Points

L A H

L A H

L A H

L A H

L A H

Function Complexity Weighting

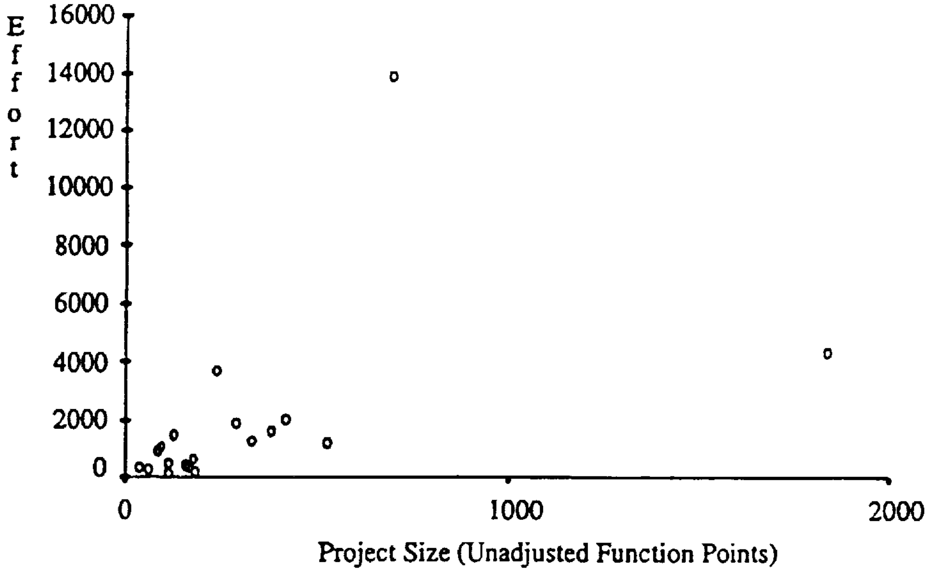
Level-1

External Inputs + # External Outputs + # External Inquiries + # Logical Internal Files + # External Interface Files = Total Unadjusted and Unweighted Function Count

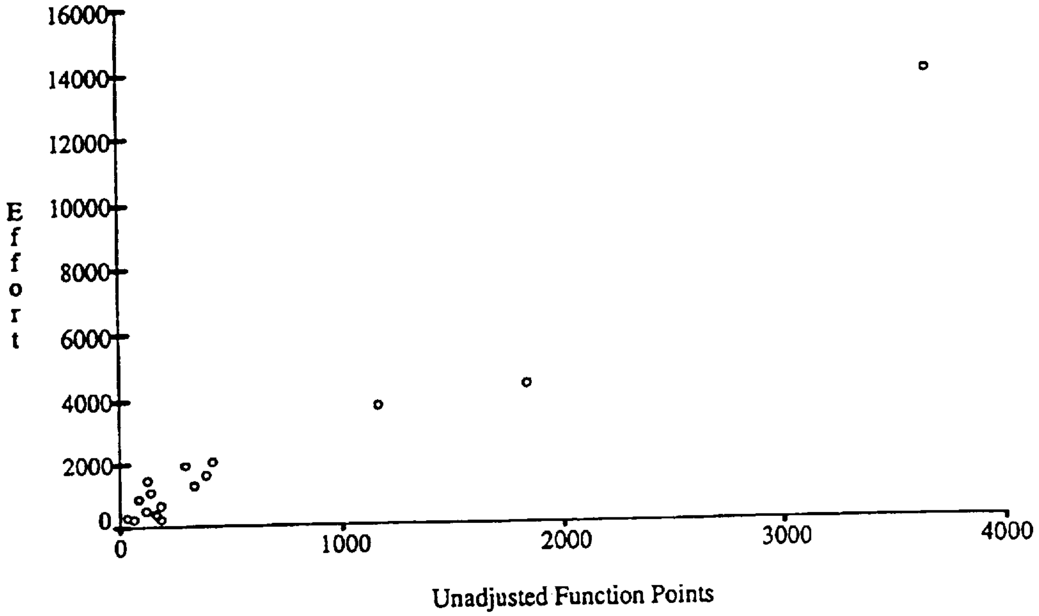
TABLE I
PROJECT SIZE AND DEVELOPMENT EFFORT DATA

	Project Size (UFP)			Development Effort (Hours)		
No. of Projects	Mean	Std. Dev.	Range	Mean	Std. Dev.	Range
17	551	923	38 - 3656	2093	3266	262 - 13905

3 *Ross Jeffery NASA SEL Workshop 1993*



4 *Ross Jeffery NASA SEL Workshop 1993*



5 *Ross Jeffery NASA SEL Workshop 1993*

TABLE II
COMPARISON OF PRE AND POST IMPLEMENTATION DATASET

	Pre Implementation FP	Post Implementation FP
No. Projects	17	17
Regression Equation (UFP against effort)	effort = 914.6 + 3.7 * UFP	
R^2 (p)	0.228 (0.05)	0.95

6 *Ross Jeffery NASA SEL Workshop 1993*

TABLE III
COMPARISON OF REDUCED AND FULL DATASET

	Full Dataset	Reduced Dataset
No. Projects	17	14
Regression Equation (UFP against effort)	effort = 192.31 + 3.45 * UFP	effort = 185.37 + 4.03 * UFP
R^2 (p)	0.95 ($p < 0.001$)	0.58 ($p < 0.01$)

TABLE IV
PREVIOUS STUDIES - UFP AGAINST EFFORT

Study	No. of Projects	Unadjusted Function Points	
		R^2	(p)
Albrecht and Gaffney, 1983	24	0.90	< 0.001
Kemerer, 1987	15	0.54	< 0.001
Kitchenham and Kansala, 1993	40	0.41	< 0.01
Jeffery et. al., 1993	64	0.36	< 0.001
Jeffery & Stathis, Current Study	14	0.58	< 0.001

TABLE V
PEARSON CORRELATION COEFFICIENTS BETWEEN UFP ELEMENTS

Function Point Element	Total Unadjusted Function Point	External Input	External Output	External Inquiry	External Interface File
External Input	0.90 ($p<0.001$)				
External Output	0.14 (n.s.)	-0.07 (n.s.)			
External Inquiry	0.93 ($p<0.001$)	0.91 ($p<0.001$)	-0.17 (n.s.)		
External Interface File	-0.33 (n.s.)	-0.46 (n.s.)	0.22 (n.s.)	-0.45 (n.s.)	
Logical Internal File	0.92 ($p<0.001$)	0.74 ($p<0.01$)	-0.06 (n.s.)	0.90 ($p<0.001$)	-0.33 (n.s.)

TABLE VI
PEARSON CORRELATION RESULTS
FUNCTION ELEMENTS AGAINST EFFORT

Function Element	R^2	p
Logical Internal File	0.73	< 0.001
External Inquiry	0.63	< 0.001
External Input	0.37	< 0.001
External Output	0.03	n.s.
External Interface File	0.005	n.s.
Sum of Function Elements (UFP)	0.58	< 0.01

TABLE VII
PEARSON CORRELATION RESULTS
FUNCTION ELEMENTS (UUFC & UFP) AGAINST EFFORT

	Level 1		Level 2	
	UUFC		UFP	
Function Element	R^2	p	R^2	p
Logical Internal File	0.75	< 0.001	0.73	< 0.001
External Inquiry	0.65	< 0.001	0.63	< 0.001
External Input	0.37	< 0.001	0.37	< 0.001
External Output	0.04	n.s.	0.03	n.s.
External Interface File	0.002	n.s.	0.005	n.s.
Sum of Function Elements	0.56	< 0.01	0.58	< 0.01

11 *Ross Jeffery NASA SEL Workshop 1993*

TABLE VIII
EFFORT ESTIMATE ARE t-TESTS FOR
UUFC AND UFP

	Unweighted and Unadjusted Function Count (UUFC)		Unadjusted Function Point (UFP)			
No. of Projects	Mean ARE	Std. Dev.	Mean ARE	Std. Dev.	*t*	*p*
17	0.53	0.64	0.51	0.60	0.70	0.492

12 *Ross Jeffery NASA SEL Workshop 1993*

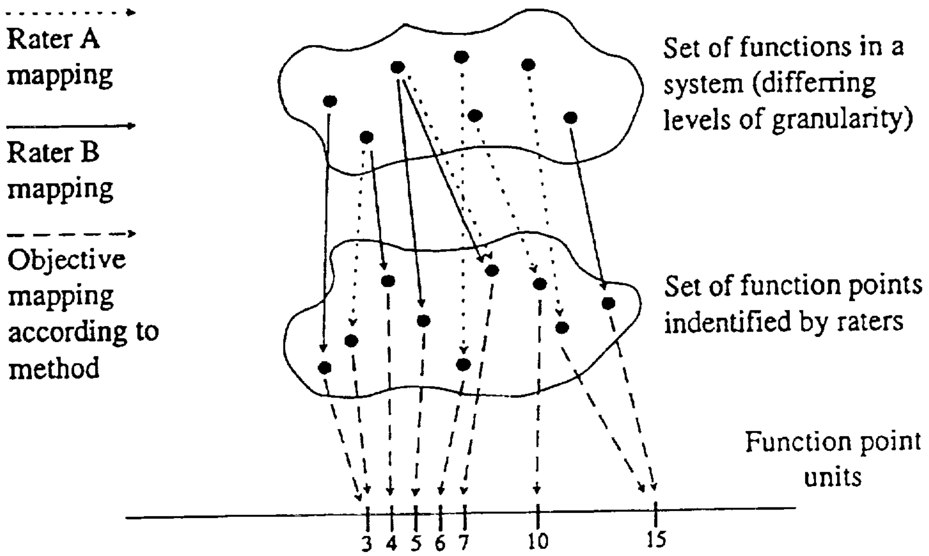

Rater A mapping
Rater B mapping
Objective mapping according to method
Set of functions in a system (differring levels of granularity)
Set of function points indentified by raters
Function point units
3
4
5
6
7
10
15

Figure II - Mapping a Set of Functions to Function Point Units

13 *Ross Jeffery NASA SEL Workshop 1993*

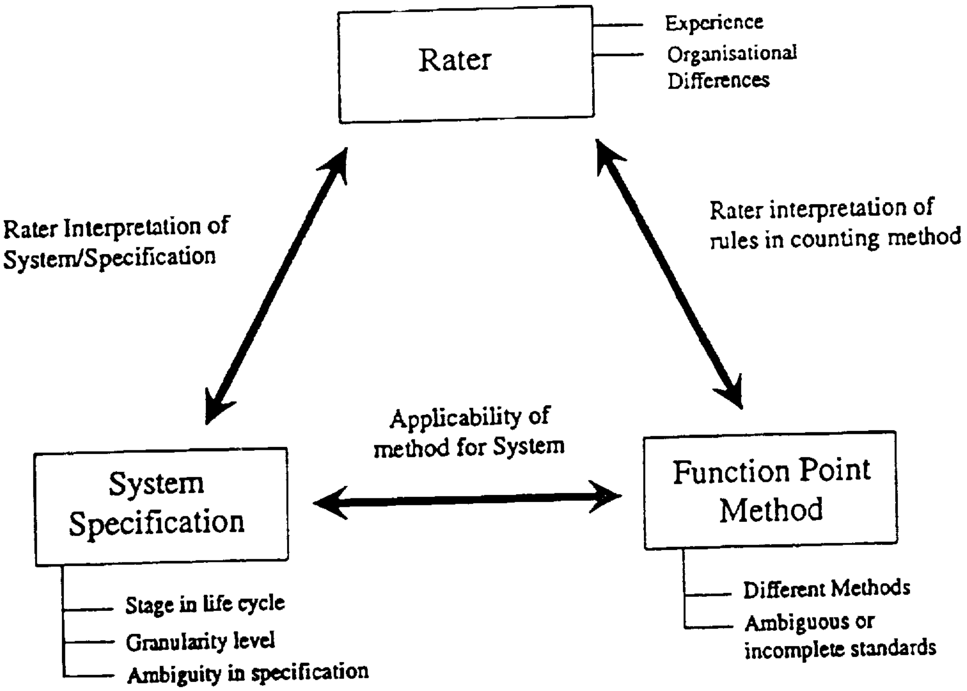
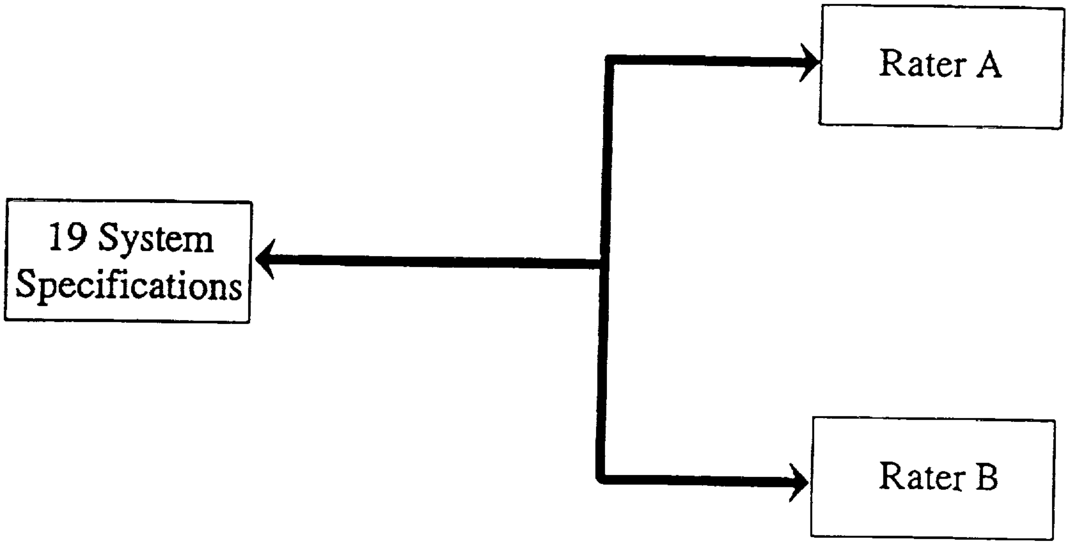

Rater
Experience
Organisational Differences
Rater Interpretation of System/Specification
Rater interpretation of rules in counting method
Applicability of method for System
System Specification
Function Point Method
Stage in life cycle
Granularity level
Ambiguity in specification
Different Methods
Ambiguous or incomplete standards

Figure III - A Model of the Factors Affecting Function Point Reliability

14 *Ross Jeffery NASA SEL Workshop 1993*



19 System Specifications

Rater A

Rater B

Figure VIII - Research Design for Current Study

15 *Ross Jeffery NASA SEL Workshop 1993*

$$\mathrm{ARD}_{\mathrm{UFP\ (Rater\ A;\ Rater\ B)}} = \frac{|\text{Rater } A_{\mathrm{UFP}} - \text{Rater } B_{\mathrm{UFP}}|}{(\text{Rater } A_{\mathrm{UFP}} + \text{Rater } B_{\mathrm{UFP}}) \ / \ 2}$$

$$= \ 31\%$$

$$\mathrm{WARD}_{\mathrm{(EI.EO.INQ.LIF.EIF;\ Rater\ A,\ Rater\ B))}}$$

$$= \mathrm{ARD}_{\mathrm{EI}} \times \frac{\overline{\mathrm{EI}}_{\mathrm{(Rater\ A,\ Rater\ B)}}}{\overline{\mathrm{UFP}}_{\mathrm{(Rater\ A,\ Rater\ B)}}} + \ldots$$

$$+ \ \mathrm{ARD}_{\mathrm{EIF}} \times \frac{\overline{\mathrm{EIF}}_{\mathrm{(Rater\ A,\ Rater\ B)}}}{\overline{\mathrm{UFP}}_{\mathrm{(Rater\ A,\ Rater\ B)}}}$$

$$= 55\%$$

16 *Ross Jeffery NASA SEL Workshop 1993*

MEAN ABSOLUTE RELATIVE DIFFERENCE (MARD) UNWEIGHTED AND WEIGHTED FUNCTION POINTS

	Total Function Point Count (UUFC) (UFP)	External Input	External Output	External Inquiry	External Interface File	Logical Internal File
Unweighted Function Points	0.33	0.76	0.69	0.65	0.54	0.45
Weighted Function Points	0.31	0.67	0.70	0.62	0.54	0.43

17 *Ross Jeffery NASA SEL Workshop 1993*

1. Strong a posteriori function points and a posteriori effort relationship for this organisation - R^2 of 0.95 for the full data set or 0.58 for the reduced data set.

2. The function point metric has some significant limitations.

   Components are not orthogonal

   Function component complexity ratings does not add to the effort explanation of the model.

3. Inconsistency has been observed between the raters' function point counts (58%)

   A high component of that difference (68%) can be ascribed to either the function points standard or the requirements specification

4. Automated size counting from data models may well be a fruitful area for further investigation.

18 *Ross Jeffery NASA SEL Workshop 1993*

Software Forecasting As It Is Really Done: A Study of JPL Software Engineers

Martha Ann Griesel
Jairus M. Hihn
Kristin J. Bruno
Thomas J. Fouser
Robert C. Tausworthe

Jet Propulsion Laboratory/California Institute of Technology
4800 Oak Grove Avenue
Pasadena, Ca. 91109

Abstract

This paper presents a summary of the results to date of a Jet Propulsion Laboratory internally funded research task to study the costing process and parameters used by internally recognized software cost estimating experts. Protocol Analysis and Markov process modeling were used to capture software engineer's forecasting mental models. While there is significant variation between the mental models that were studied, it was nevertheless possible to identify a core set of cost forecasting activities, and it was also found that the mental models cluster around three forecasting techniques. Further partitioning of the mental models revealed clustering of activities, that is very suggestive of a forecasting lifecyle. The different forecasting methods identified were based on the use of multiple-decomposition steps or multiple forecasting steps. The multiple forecasting steps involved either forecasting software size or an additional effort forecast. Virtually no subject used risk reduction steps in combination. The results of the analysis include: the identification of a core set of well defined costing activities, a proposed software forecasting life cycle, and the identification of several basic software forecasting mental models. The paper concludes with a discussion of the implications of the results for current individual and institutional practices.

1.0 Introduction

In today's cost constrained environment, cost estimation is becoming an integral part of the engineer's job. Therefore, tools and databases are needed that are consistent with engineering based costing methods. Previous surveys have shown that engineers in general do not use tools and databases, finding them inconsistent with their intuitive engineering-based costing methods, in particular analogy-related techniques. (Hihn and Habib-agahi, 1991) This lack of correspondence between software forecasting practices and available computer-based tools prompted the current research.

To be able to design and develop tools and databases that are more consistent with engineering-based costing methods requires that there exist a relatively small number of costing activities and that these activities are primarily used in a few well defined sequences. A sequence of activities is what makes up a costing method or, in cognitive psychology terminology, the cost forecaster's mental model. The existence of a small number of basic forecasting mental models requires that the mental models depend on high-level domain and environment conditions, rather than personal style and low-level domain details.

To the best of the authors' knowledge, there have been only three attempts to develop such mental models of the forecasting process that are documented in the literature: Vicinanza et. al. (1991), Howard (1992), and Hihn et. al. (1993).

Vicinanza et. al. completed an exploratory study of the methods used by experts. In Vicinanza et. al. five respondents who ranked a series of cost drivers and then estimated the development effort that would be required for 10 projects. The forecasters' methods were categorized into four groups: algorithmic initial condition, algorithmic effort estimate, analogical initial condition, and analogical effort estimate. For a method to be algorithmic the forecaster had to mention and use productivity figures. For a method to be analogical the forecaster had to mention a reference project. Four of the estimators used an algorithmic approach and only one used analogy. Vicinanza et. al. propose a logic flow (mental model) for algorithmic and analogical forecasting (see Figure 1 for the analogy model). Given their simple categorization scheme it is unclear how they derived their mental model. Also, the experimental design required that the engineers use COCOMO cost drivers (Boehm, 1981) and function point descriptors (Albrecht and Gaffney, 1983), neither of which may have been natural to them; and the terms used in the proposed mental models are neither goals nor the vocabulary that are commonly used by software engineers.

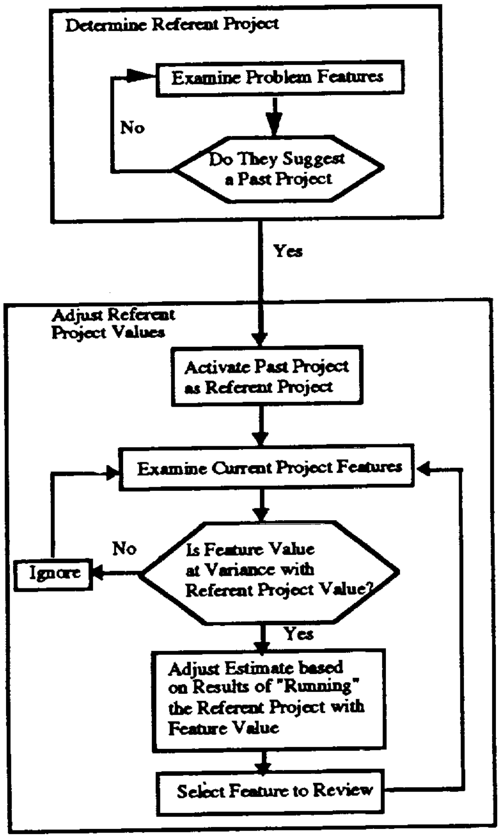

Determine Referent Project
Examine Problem Features
No
Do They Suggest a Past Project
Yes
Adjust Referent Project Values
Activate Past Project as Referent Project
Examine Current Project Features
No
Ignore
Is Feature Value at Variance with Referent Project Value?
Yes
Adjust Estimate based on Results of "Running" the Referent Project with Feature Value
Select Feature to Review

Figure 1 : Abstraction of Analogical-Estimation Strategy from Vicinanza et. al. (1991)

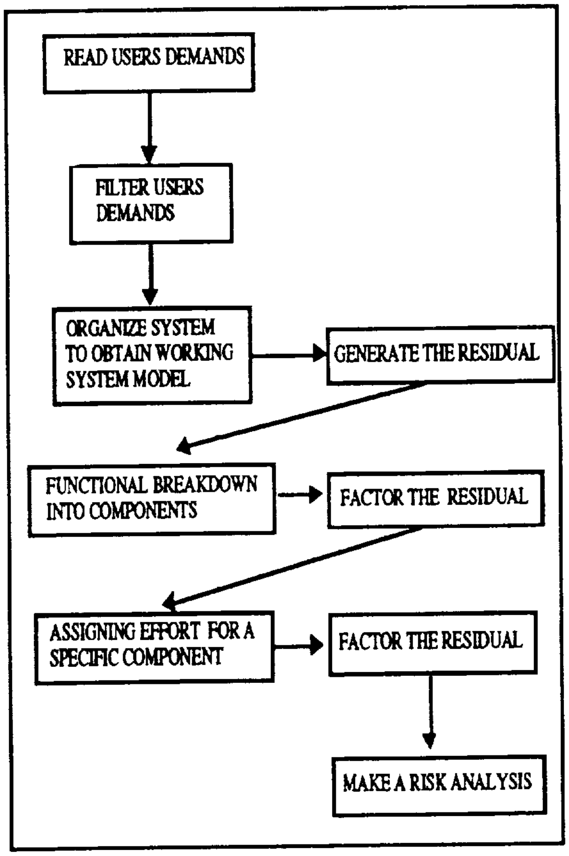

READ USERS DEMANDS
FILTER USERS DEMANDS
ORGANIZE SYSTEM TO OBTAIN WORKING SYSTEM MODEL
GENERATE THE RESIDUAL
FUNCTIONAL BREAKDOWN INTO COMPONENTS
FACTOR THE RESIDUAL
ASSIGNING EFFORT FOR A SPECIFIC COMPONENT
FACTOR THE RESIDUAL
MAKE A RISK ANALYSIS

Figure 2 : A Bottom Up Approach to Estimation from Howard (1992)

Howard (1992) reports the results of two surveys on software cost estimation practices for standard information systems such as a banking transaction system. Approximately 50 observations were collected using a survey form. Twelve observations were collected using semi-structured face-to-face, interviews based on a case description, given to the subjects before the interview. The main objective of the research is to study how cost estimates are developed in group settings. The objective of the reported portion of the research task was to develop a mental model of the processing steps that estimators follow that could be used to support the study of group cost forecasting. A very high level model with about 20 possible steps based upon cognitive processing theories was proposed. Figure 2 illustrates the mental model of individuals applying the "bottom up" process. Interestingly, aggregation was never mentioned, even though "functional breakdown into components" is explicitly shown. The model proposed is intuitively appealing. However, the respondents provided quite generic responses in describing how they normally do cost estimating. Howard reports this is because the case example was found to be too poorly defined. Verbal reports of this type are well known to lead to biased, and very likely, inconsistent results [see Ericson and Simon, (1984)].

In both of the papers described above, the bases by which the proposed software forecasting mental models were derived is not explained. Howard followed some basic cognitive psychology techniques, but it was not clear that they were derived by a repeatable analysis. A significant problem, from the perspective of identifying a more detailed picture of the underlying mental model, was that most of what distinguishes an expert from a novice is in how they generate and "factor residuals" or, in other words, incorporate their cost drivers and adjustment factors.

Hihn et. al.(1993) attempt to address these problems by using a more precise data capture and analysis technique. In Hihn et. al.(1993) a combination of Protocol Analysis Ericson and Simon (1984) and Markov process modeling Papoulis (1991) is shown to be a viable technique for capturing the engineers' cost forecasting mental models in a repeatable manner. With this technique, Protocol Analysis was used to extract a common forecasting vocabulary across engineers and application domains by translating the engineers' self reports into verbal protocols, and Markov analysis was used to identify the common transitions, or steps, in the engineers' mental models. Seven primary cost forecasting activities were identified that clustered into 6 different, but not mutually exclusive, sequences (mental models) using this analysis technique. The 7 activities that were identified are requirements identification, attribute identification, attribute application, decomposition, estimation, aggregation, and adjustments. The definition of these terms are reviewed in Section 3.0. The original clusters of sequences were derived based upon purely data descriptive criteria. For example, a sequence that contains a single decomposition and single estimation activity is in a different sequence cluster then a sequence with multiple decomposition and multiple estimation activities. A very simplified example of the type of mental model this approach produces is displayed in Figure 3.

In this paper we are reporting an extension of these results that incorporates an increased number of cost forecasting activities and the identification of activity sequences (mental models) that correspond to software domain and development environment criteria. In addition, as part of identifying a number of basic mental models, it was possible to derive the components of a software cost forecasting life cycle based upon actual costing behavior.

2.0 Sample Definition and Institutional Background Information

Jet Propulsion Laboratory (JPL) is a Federally Funded Research and Development Center run by the California Institute of Technology under a government contract with National Aeronautics and Space Administration. As a national laboratory, it performs research and development activities in the national interest, primarily the development of robotic spacecraft for interplanetary studies. In addition, a portion of JPL's budget is supplied by non-NASA organizations such as the Department of Defense.

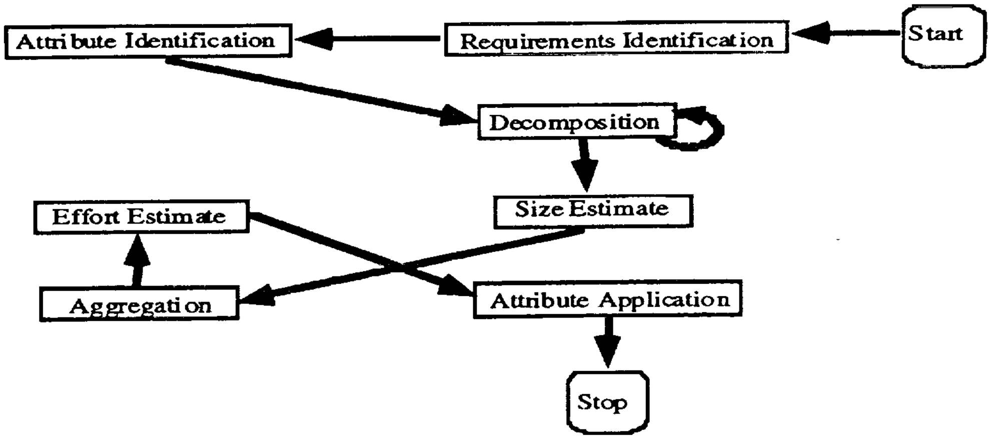


Figure 3 : Example of a CER-Based Forecasting Mental Model

A survey was conducted of the technical staff that had experience forecasting software development costs during the summer and fall of 1989. Over 185 software engineers were contacted for participation in the original survey. Of the 185 contacted, over 100 were identified who estimate effort, size and/or cost for software tasks. Of these, 83 were willing to complete a questionnaire on current software cost estimation practices. Of these, 28 responses provided sufficient information for use with the current analysis. For a detailed discussion of how the original data was collected see Hihn and Habib-agahi (1991).

The original purpose of the survey was to study the ability of software engineers to estimate effort and size given an architectural design document. In addition, the survey included a brief description of the typical approach each estimate used. The verbal protocols describing the cost forecasts used in the study were made during the system functional design and software requirements analysis phases (see Figure 4). Since data collected in this manner is not strictly appropriate for Protocol Analysis, conclusions drawn from this secondary analysis of the data may be questionable.

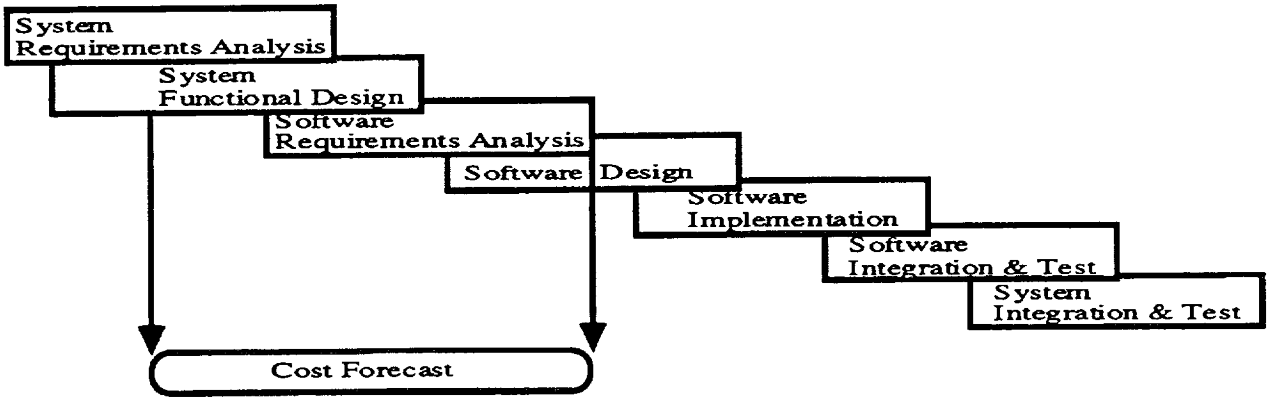

System Requirements Analysis
System Functional Design
Software Requirements Analysis
Software Design
Software Implementation
Software Integration & Test
System Integration & Test
Cost Forecast

Figure 4: Timing of Cost Forecasting Verbal Protocol Collection Relative to the Software System Development Lifecyle

Table 1: Hypothetical Software Cost Forecasting Activities

Activity	Definition
Requirements Identification	The obtaining or retrieval of information. Key vocabulary words are: read requirements, talk to experts, review requirements, and obtain requirements.
Attribute Identification	Attributes are key aspects of a task that are used in forming the system mental model and are also used as analogy discriminators and cost drivers. This is one of the main products of the analysis of the requirements. Attribute identification is generally described by the basic activity that was undertaken with the result that precise attributes are rarely specified at this point. These consist of both product and process attributes. Key vocabulary words are: identify, understand, analyze, and include.
Decomposition	The breaking down of a software entity (system, subsystem, etc.) into smaller and simpler pieces. The types of decomposition that have so far been identified are: functional, work breakdown structure (WBS), new vs old system components requirements. Key vocabulary words are: breakdown (functions), identify sub-tasks, develop WBS.
Estimation	The prediction of future cost and other key project management dimensions. Three types of forecasts were reported: size, effort, and cost. Estimation was further divided by type of technique used: analogical expert judgement explicit analogy algorithmic rules of thumb cost estimating relationships Key vocabulary words are: use (analogy, rule of thumb), estimate (SLOC, effort), and cost.
Attribute Application	The explicit use of the system attributes to discriminate between systems for purposes of analogical comparison or as cost drivers when using an algorithmic approach. Identification primarily depends upon specific mention of attribute. While there is less homogeneity in the vocabulary some common phrases are: adjust, use (fog factor), add (change, fog factor, etc.), multiply.
Aggregation	The combination of forecasted values associated with the system pieces produced by decomposition. Key vocabulary words are: add-up, and run SRM (JPL resource management tool)
Adjustments	Multipliers used independently of the system being estimated. Usually applied at a higher level then attributes. Consist of adjustments for purposes of risk, scaling, and bias(error). Key vocabulary word is: add percent.
Evaluation	Any activity performed as part of checking that a forecast meets certain criteria. Most often this is the comparison of effort or cost estimate and is the last activity completed. Can also be a design-to-cost activity. Key vocabulary word is: compare to (cost of last task, budget).

3.0 Cost Forecasting Activity Definitions

Table 1 contains a list of the software forecasting activities and sub-activities that were identified in the process of converting the verbal protocols into data. These activities constitute an abstract vocabulary that was used to describe the forecasting process. The activities and their definitions were derived from the literature, JPL experiences documented in Lessons Learned, and the personal costing experiences of the authors, then modified by the data available in the verbal protocols to maximize the scoring of the linguistic units into one and only one scoring category. The level of granularity of the activities determined the information obtainable from analysis of the forecasters' activity sequences. An activity set defined at too coarse a granularity can not distinguish between sequences and all protocols will appear identical. An activity set defined with to much detail, at too fine a granularity, makes every protocol appear unique. Hence identifying the right granularity, or level of abstraction, is crucial. For a detailed description of the mapping of the vocabulary used in the verbal protocols to these activities see Appendix A in Hihn et. al. (1993).

The activities that have been added or changed since the analysis documented in Hihn et. al. (1993) are Evaluation and a re-grouping of the estimation sub-activities. The Estimation activity has been disaggregated into Size Estimation, Effort Estimation and Cost (dollar) Estimation. A distinction has also been made between Formal and Informal Effort Estimation. Formal Effort Estimation corresponds to the use of a CER or an analogical reference to a specific task or cost or size database, and Informal Effort Estimation corresponds to the use of a rule-of-thumb or any form of expert (engineering) judgment. When Effort Estimation is referred to as part of a specific mental model, it always should be understood to mean Informal Effort Estimation. The addition of the Evaluation activity to the activity list is the most fundamental change because it is a completely new activity. The specific activities that are used for describing forecasters mental models in the current analysis are Requirements Identification, Attribute Identification, Attribute Application, Decomposition, WBS Decomposition, New/Old Decomposition, Size Estimation, Cost Estimation, Informal Effort Estimation, Formal Effort Estimation, Aggregation, Adjustments, and Evaluation.

4.0 Software Forecasting Activity Analysis

The cost forecasting activities were analyzed several different ways in order to discern if there were any well defined patterns in the data. The purpose in this part of the analysis was to see if the frequency of use of an activity could be explained by some aspect of the system, environment, or an overall method that was being used. The most significant relationship we found is displayed in Table 2. For additional analysis of the activities see Hihn et. al. (1992). Some activities such as requirements identification and attribute identification were used by all the engineers interviewed. Some activities were used infrequently, e.g. adjustments and evaluation. There were three activities that were found to define relatively distinct sub-populations and correlated with the type of system being developed. These were the use of New/Old decomposition, size estimation, and the execution of a second effort estimate, which we shall call an assessment [1]. The other category consisted of cases where no pattern of activity use could be discerned. If a protocol used both a size estimate and an assessment it was counted twice. As will be seen in Section 6, the occurrence of these activities drives the whole sequence of activities.

The different types of software systems identified were rapid prototyping, formal military, research and development (R&D), evolving ground systems, and flight software. At JPL, rapid protoyping is used primarily to support military systems that automate support activities and also have vague requirements. There is a delivery at least once per year, with extensive user evaluation.

[1] As will be seen in section 6 the use of multiple effort estimation activities was used to identify a Cost Assessment life cycle phase.

Documentation is kept to a minimum. The requirements are revisited with every delivery and a new rank ordering of the requirements is produced. Formal military systems follow DOD-STD-2167A. The R&D tasks cover a wide range of types of software from artificial intelligence to human-computer interface to network protocols. The evolving ground systems consist of software that supports the Deep Space Network and Space Flight Operations Center. Flight software consists of on-board or flight support software, such as software that helps to develop the navigation commands. Both ground and flight systems follow the JPL Software Management Standard. Our analysis indicates that forecasters working with Rapid Prototyping systems use assessment more extensively, Flight and Formal Military systems use size estimates more extensively, Evolving Ground Systems use New/Old Decomposition more extensively, and the R&D systems are uniform across the different key activities. The implications of these results are that, while there is diversity in engineering-based costing approaches, there is also a clustering around a few basic techniques.

Table 2: Sample Breakdown by Type of System and Forecasting Technique

System	New/Old Decomposition	Assessment	Size Estimate	Other	System Type Percentage
Rapid Prototype	20 %	60 %	20 %		13 %
Formal Military		20 %	80 %		13 %
Research	18 %	18 %	27 %	37 %	28 %
Evolving Ground System	43 %	21 %	14 %	21 %	36 %
Flight		25 %	75 %		10 %
Technique Percentage	23 %	26 %	33 %	18 %	100 %

5.0 Software Forecasting Life Cycle

As the focus of the analysis shifted from a static, or snapshot, view of what activities were verbalized to a dynamic view of the data, or time sequencing of the activities, the variation in the mental models due to personal style became even more apparent. The result is that most summaries of the mental models basically produced a blur. This is shown very well by the graph in Figure 5 , which maps the sequence of activities to the order that they were verbalized.

Thus, we needed objective criteria by which to partition the set of verbal protocols to determine if there was any clustering. The criteria could either partition the cases or partition time. As discussed above (see Section 1), a number of approaches were tried . These were refined as described in Section 4 to actually correlate the types of software systems with use of specific decomposition and estimation activities. However, this was not enough, as analysis of the probability transition networks revealed the existence of cyclic behavior. Breaking up these cycles required that the mental models be partitioned over time as well. One systematic way to define a partitioning over time is to specify a forecasting life cycle. Four phases were initially identified; Problem Definition, Problem Analysis, Cost Determination and Cost Assessment. Due to the nature of the verbal reports, it was not possible to distinguish between the first two phases, so for purposes of analysis they were combined into a single Problem Definition and Analysis phase.

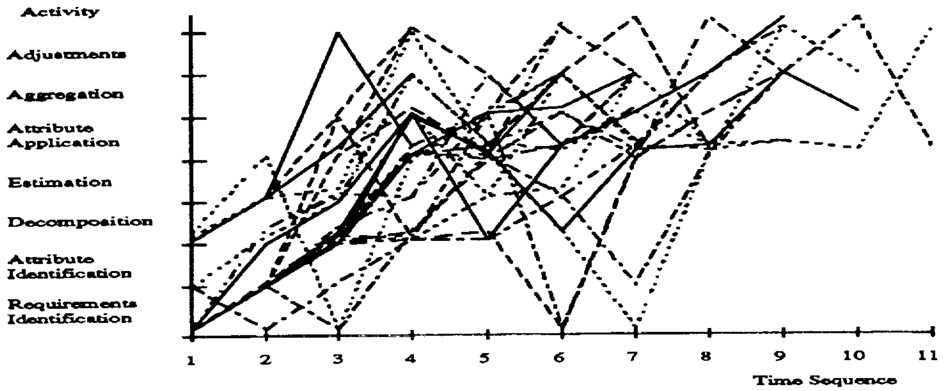


Figure 5 : Graphical Summary of Time Sequence of Activities (Hihn et. al., 1993)

Table 2: List of Activities by Cost Forecasting Phase

Problem Definition and Analysis	Cost Determination	Cost Assessment
Attribute Identification Attribute Application Requirements Identification Decomposition WBS New/Old	Attribute Identification Attribute Application Estimation Size Effort Cost Aggregation Adjustment	Attribute Application Estimation Informal Effort Formal Effort Evaluation

The assignment of activities, in the sample, to the phases is displayed in Table 2. The assignment is based on the protocols that were available. It is expected that the number of activities, with further studies, could increase in each phase due to access to more detailed protocols. Some activities, such as attribute Identification and Application, are ubiquitous, appearing in every phase. Other activities appeared only once, for example, Requirements Identification and Decomposition appeared only as part of the Problem Definition and Analysis phase. Some care had to be taken in determining when a verbal report transitioned between phases. The transition between Cost Determination and Cost Assessment was signalled by phrases such as "and then we did a backup estimate" or "compared our estimated cost to what it cost last time." The transition between the Problem Definition and Analysis phase and the Cost Determination phase was signalled when any type of estimate was mentioned. The one problem that arose in the verbal reports related to Attribute Identification that supported both Decomposition and Estimation activities. When Attribute Identification supported Decomposition, it was recorded in the Problem Definition and Analysis phase; when it supported estimation, it was recorded in the Cost Determination phase. When Attribute Identification occurred on the boundary between the phases, it was recorded as part of the Problem Definition and Analysis phase. In only one case was there compelling evidence to do otherwise.

Figure 6 displays how this costing life cycle relates to the software development life cycle for the verbal protocols used for this analysis is displayed in Figure 6. Cost estimates were made throughout the life of a software development task. Clearly, the amount of effort put into the different cost forecasting phases changes over the development life cycle. It is believed that, in the early stages of the development life cycle, more time tends to be spent in Assessment due to a lack of information required to do a comprehensive detailed cost estimate. The main changes in our model with respect to the Problem Definition and Analysis phase should be in the level of detail in the decomposition. The overall result should show a decrease in time spent in the first phase because each re-estimate builds on the previous one. The current data does not provide sufficient information to test these hypothesis.

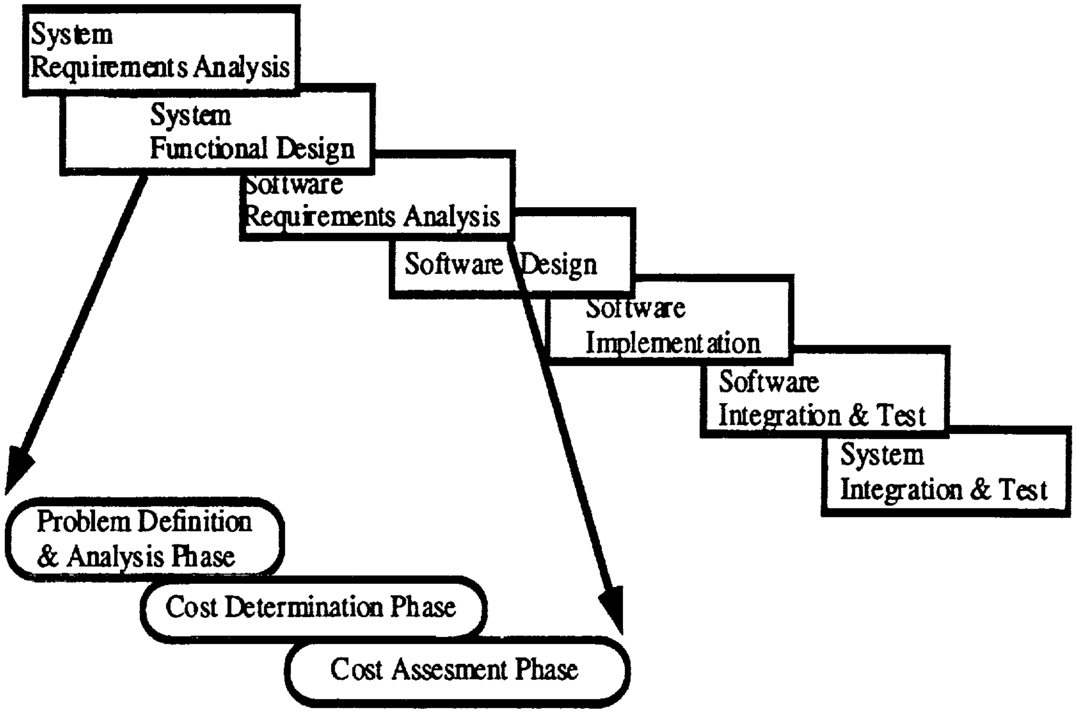


Figure 6 : Forecasting Life cycle Compared to the Software Development Life cycle

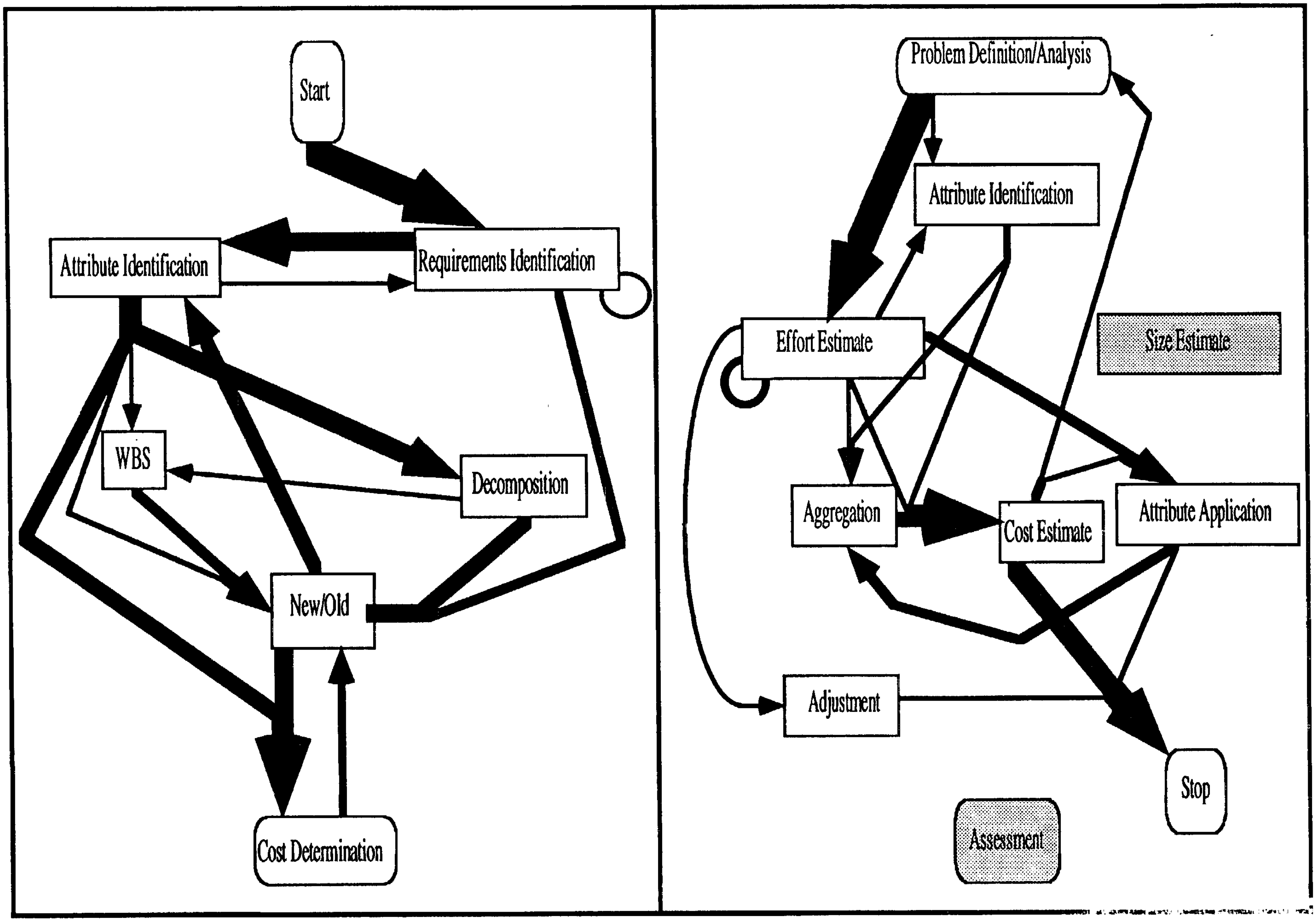
6.0 Software Forecasting Mental Models

The forecasters' mental models can be represented, using Markov process modeling, by activity flow diagrams. It was possible to identify four mental models that partitioned the data. The activities and their transitions for each mental model are shown in Figures 7 through 11. Figure 7 shows the mental model of those who always used a New/Old Decomposition to support their cost estimate. Figure 8 shows the mental model of those who always used a size forecast to support their cost estimate. Figure 9 shows the mental model of those who always used an assessment effort estimate to support their cost estimate. Figure 11 shows the mental model of those who used both size and assessment. Figure 10 shows the activities and sequences for everyone in the sample who had a cost assessment phase. The thickness of the line indicates the number of transitions between activities, making it easier to visually discern where the major activity transitions occur. The thickness of the line is 2 pixels for each observation.

New/Old Only

Problem Definition and Analysis Phase

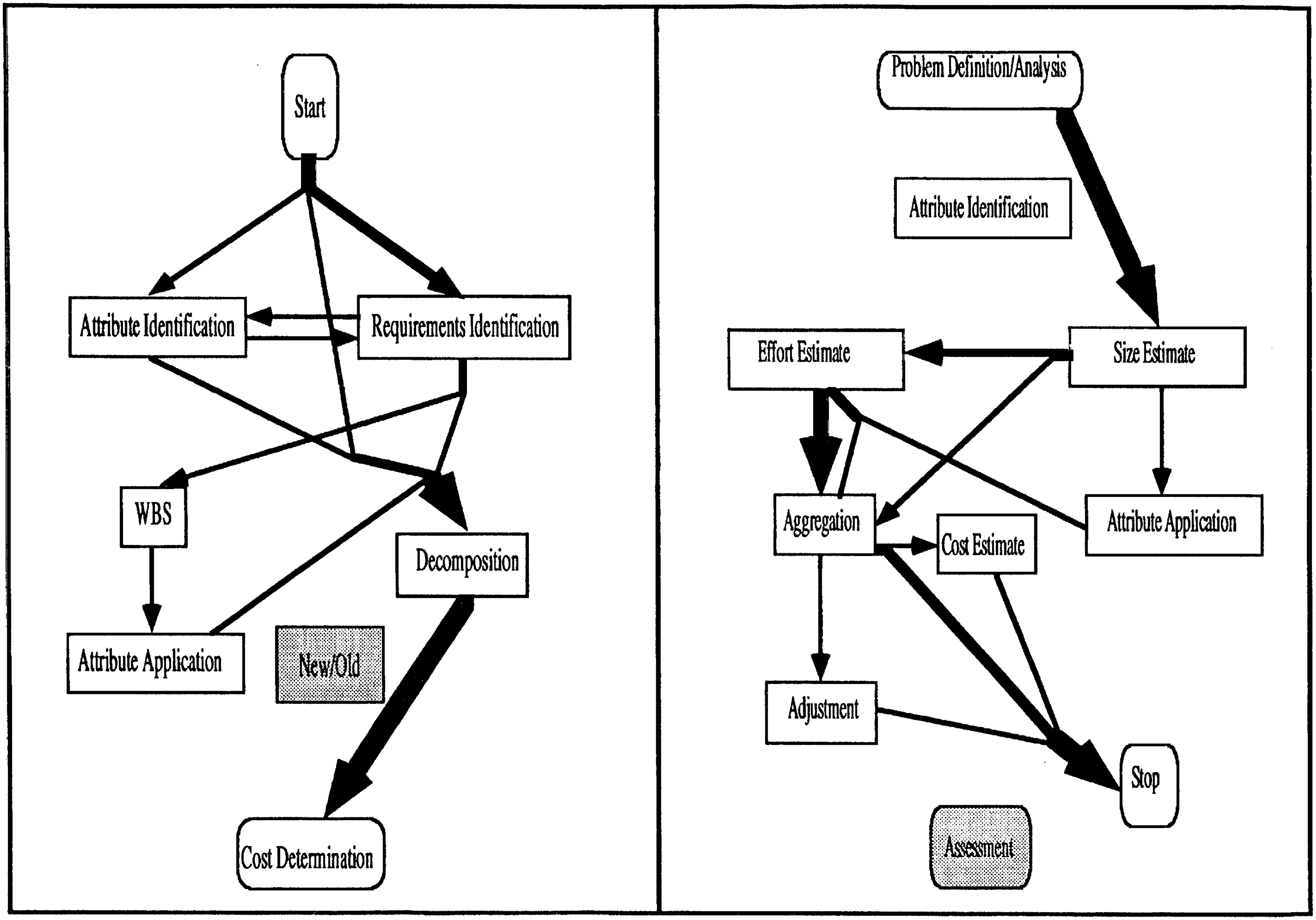
Cost Determination Phase



Size Forecasts Only

Problem Definition and Analysis Phase

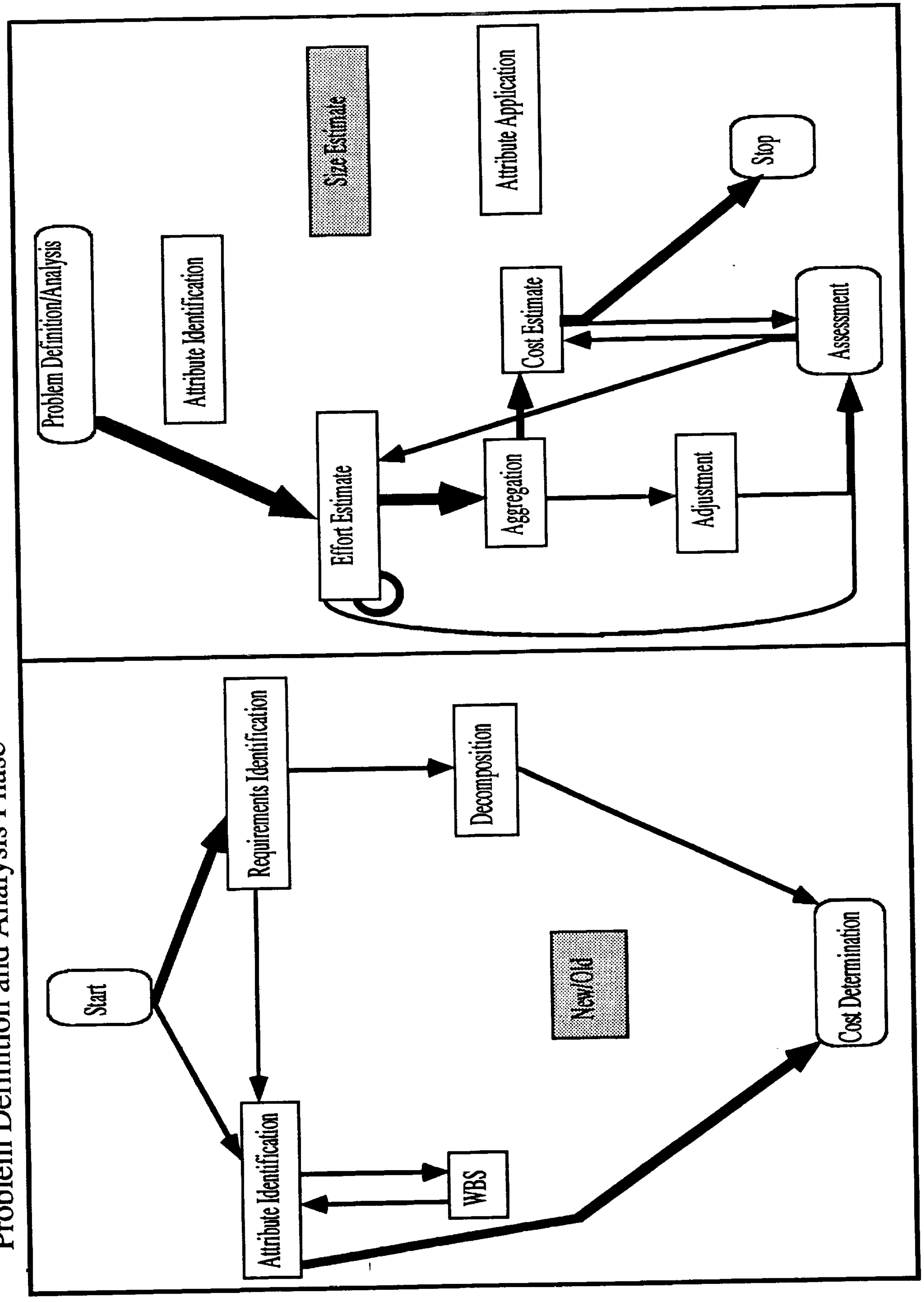
Cost Determination Phase



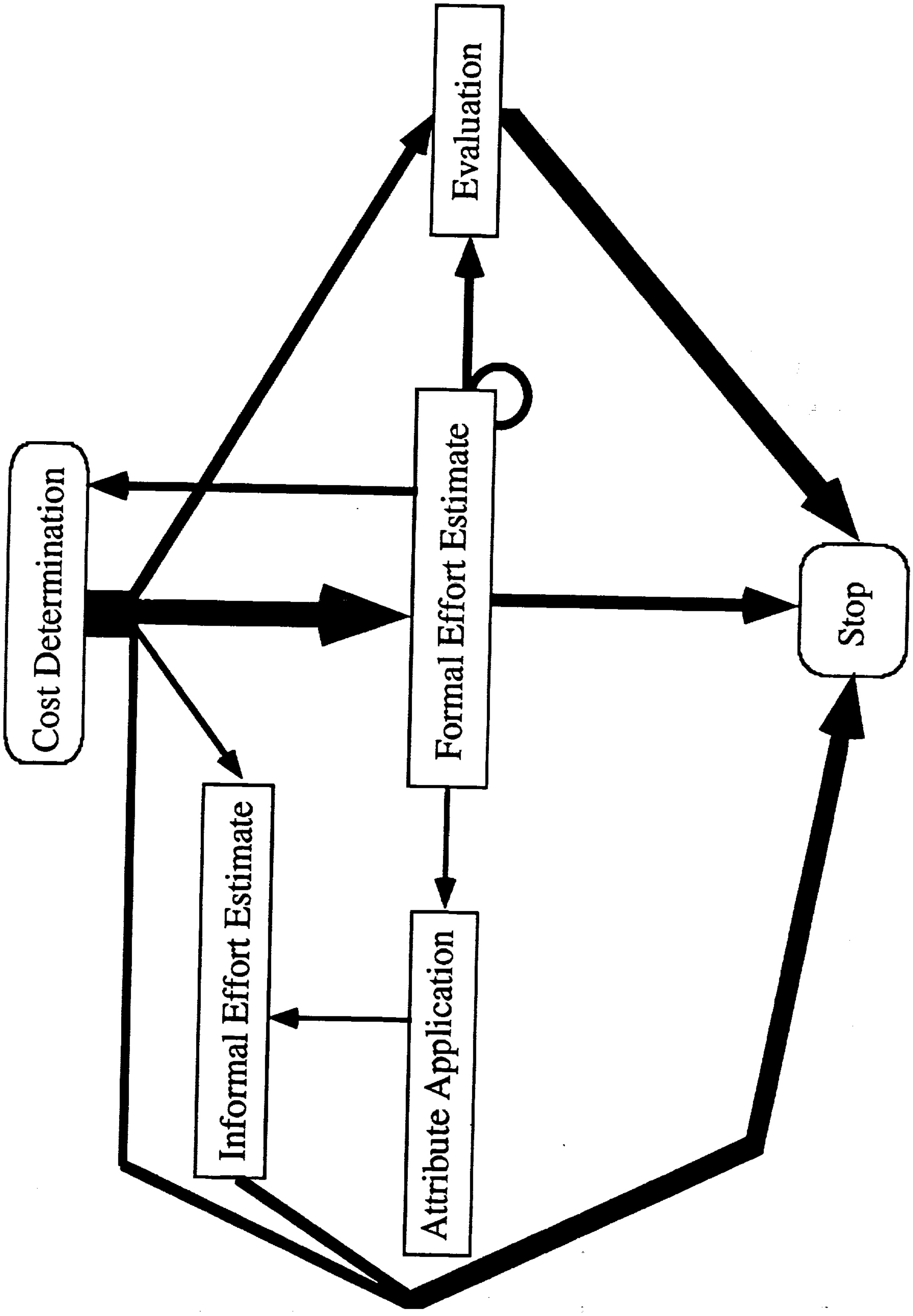
Assessment Forecasts Only

Problem Definition and Analysis Phase

Cost Determination Phase



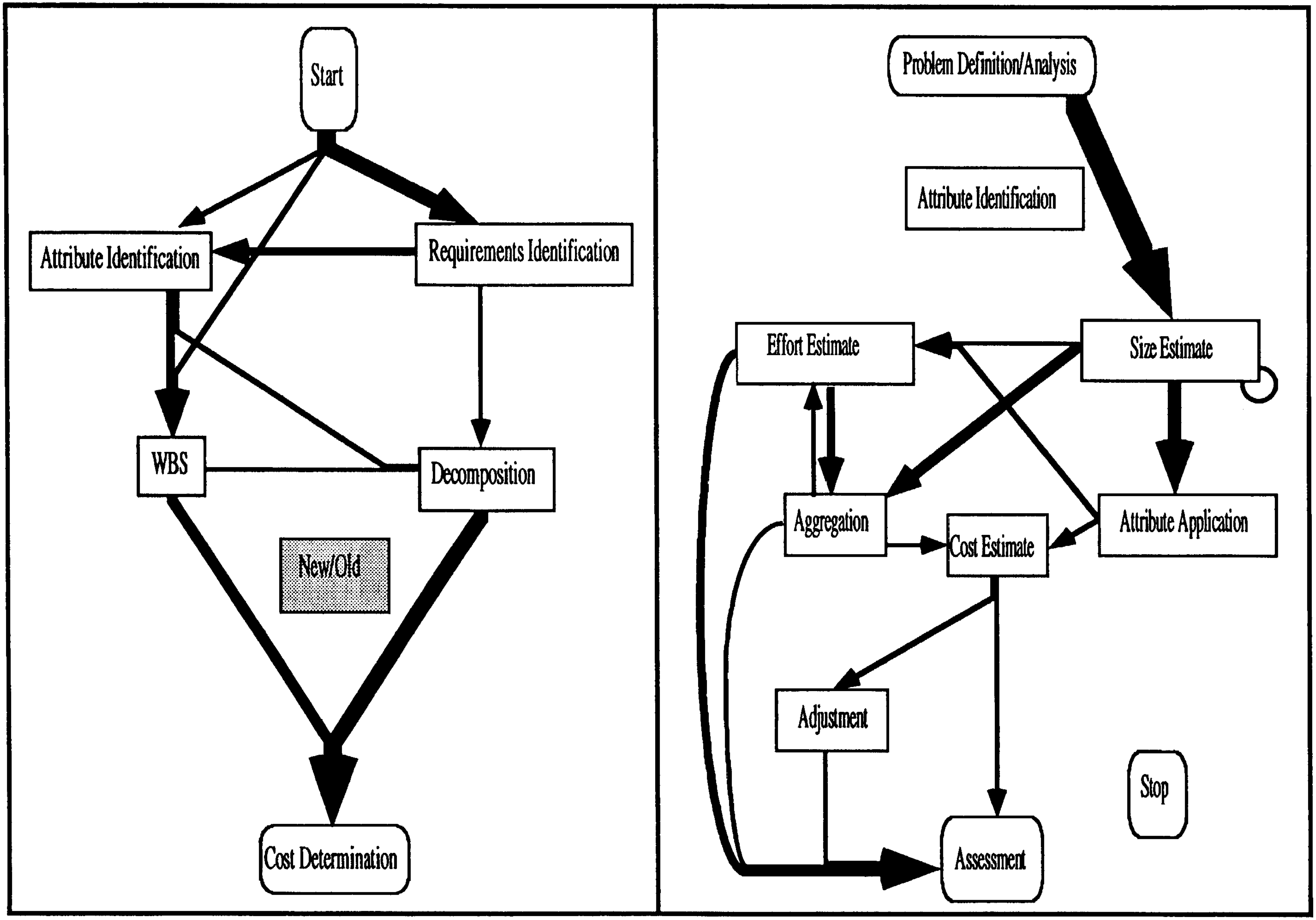
Cost Assessment Phase



Size and Assessment Forecasts

Problem Definition and Analysis Phase

Cost Determination Phase



Start

Attribute Identification

Requirements Identification

WBS

Decomposition

New/Old

Cost Determination

Problem Definition/Analysis

Attribute Identification

Effort Estimate

Size Estimate

Aggregation

Cost Estimate

Attribute Application

Adjustment

Stop

Assessment

Note that in Figures 7-9 and 11 the Effort Estimate, Aggregation, and Cost Estimate activities are shaded in grey because there was some difficulty in discerning the actual sequence of these activities. This was primarily due to the way in which the System Resource Management (SRM) Tool, a cost accounting tool, was used. In many cases the respondent simply said and then "run an SRM". This tool can be used in a variety of ways, however, because it aggregates effort levels, adds planned procurement expenditures, and calculates overhead rates. It was frequently not clear how detailed the work was in determining the effort levels and procurements. Therefore, one level of interpretation of these activities in the mental models was simply into and out of the box that represents the combination of Effort Estimate, Aggregation and Cost Estimate.

It can be seen that, while there is a variety of activity sequences for each cost life cycle phase, there is also a clear dominant route. In Figure 7, the New/Old Decomposition Mental Model, the route was Requirements Identification, Attribute Identification, Decomposition (usually functional), New/Old Decomposition, a branch between exiting to the Cost Determination Phase or repeating Attribute Identification, finally exiting to the next phase. The Cost Determination Phase is less clear but the most likely route appears to have been: Effort Estimate, Attribute Application, Aggregation, Cost Estimate, Stop.

The dominant routes for the other mental models,while having similarities, do differ. Table 4 presents a summary of the sequence of activities for the main paths of the four mental models. Two interesting behavior patterns appear: the increased use of attributes among those using New/Old Decomposition and the lack of a Decomposition activity on the dominant path for those using only Assessment. The latter most likely occurs because those who reported only using Assessment did not have sufficient access to information: either because these were done as early, high level estimates or cost estimates for R&D tasks. In the New/Old mental model, the increased use of Attribute Identification reflects the impact of grouping functions by degree of inheritance. This is important because how the effort estimate was made depends upon the degree of experience of those developing the functions.

Table 4: Activity Sequence Summary of Major Activity Transitions for Forecasting Mental Models

Activity	New/Old	Size	Assessment	Size and Assessment	
				A	B
Requirements Identification	1	1	1	1	1
Attribute Identification	2,5		2	2	2
Decomposition	3	2		3	3
New/Old Decomposition	4				
Size Estimation		3		4	4
Effort and Cost Estimate	6,8	4	3	5	6
Attribute Application	7				5
Assessment			4	6	7
Stop	9	5	5	7	8

A cursory review of the different mental models revealed to us that there exist substantial personal style variations because there seems to be no one way to get a job done. However, there were dominant pathways, and the mental models are clearly different. We interpret the primary differences in the mental models as representing the different ways that forecasters attempted to reduce risk in their cost forecasts. The risk reduction techniques were based upon the use of either multiple-decomposition steps, in this case additional New/Old Decompositions or multiple forecasting steps. The multiple forecasting steps involve either forecasting software size or an additional effort forecast (Assessment). Very few used these risk reduction steps in combination.

7.0 Summary and Conclusions

A viable process for capturing and analyzing the mental models software engineers use for cost and size forecasting has been demonstrated. Our analysis demonstrates the existence of three interdependent cost forecasting life cycle phases. The data analysis of the last few sections provides a basis for us to begin to identify where software engineers can best use supporting methods, tools, and data. Unfortunately, the currently available costing methods and tools only support the Cost Determination phase. Methods, tools and data are needed that will:

support sequential estimation steps

support different techniques, save and assist in comparing results

store design information and supporting estimates

provide assistance in identifying task analogies

In addition the idiosyncratic nature of the individual protocols indicates that supporting methods and tools need to capture and record the steps followed and information used by the forecaster.This will provide a record of the assumptions and context within which the estimate was made, and should improve the quality of updated estimates.

Finally, previously published analysis of this data showed that for experienced forecasters, those who forecast frequently (at least every 6 months) on the average forecast effort 12% high, whereas those who forecast less frequently (at greater than 6 month intervals) on the average forecast effort 44% low. This suggests examining the mental models of those activities and transitions most dependent on memory and determining corrective support methods, tools and data.

forecasts. The risk reduction techniques were based upon the use of either multiple-decomposition steps, in this case additional New/Old Decompositions or multiple forecasting steps. The multiple forecasting steps involve either forecasting software size or an additional effort forecast (Assessment) Very few used these risk reduction steps in combination.

7.0 Summary and Conclusions

A viable process for capturing and analyzing the mental models software engineers use for cost and size forecasting has been demonstrated. Our analysis demonstrates the existence of three interdependent cost forecasting life cycle phases. The data analysis of the last few sections provides a basis for us to begin to identify where software engineers can best use supporting methods, tools, and data. Unfortunately, the currently available costing methods and tools only support the Cost Determination phase. Methods, tools and data are needed that will:

support sequential estimation steps

support different techniques, save and assist in comparing results

store design information and supporting estimates

provide assistance in identifying task analogies

In addition the idiosyncratic nature of the individual protocols indicates that supporting methods and tools need to capture and record the steps followed and information used by the forecaster.This will provide a record of the assumptions and context within which the estimate was made, and should improve the quality of updated estimates.

Finally, previously published analysis of this data showed that for experienced forecasters, those who forecast frequently (at least every 6 months) on the average forecast effort 12% high, whereas those who forecast less frequently (at greater than 6 month intervals) on the average forecast effort 44% low. This suggests examining the mental models of those activities and transitions most dependent on memory and determining corrective support methods, tools and data.

References

Albrecht, A. and J. Gaffney, ``Software Function, Source LOC and Development Effort Prediction, A Software Science Validation," **Transactions of Software Engineering**, Vol. SE-9, No. 6, November, 1983, p. 639-648.

Boehm, B., **Software Engineering Economics**, Prentice Hall, 1981.

Ericson, K. and Simon, H., **Protocol Analysis**, MIT press, 1984

Hihn, J, Griesel, A., Bruno, K, Fowser, T., and Tausworthe, R., Mental Models of Software Forecasting, **Proceedings of the 15th Annual Conference of the International Society of Parametric Analysts**, San Francisco, Ca, June 1-4, 1993, pp. K2-K28.

Hihn, J. and Habib-agahi, H., Cost Estimation of Software Intensive Projects: A Survey of Current Practices, **Proceedings of the 13th International Conference on Software Engineering**, May 17-19, 1991, pp. 276-287.

Howard, M., "The Creation of a Research Model for Estimation", **Proceedings of the European Software Cost Modelling meeting 1992** (ESCOM), Munich, germany, May 27-29, 1992.

Papoulis, A., **Probability, Random Variables and Stochastic Processes**, Mcgraw-Hill Inc, 1991.
Vicinanza, S., Mukhopadhyay and Prietula, M., Software-Effort Estimation: An Exploratory Study of Expert Performance, **Information Systems Research**, vol 2, December 1991, pp. 243-262.

Software Cost Forecasting As It Is Really Done: A Study of JPL Software Engineers

NASA Goddard Space Flight Center
SEL Software Engineering Workshop
December 1-2, 1993

Martha Ann Griesel
Jairus M. Hihn
Kristin J. Bruno
Thomas J. Fouser
Robert C. Tausworthe

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91109

Why Should We Care How Experts Forecast Software Costs?

In today's cost-constrained environment, cost estimation is an integral part of the engineer's job

Therefore, tools and databases are needed to support integrating cost analyses with traditional engineering practices

Previous surveys have shown that engineers in general do not use tools and databases they perceive to be inconsistent with their software cost forecasting mental models

The purpose of this study was to determine the requirements for methods, tools and databases that are consistent with engineers' software cost forecasting mental models

Questions That Needed to be Answered

Is there a set of well-defined software cost forecasting activities?

Do these activities combine into a small number of well-defined mental models?

To what degree are differences in the mental models dependent upon personal style, problem domain, and environment?

Do the different mental models fit within a single software forecasting lifecycle?

How can a better understanding of existing software cost forecasting practices improve the implementation of those practices?

Background

Literature on mental models of forecasting is sparse:

results are not repeatable

previous studies support the assumption that there are a small number of basic activities

Previous work by the authors identified more rigorous methods of data capture and analysis

Cognitive Psychology provides a method of data capture (Protocol Analysis)

Stochastic processes provide a method of analysis (Transition Probability Matrices)

Background (Cont.)

The current analysis uses data that existed from a previous study

We have been able to identify 28 observations that provide sufficient detail for analysis

Respondents types and number of years of software experience varied

Protocols reflect forecasts made during either System Architectural Design or Software Requirements Analysis

Forecasting Activities

Requirements Identification

Attribute Identification - People, Product, Process

Attribute Application - People, Product, Process

Decomposition - WBS, New/Old, Functional, Requirements

Aggregation

Size Estimation - Expert Judgement, Analogy, Rules of Thumb, CER

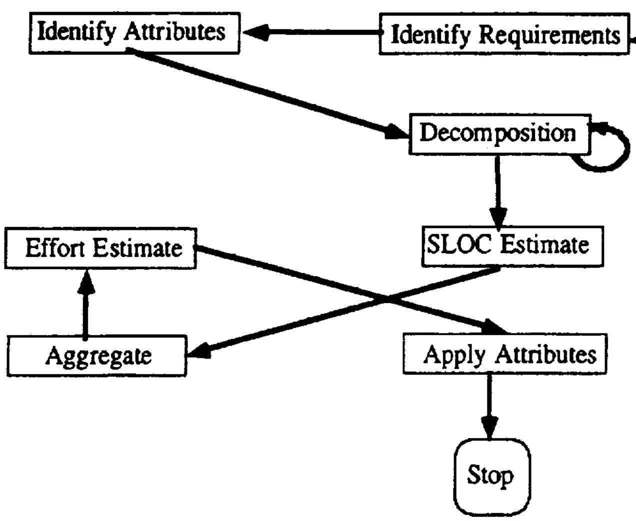
Effort Estimation - Expert Judgement, Analogy, Rules of Thumb, CER

Cost Estimation - Generic, SRM

Adjustment - Risk, Scaling, Bias

Evaluation

An Example of a COCOMO Software Cost Forecasting Mental Model



Start

Identify Requirements

Identify Attributes

Decomposition

SLOC Estimate

Aggregate

Effort Estimate

Apply Attributes

Stop

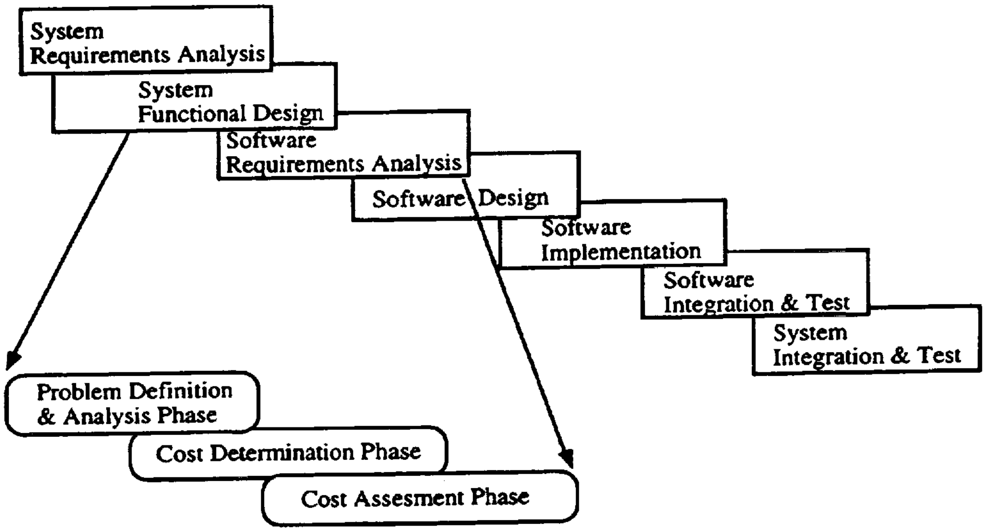
Activity Clustering and Differences

Sample Breakdown by Type of System and Forecasting Technique

System	New/Old Decomposition	Assessment	Size Estimate	Other	System Type Percentage
Rapid Prototype	20 %	60 %	20 %		13 %
Formal Military		20 %	80 %		13 %
Research	18 %	18 %	27 %	37 %	28 %
Evolving Ground System	43 %	21 %	14 %	21 %	36 %
Flight		25 %	75 %		10 %
Technique Percentage	23 %	26 %	33 %	18 %	100 %

Sample Size equals 28, Due to use of multiple techniques total count is 39.

Scope of the Current Software Cost Forecasting Lifecycle Relative to the Software Lifecycle



System Requirements Analysis

System Functional Design

Software Requirements Analysis

Software Design

Software Implementation

Software Integration & Test

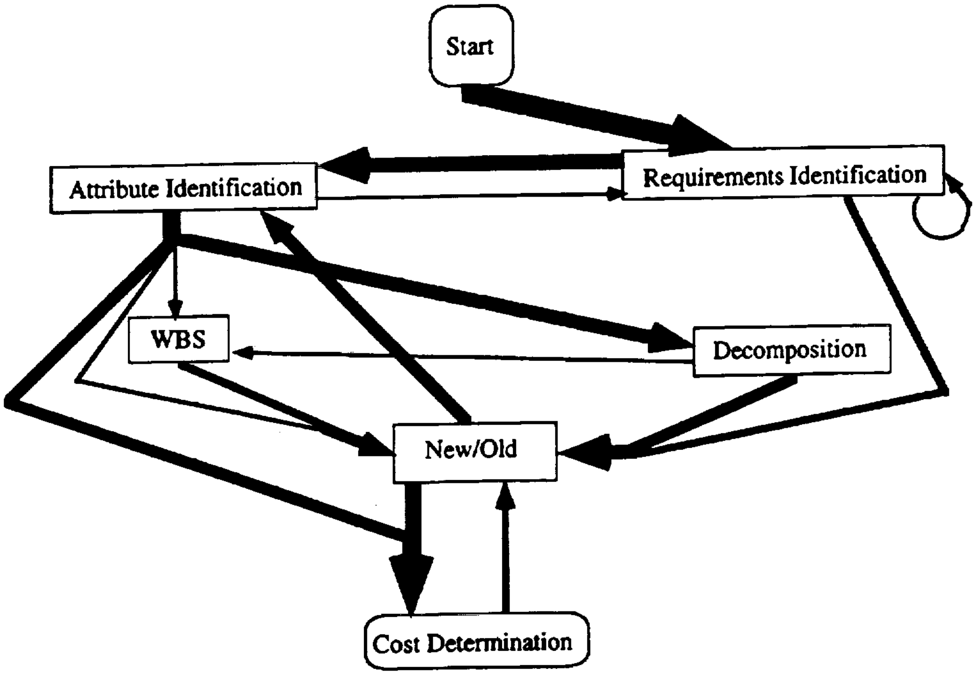
System Integration & Test

Problem Definition & Analysis Phase

Cost Determination Phase

Cost Assesment Phase

Problem Definition and Analysis Phase New/Old Only



Start

Attribute Identification

Requirements Identification

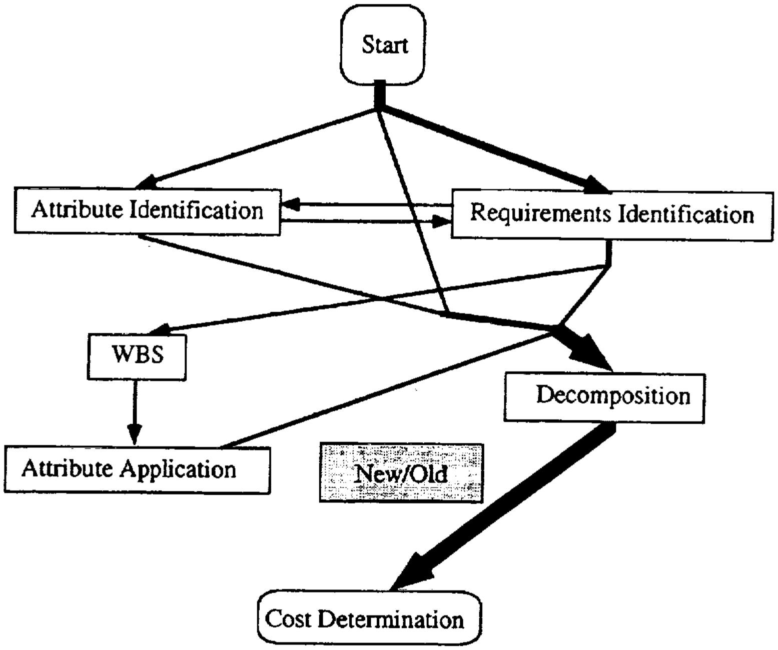
WBS

Decomposition

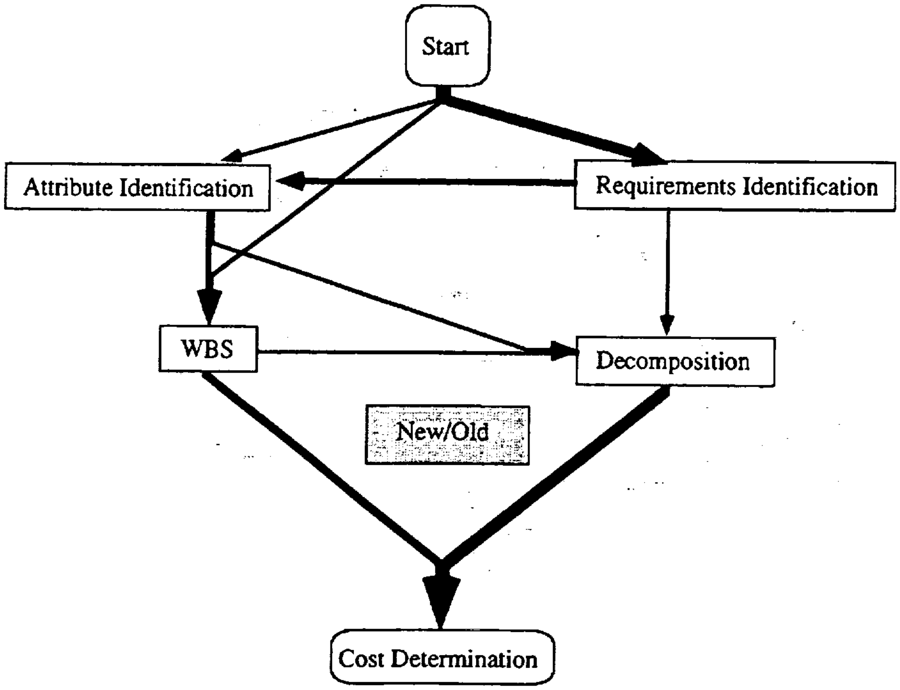
New/Old

Cost Determination

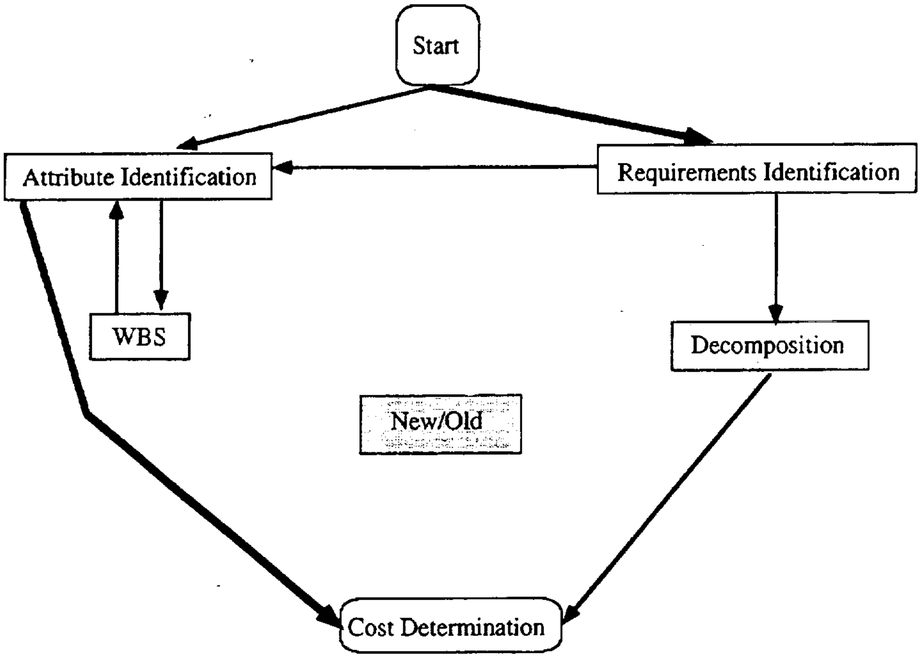
Problem Definition and Analysis Phase
Size Forecasts Only



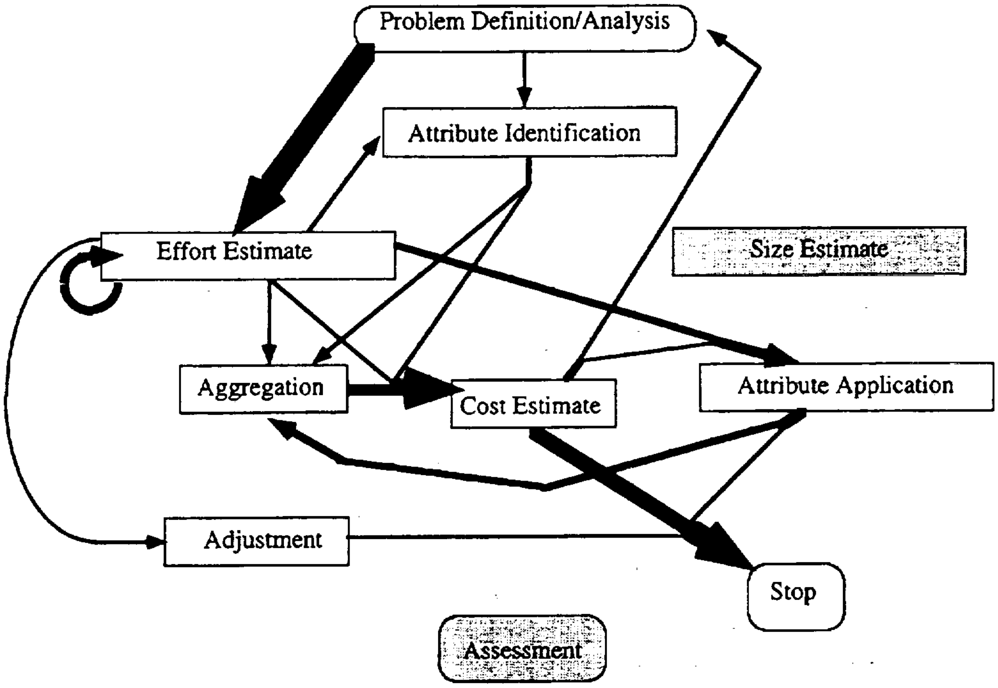
Problem Definition and Analysis Phase
Size and Assessment Forecasts



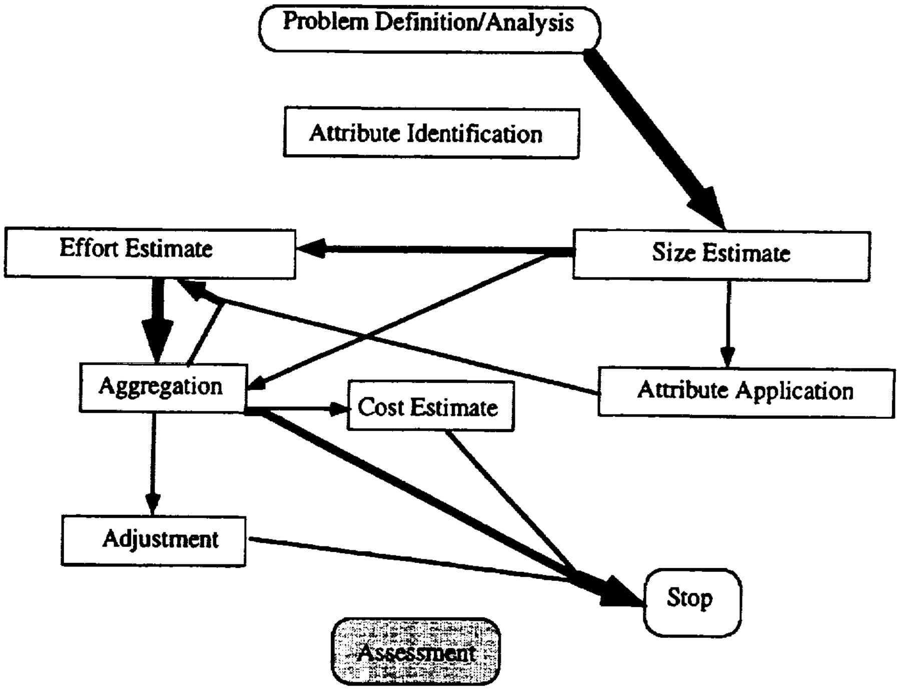
Problem Definition and Analysis Phase
Assessment Forecasts Only



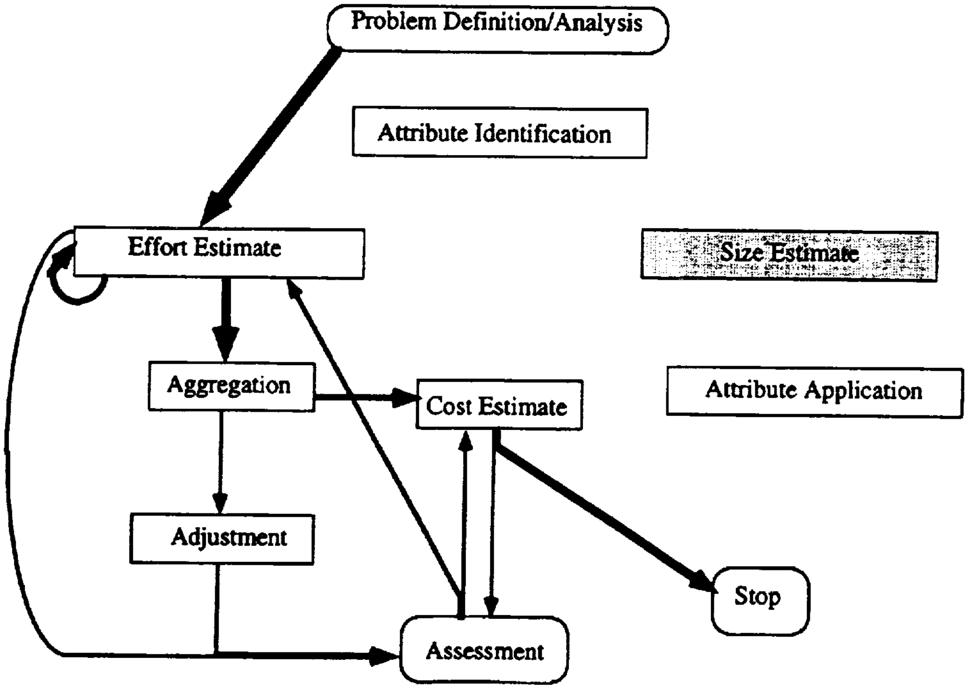
Cost Determination Phase
New/Old Only



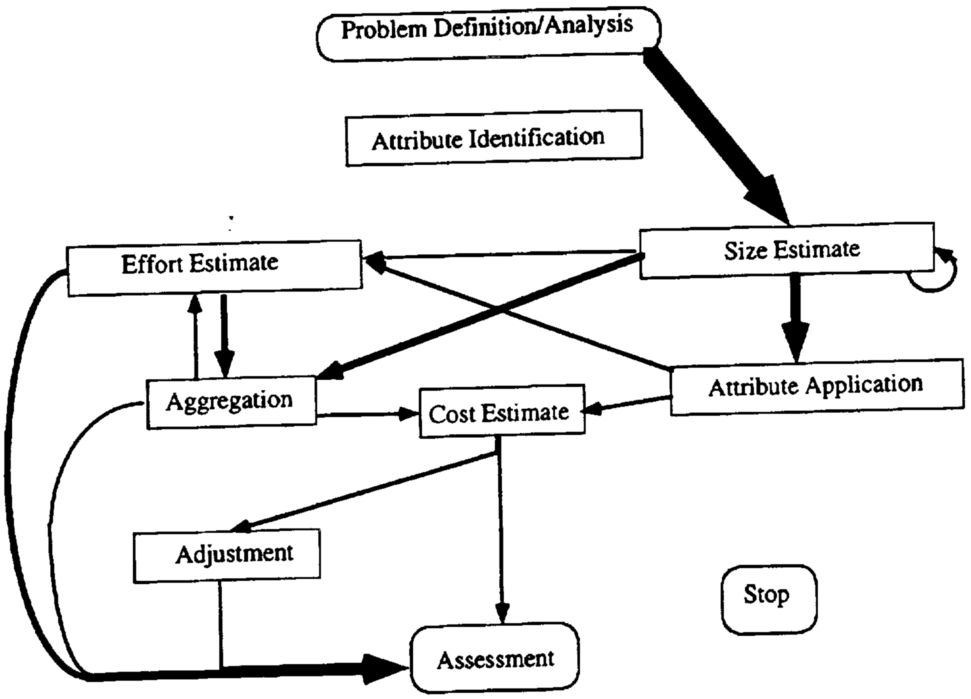
Cost Determination Phase
Size Forecasts Only



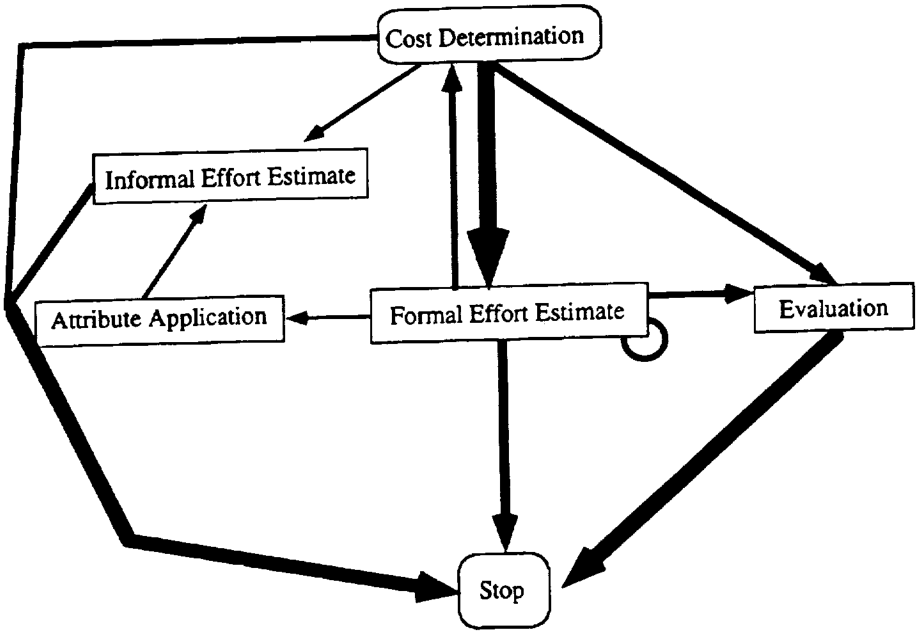
Cost Determination Phase
Assessment Forecasts Only



Cost Determination Phase
Size and Assessment Forecasts



Cost Assessment Phase



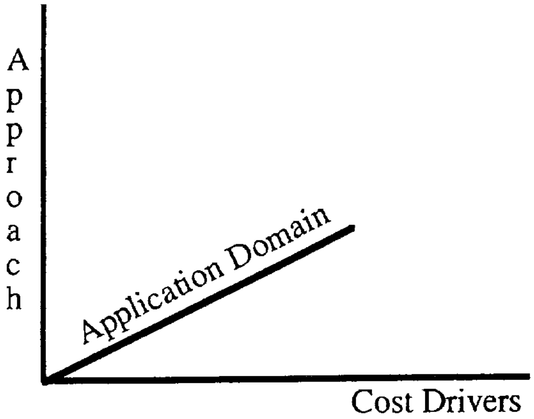
Conclusions

- Expert forecasters use simplification in the face of complexity
  - 86 % only use one technique to reduce cost forecast risk
    - more detailed decompositions
    - more detailed forecasts
  - they keep cost techniques simple and use only a few cost drivers
    - Personnel Quality, Complexity, Language
- Consistent with Cognitive Psychology findings in other fields

Conclusions

- Experts tend to use techniques based on domain knowledge and rules of thumb
  - single domain experts generally get into detailed forecasting quickly
  - multiple domain experts do more abstract or generic forecasting
- Design-To-Cost differs in that
  - Attribute Identification is more likely to be used as a first step
  - forecasts are iterated based upon cost-budget comparison

Summary
Spanning the Mental Model Problem Space



MAG/JMH/JPL/November 29, 1993

11

56-61
12688
p- 15

Assessing Efficiency of Software Production for NASA–SEL Data

Anneliese von Mayrhauser Armin Roeseler

Computer Science Department
Colorado State University
Fort Collins, CO 80523

Abstract

This paper uses production models to identify and quantify efficient allocation of resources and key drivers of software productivity for project data in the NASA-SEL database. While to analysis allows identification of efficient projects, many of the metrics that could have provided a more detailed analysis are not at a level of measurement to allow production model analysis. Production models must be used with proper parameterization to be successful. This may mean a new look at which metrics are helpful for efficiency assessment.

1 Introduction

Many organizations collect a plethora of metrics to help them analyze efficiency of software development and maintenance. Just how helpful are they? We used production models and associated metrics to assessment efficiency of NASA-SEL projects.

While production models have been used in Operations Research for quite a while, their use in the computing field has been limited [1], [10], [13]. One reason for this is that successful development of such models for the software development and maintenance process requires appropriate parameterization (i. e. metrics that are able to help with root cause analysis for process inefficiencies). We can consider software development and maintenance as a production process and model it accordingly. Inputs to the production model are various indicators of resources (effort, tools, capital, expertise, etc.). Outputs reflect the characteristics of software produced (size, quality, etc.). The production model then identifies which development activities were efficient and which factors are related how much to inefficiencies found. This targets specific production activities for improvement. At this point we need to look at more detailed metrics to identify further cause and possible improvement actions. Except for [10, 9] all other applications in the computing field have not provided metrics with the production model that make process improvement possible. Further, [1] only models maintenance requests. To make production models useful for quantitative assessment (and improvement) we must provide a hierarchy of metrics for further analysis of production model results. Both metrics and production model are then bound into a process improvement program.

Section 2 gives a short synopsis of how production models work and how they analyze parameters. A more complete description can be found in [13]. Section 3 reports on the efficiency analysis of NASA-SEL project data. Section 4 gives recommendations for using a production model for efficiency assessment.

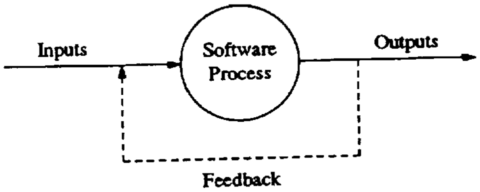


Figure 1: A Simple Causal Model of the Software Process

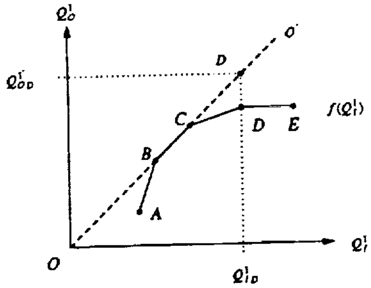


Figure 2: Production Function and Efficiency Frontier

2 Production Models

Production models are causal models that integrate technical and economical analysis perspectives to assess the efficiency of resources used to achieve software development goals. The motivating idea of the production system model is the notion that the software development process transforms resources (e.g., Programmer time, CPU time) into software products. The production function f relates the inputs to the outputs and describes the resource transformation process. We say a software development process is optimized if maximal levels of outputs are attained, given a set of production input quantities.

Figure 1 depicts a production system model that describes how (possibly multiple) input factors (resources) are transformed into (possibly multiple) output factors (e.g., deliverables, quality aspects) using a software development process. Feedback in the production system model is provided through managerial decision making (e.g., deciding on process, project plan and resources).

Because of the complex interactions of software development components, the analytic specification of the production function f is rarely feasible. In the absence of quantitative means to determine the interactions and causalities of components in the production process *directly*, we take an empirical approach to identify optimal production conditions based on historical data on the software development process.

Via linear programming techniques, a convex set of production component data is constructed, and a piecewise linear description of the efficient production frontier is obtained. The efficient production frontier consists of observations that maximize software development goals, given resources consumed, and sets the standard against which other projects or development periods are evaluated. In the process, input efficiencies (slack values), and desired output targets are obtained. Figure 3 shows a production function (A, B, C, D, E) and an efficient production frontier (OO') for a model with one input and one output.

Knowledge of a project's relative (in)efficiency, amount of excess input, and desired output goals can then be used to

- evaluate the efficiency of a SW process,
- decide on strategies to improve efficiency, and
- develop improved SW processes.

3 Production Model Analysis of Project Data in the NASA-SEL Database

We are evaluating project data in the NASA-SEL database with regards to the following:

- does the production model identify efficient and inefficient periods of production?
- are the metrics pinpointing the proper cause?

3.1 Production Model Analysis

We selected 49 projects for analysis. The selection criteria was completeness of project data recorded. We wanted to start our analysis with a rich set of project descriptors. The following projects qualified:

2,6,8,10,19,26,34,35,36,37,38,39,40,46,47,48,49,50,51,52,53,54,55,56, 65,68,70,73,74,80,81,90, 101,102,103,104,105,106,108,110,114,115,116, 117,126,131,132,134,135

Production model analysis must identify inputs and outputs to the production process. These must be at a ratio level of measurement. Rightaway, we face a severe restriction in possible inputs and outputs to the model, since many of the data items are really ranks (e. g. Stability of Requirements). Pretending such data is ratio level is inappropriate, much as we would like to include such key productivity drivers. We decided on a two phase analysis, the first phase uses the (small) set of production inputs and outputs that are at the proper level of measurement. The second phase analyzes rank data, how they appear to influence efficent and inefficient projects.

Input Factors

P132 Total technical and management hours expended on project
P135 CPU hours used

Output Factors

P139 Number of changes made to system components
P141 Total SLOC for all components in the system

The production model clearly identified efficient and inefficient projects. The efficient projects were: 53,54,55,74,110,134. We also included Project 48 with an Efficiency Score of 0.98. The production model clearly identified the periods of inefficiency and the magnitude of inefficient resource usage.

3.2 Metrics Analysis

Next lower level analysis uses the remaining metrics to identify:

- major factors that impact overall project efficiency

- factors that pertain to efficient or inefficient projects only
- Identify factors that most sharply divide efficient from inefficient projects

A properly defined set of metrics is indispensable for successful use of the production model. We found the following results:

1. Factors that correlate with both efficient and inefficient projects.
    - Pos. Correlation (i.e., 'more' is beneficial)
        - P90 Stability of Requirements
        - P100 Stability of Management Team

      While this confirms other analyses (e. g. those underlying the COCOMO model), this correlation by itself does not tell us whether a lack of stability in requirements and management team caused the inefficiency. Had this data been collected at a ratio level of measurement, we could have identified cause.
    - Neg. Correlation (i.e., 'less' is beneficial)
        - P93 Rigor of Requirements Review
        - P115 System Response Time

      Again, this data is ordinal, and gives rise to possible interpretations. One might venture to say that this result indicates that rigor can go overboard and thus causes inefficiencies, but this might also be due to inconsistent data collection, specifically lack of inter-rater reliability for P93. This would point to a need for metrics validation before collecting them on a large scale.
2. Factors that correlate with inefficient projects, but do not correlate with efficient projects. That is, efficient projects are immune to the factors listed below, while inefficient projects are influenced by them.
    - Pos. Correlation (i.e., 'more' is beneficial)
        - P95 Development Team Application Experience

      This is a very interesting result as it appears to say that "if you're not as efficient as you could be, the experts bail you out".
    - Neg. Correlation (i.e., 'less' is beneficial)
        - P88 Problem Complexity
        - P105 Discipline in Requirements Methodology
        - P112 Access to Development System
        - P113 Ratio of Developers to Terminals

      Again, much of this is ordinal data, some might be dependent, so if anything we should investigate this further. What is quite interesting is that the efficient projects don't seem to be affected by this.
3. Factors that most sharply discriminated efficient and inefficient projects (statistics probably not significant)
    - Pos. Correlation (i.e., 'more' is beneficial)
        - P106 Discipline in Design Methodology

- P119 Quality of SW

- Neg. Correlation (i.e., 'less' is beneficial)
  - P91 Quality of Requirements

4. All Other Factors:
   - No significant correlations to efficient and inefficient projects detected.
   - Incomplete data for factors P104 and P117.

While we could identify efficent and inefficient production outcomes for these 50 projects, many of the State-of-the-art metrics in the NASA-SEL database do not have enough power to assess the efficiency of software production in enough detail to suggest improvements. Two problems, subjective ranking and questions about inter-rate reliability are key to the situation. Unfortunately, this is a very common problem.

4 Conclusion

We clearly need methods to assess efficiency of software production. Reliable quantitative methods paired with engineering judgement appear most promising. This paper described production models and how to use them for efficiency assessment. We applied the approach to data from 49 projects in the NASA-SEL database to show its benefits and the needs for better, more relevant metrics to drive any quantitative evaluation.

References

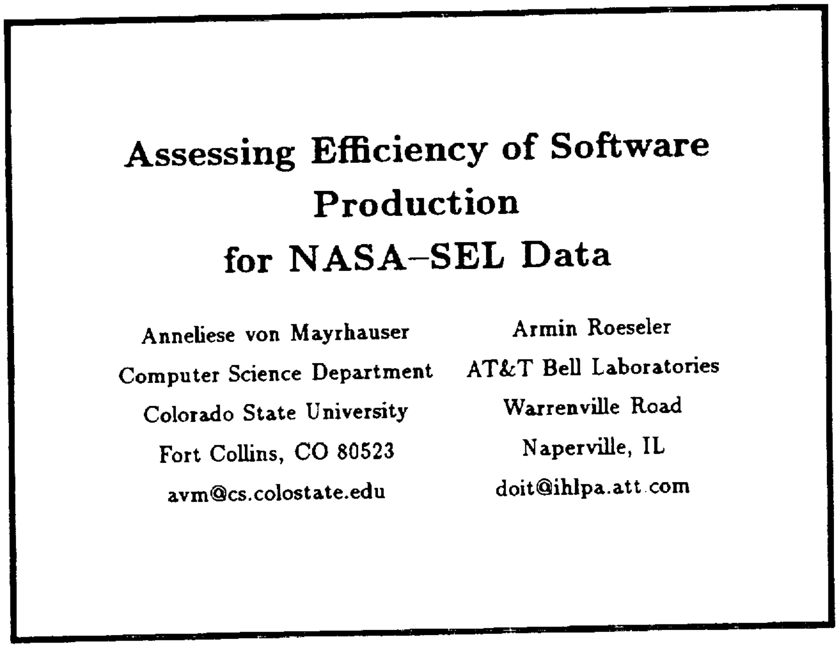
[1] R. Banker, S. Datar, Ch. Kemmerer; "A Model to Evaluate Variables Impacting the productivity of Software Maintenance Projects", *Management Science 37*, 1(Jan. 1991), pp. 1–18.

[2] A. Roeseler; *A Production-Based Approach to Performance Evaluation of Computing Technology*, PhD Thesis, Illinois Institute of Technology, 1991.

[3] A. Roeseler, A. von Mayrhauser, "A Production-Based Approach to Performance Evaluation of Computing Technology", *Journal of Systems and Software*, to appear 1993.

[4] von Mayrhausr, A., Roeseler, A.; "Software Process Assessment and Improvement using Production Models", *Procs. COMPSAC 93*, Nov. 1993, Phoenix, AZ.

Slide 1

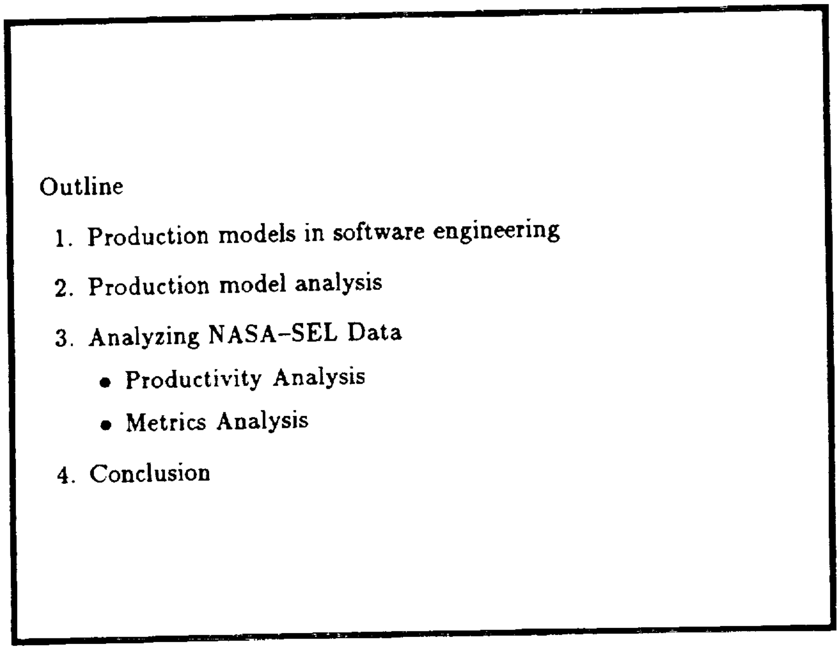


Assessing Efficiency of Software Production for NASA–SEL Data

Anneliese von Mayrhauser
Computer Science Department
Colorado State University
Fort Collins, CO 80523
avm@cs.colostate.edu

Armin Roeseler
AT&T Bell Laboratories
Warrenville Road
Naperville, IL
doit@ihlpa.att.com

Slide 2



Outline

1. Production models in software engineering
2. Production model analysis
3. Analyzing NASA–SEL Data
   - Productivity Analysis
   - Metrics Analysis
4. Conclusion

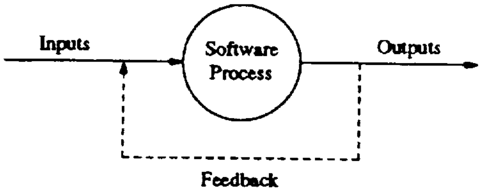
Slide 3

1. Production Models in Software Engineering

- Productivity is multi-faceted
- Analyze software development & maintenance as a microeconomic production process
  - resource transformation (empirically-based)
  - mathematical approach to efficiency measurement
  - incremental process of achieving/maintaining successively higher levels of efficiency

Slide 4

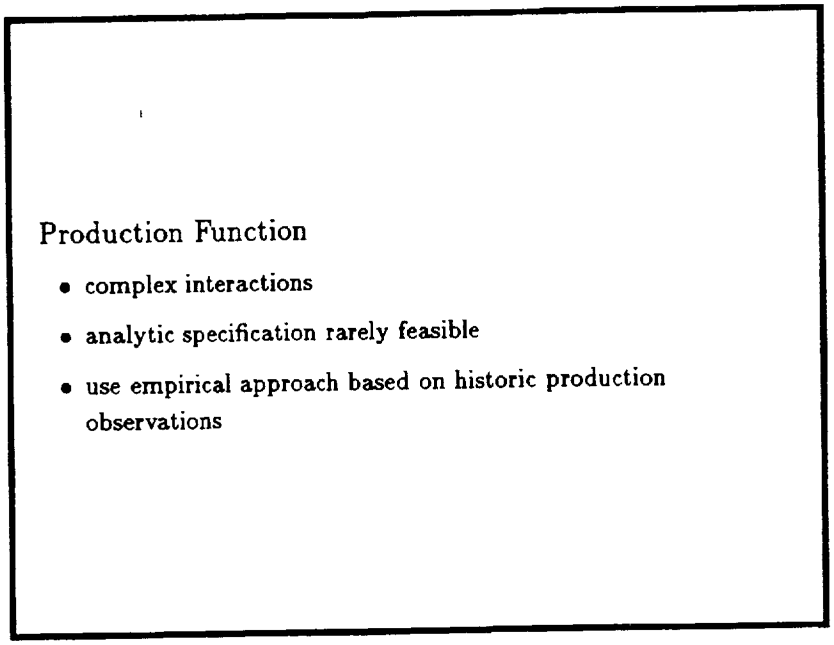
2. Production Model Analysis



$$f(Q_O;Q_I) = f(Q_O^1,\ldots,Q_O^M;Q_I^1,\ldots,Q_I^N) = 0$$

- Q_O^m level of m-th output, $m = 1,\ldots,M$
- Q_I^n level of n-th input, $n = 1,\ldots,N$
- $f : R_+^N \mapsto R_+^M$ production function giving maximal output for given input

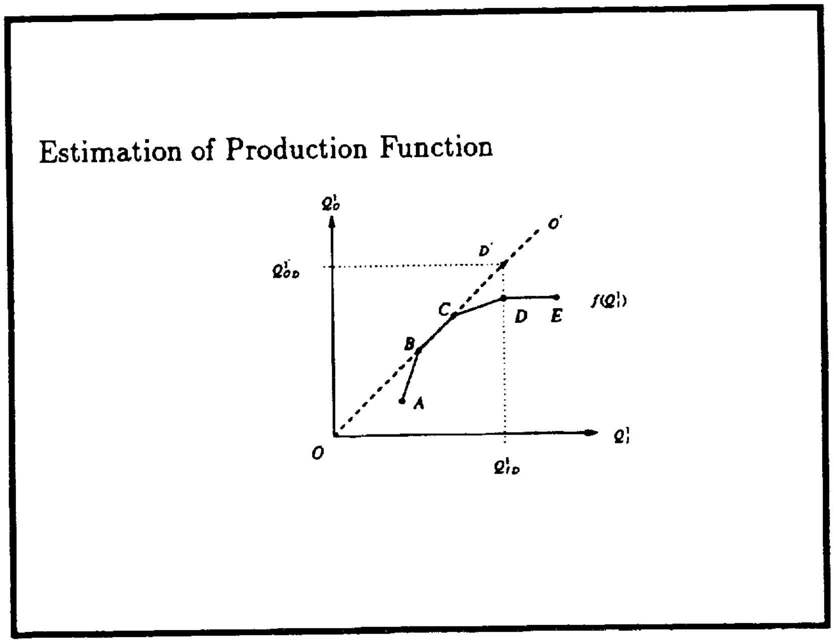
Slide 5



Production Function

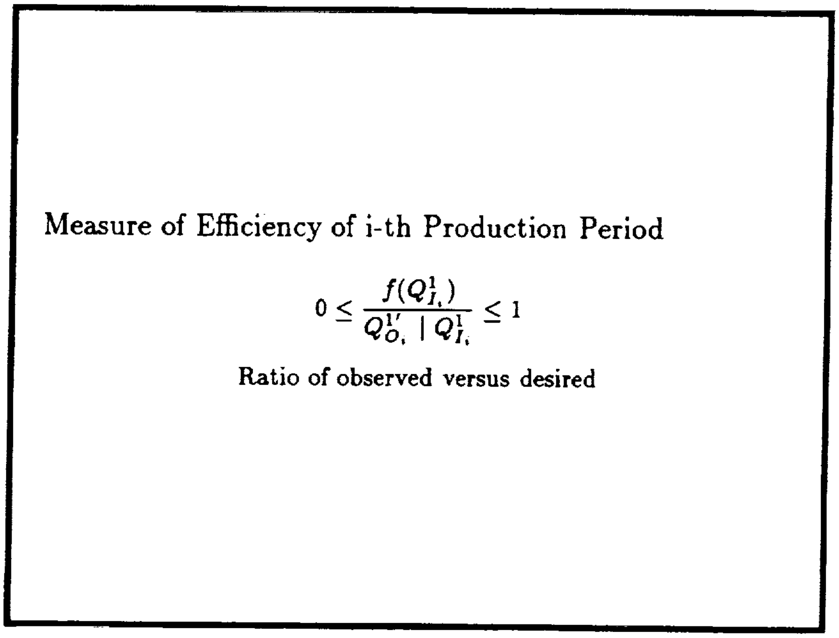
- complex interactions
- analytic specification rarely feasible
- use empirical approach based on historic production observations

Slide 6



Estimation of Production Function

Slide 7

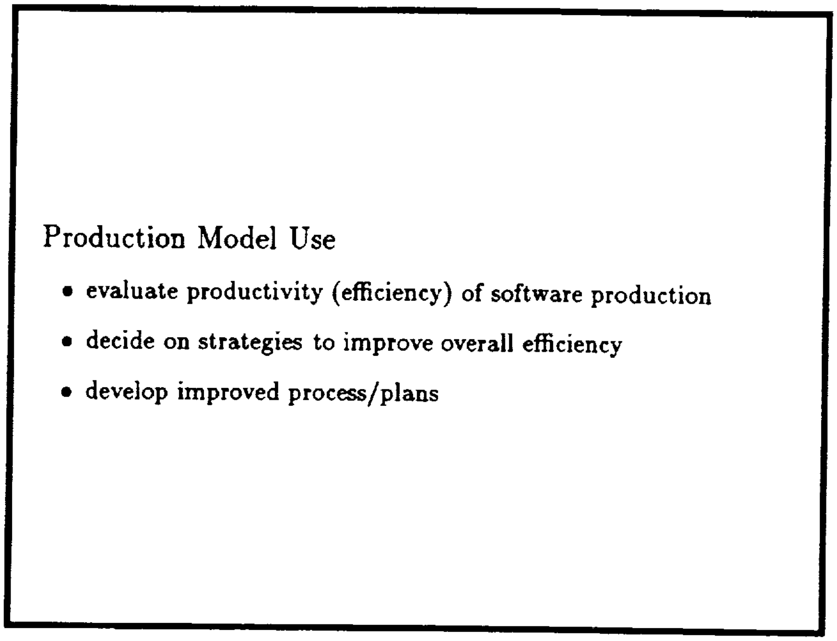


Measure of Efficiency of i-th Production Period

$$0 \le \frac{f(Q^1_{I_i})}{Q^{1'}_{O_i} \mid Q^1_{I_i}} \le 1$$

Ratio of observed versus desired

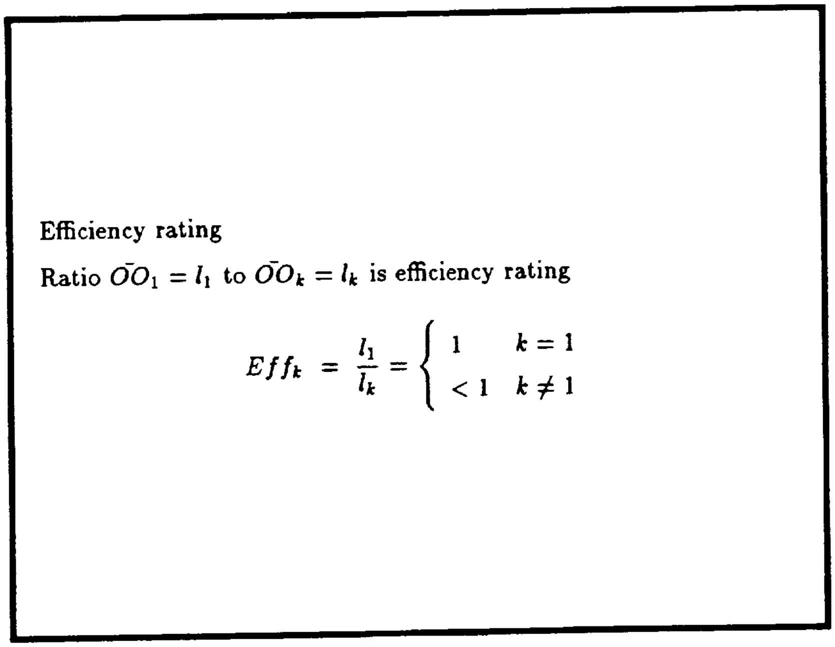
Slide 8



Production Model Use

- evaluate productivity (efficiency) of software production
- decide on strategies to improve overall efficiency
- develop improved process/plans

Slide 9

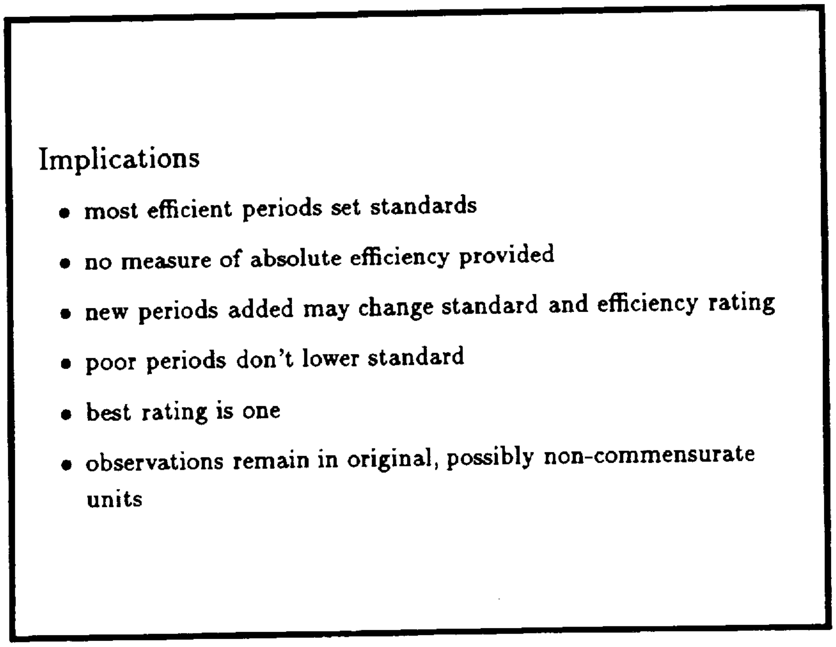


Efficiency rating

Ratio $\bar{O}O_1 = l_1$ to $\bar{O}O_k = l_k$ is efficiency rating

$$Eff_k = \frac{l_1}{l_k} = \begin{cases} 1 & k = 1 \\ < 1 & k \neq 1 \end{cases}$$

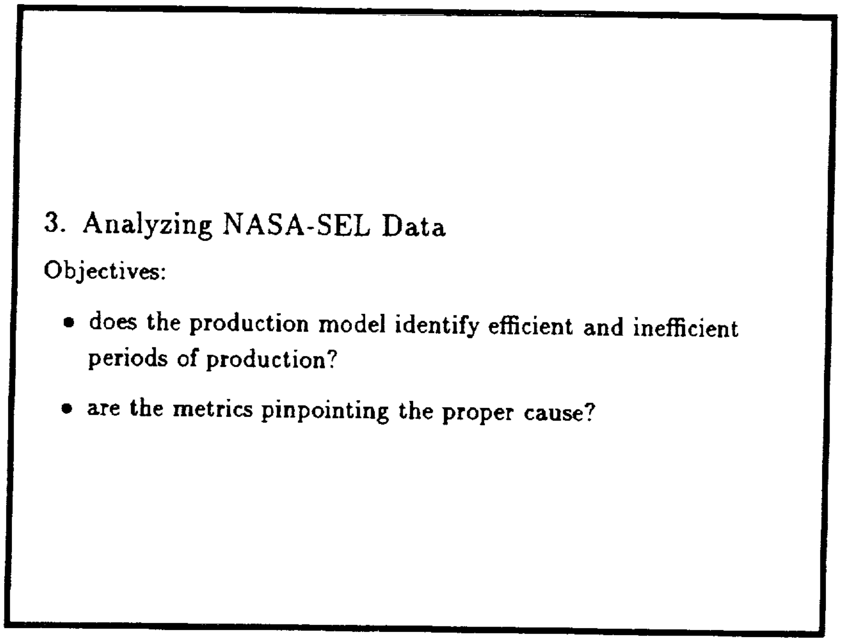
Slide 10



Implications

- most efficient periods set standards
- no measure of absolute efficiency provided
- new periods added may change standard and efficiency rating
- poor periods don't lower standard
- best rating is one
- observations remain in original, possibly non-commensurate units

Slide 11

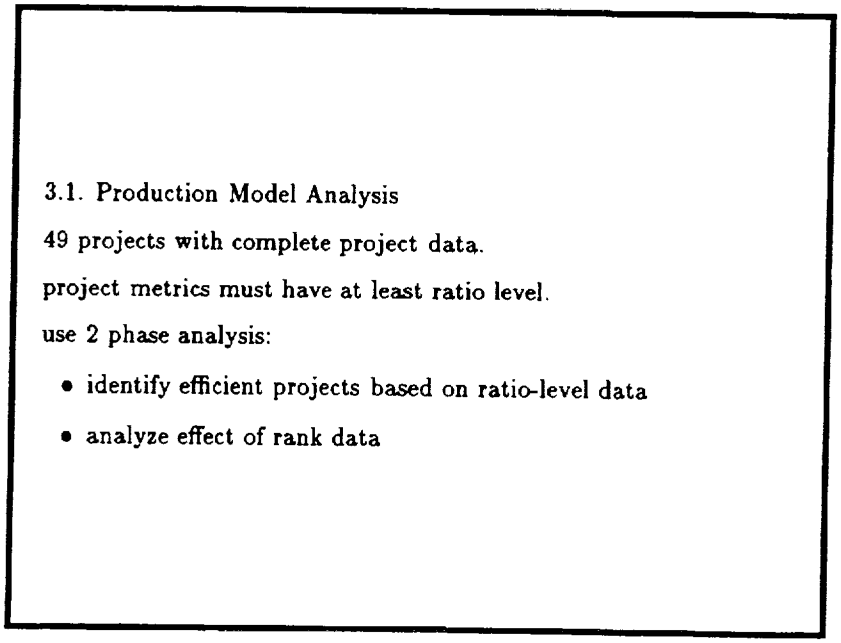


3. Analyzing NASA-SEL Data

Objectives:

- does the production model identify efficient and inefficient periods of production?
- are the metrics pinpointing the proper cause?

Slide 12



3.1. Production Model Analysis

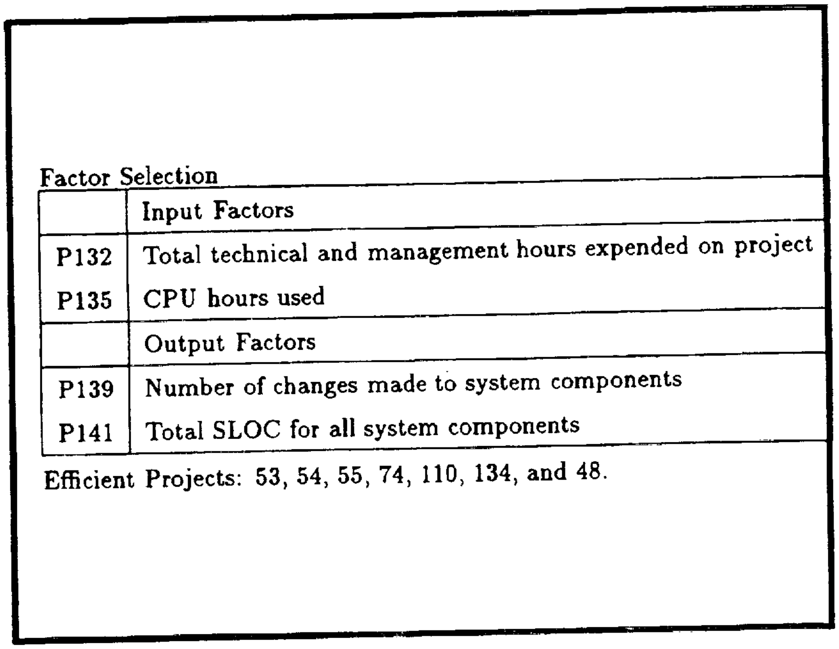
49 projects with complete project data.

project metrics must have at least ratio level.

use 2 phase analysis:

- identify efficient projects based on ratio-level data
- analyze effect of rank data

Slide 13

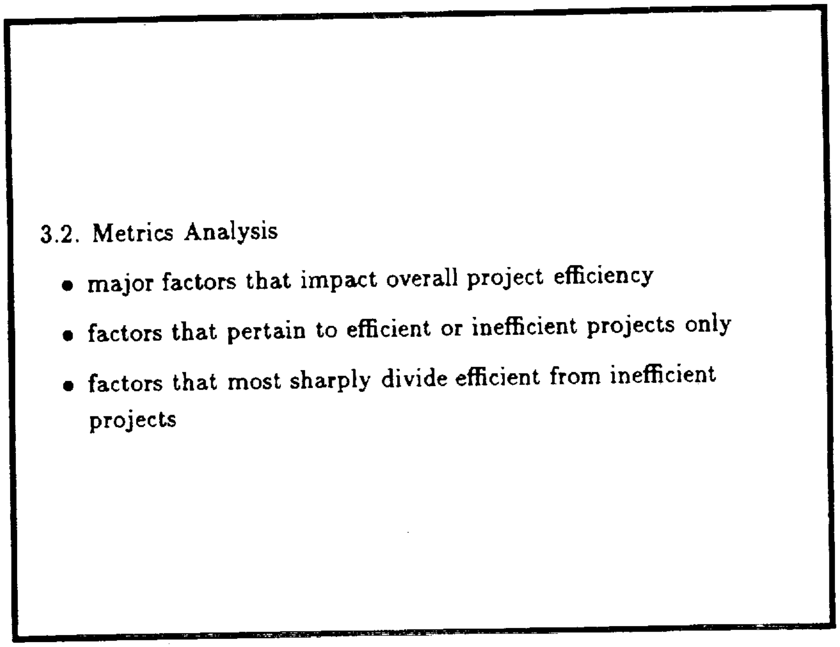


Factor Selection

	Input Factors
P132	Total technical and management hours expended on project
P135	CPU hours used
	Output Factors
P139	Number of changes made to system components
P141	Total SLOC for all system components

Efficient Projects: 53, 54, 55, 74, 110, 134, and 48.

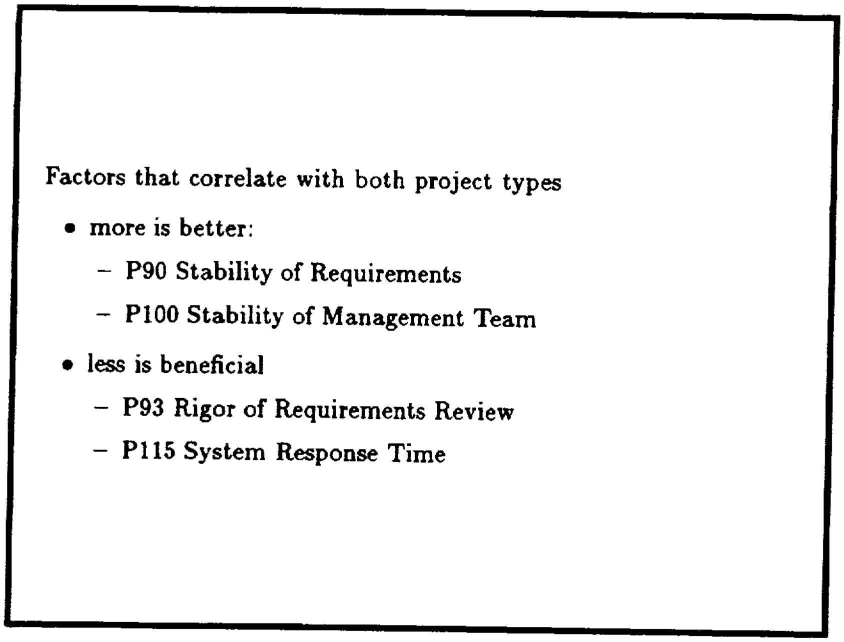
Slide 14



3.2. Metrics Analysis

- major factors that impact overall project efficiency
- factors that pertain to efficient or inefficient projects only
- factors that most sharply divide efficient from inefficient projects

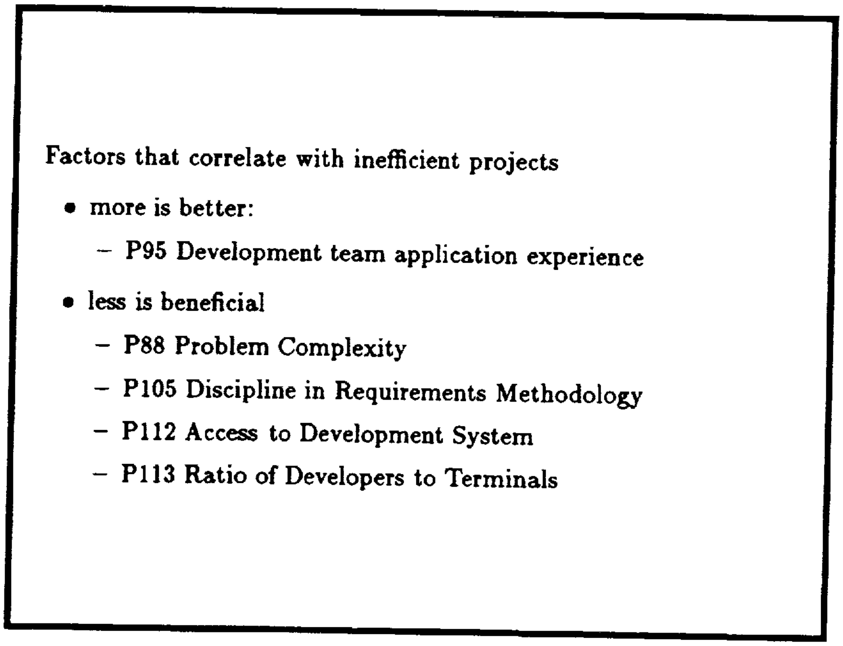
Slide 15



Factors that correlate with both project types

- more is better:
  - P90 Stability of Requirements
  - P100 Stability of Management Team
- less is beneficial
  - P93 Rigor of Requirements Review
  - P115 System Response Time

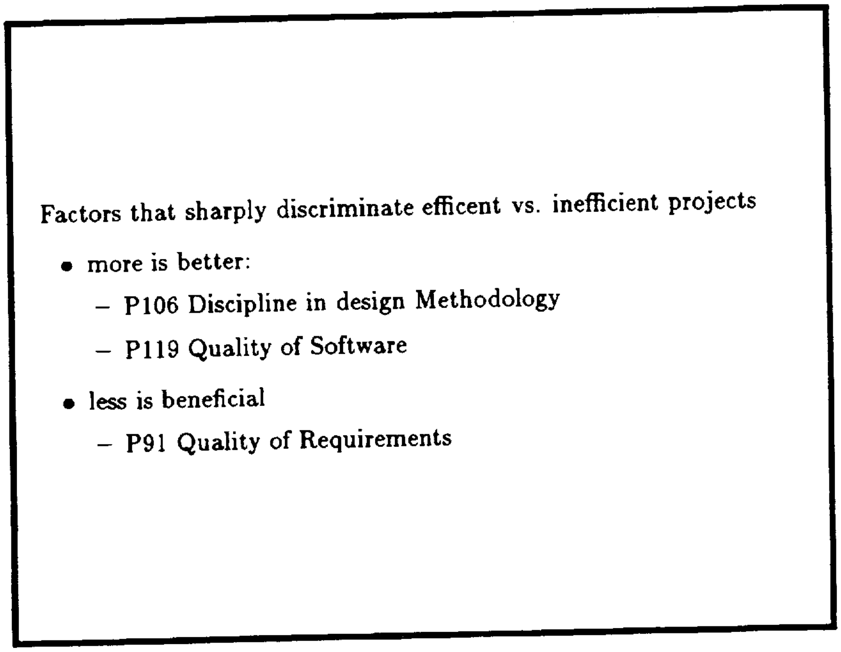
Slide 16



Factors that correlate with inefficient projects

- more is better:
  - P95 Development team application experience
- less is beneficial
  - P88 Problem Complexity
  - P105 Discipline in Requirements Methodology
  - P112 Access to Development System
  - P113 Ratio of Developers to Terminals

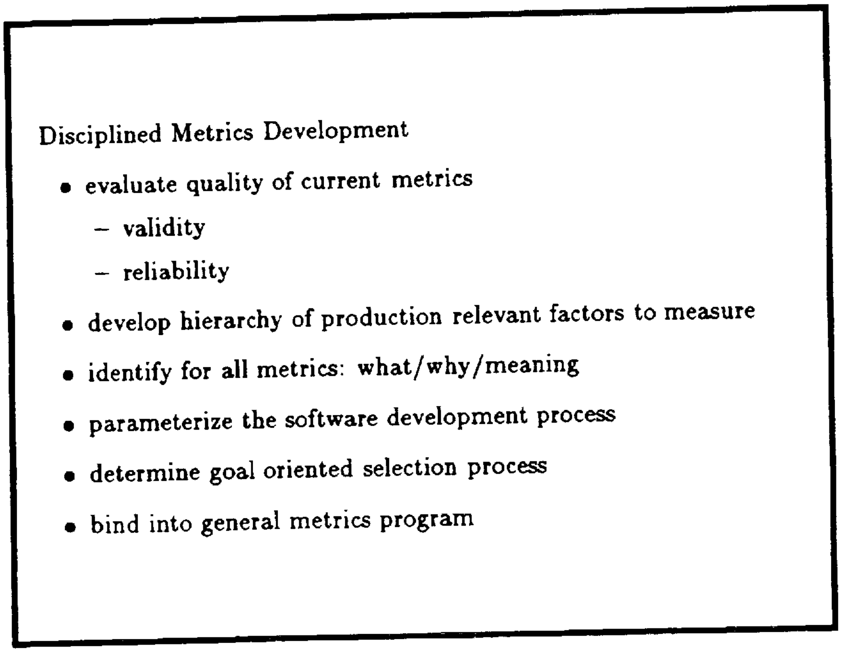
Slide 17



Factors that sharply discriminate efficent vs. inefficient projects

- more is better:
  - P106 Discipline in design Methodology
  - P119 Quality of Software
- less is beneficial
  - P91 Quality of Requirements

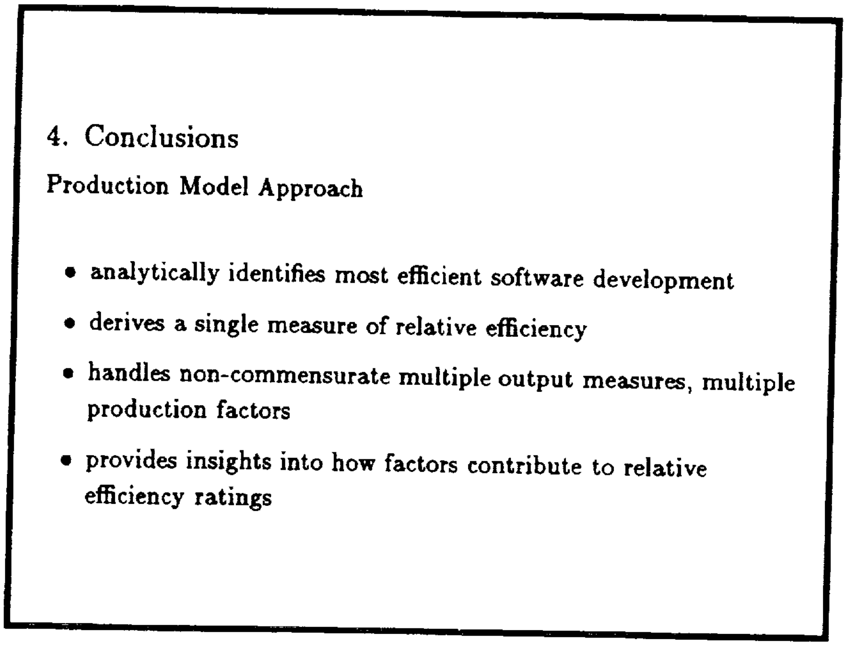
Slide 18



Disciplined Metrics Development

- evaluate quality of current metrics
  - validity
  - reliability
- develop hierarchy of production relevant factors to measure
- identify for all metrics: what/why/meaning
- parameterize the software development process
- determine goal oriented selection process
- bind into general metrics program

Slide 19



4. Conclusions

Production Model Approach

- analytically identifies most efficient software development
- derives a single measure of relative efficiency
- handles non-commensurate multiple output measures, multiple production factors
- provides insights into how factors contribute to relative efficiency ratings

Session 3: Technology Assessment

omit

Jon D. Valett, NASA/Goddard

Ray Madachy, Litton Data Systems

Maurice H. Blumberg, IBM Federal Systems Company

THE (MIS)USE OF SUBJECTIVE PROCESS MEASURES IN SOFTWARE ENGINEERING

Jon D. Valett

SOFTWARE ENGINEERING BRANCH
Code 552
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771

Steven E. Condon

COMPUTER SCIENCES CORPORATION
GreenTec II—10110 Aerospace Road
Lanham-Seabrook, Maryland 20706

WHAT ARE SUBJECTIVE PROCESS MEASURES?

A variety of measures are used in software engineering research to develop an understanding of the software process and product. These measures fall into three broad categories: quantitative, characteristics, and subjective. Quantitative measures are those to which a numerical value can be assigned, for example effort or lines of code (LOC). Characteristics describe the software process or product; they might include programming language or the type of application. While such factors do not provide a quantitative measurement of a process or product, they do help characterize them. Subjective measures (as defined in this study) are those that are based on the opinion or opinions of individuals; they are somewhat unique and difficult to quantify.

Capturing of subjective measure data typically involves development of some type of scale. For example, "team experience" is one of the subjective measures that were collected and studied by the Software Engineering Laboratory (SEL). Certainly, team experience could have an impact on the software process or product; actually measuring a team's experience, however, is not a strictly mathematical exercise. Simply adding up each team member's years of experience appears inadequate. In fact, most researchers would agree that "years" do not directly translate into "experience." Team experience must be defined subjectively and then a scale must be developed—e.g., high experience versus low experience; or high, medium, low experience; or a different or more granular scale. Using this type of scale, a particular team's overall experience can be compared with that of other teams in the development environment.

Defining, collecting, and scaling subjective measures is difficult. First, precise definitions of the measures must be established. Next, choices must be made about whose opinions will be solicited to constitute the data. Finally, care must be given to defining the right scale and level of granularity for measurement.

WHY DO SOFTWARE ENGINEERS NEED SUBJECTIVE MEASURES?

Despite the difficulties inherent in working with subjective measures, many researchers propose that the software process and product can not be characterized fully without them. Early work by Walston and Felix[1] used subjective data for characterizing software. Intermediate COCOMO[2] uses 16 subjective cost drivers for estimating software cost. These subjective measures range from "amount of experience with the development programming language" to "product complexity." For a given project, each of the 16 factors is rated and used to develop the basic cost estimate. The expectation is that inclusion of these factors will yield a more precise/accurate cost estimate. In fact, almost all cost models use some subjective factors.

In addition to cost modeling, software engineering researchers use subjective measures to help quantify other aspects of the software process. For example, they might try to determine if the team experience factor has any impact on productivity or reliability. In developing a reliability model, they might look at the quality of the team's code reading. Subjective measures can also be used in defining software domains. In this application, a subjective measure might be considered a defining factor in placing particular software in one domain versus another. Projects that use formal structured analysis, for example, may be in a different domain from those that use other methods.

This research examines the use of subjective measures in software engineering experimentation. In the sections that follow, this paper discusses the early experiences of the SEL collecting and applying subjective measure data, looks at refinements the SEL made to their collection and analysis process, and then reports on more recent SEL studies using subjective data. Some general recommendations are made for the collection and use of subjective data based on lessons learned in the SEL.

THE SEL AND SUBJECTIVE MEASURES

The SEL is a research organization that supports the Flight Dynamics Division of NASA/Goddard Space Flight Center. Its purpose is to investigate the effectiveness of software engineering technologies applied to the development of flight dynamics software.

The SEL collects a variety of data from application software projects for use in its research and experiments. These data include information on effort, size, computer resources, project characteristics, and a number of subjective measures. (For a complete description of the data collected see Reference 3.) The SEL began collecting subjective measures data in 1977. The primary goals for these data were to validate the models of other software engineering researchers and to fully characterize the SEL environment. As with many early SEL data collection efforts, an attempt was made in this case to collect every possible piece of data. On each project, over 300 individual subjective measures were collected.

For each measure, managers gave an opinion expressed as a rating based on a 0-5 scale. The data were not validated/cross-checked in any way before being stored in the SEL database. No one else examined the ratings given or tried to provide consistency across projects. Furthermore, the 0-5 ratings were not defined. Thus, for the same measure on the same project, two different individuals might have given different ratings. While this was somewhat minimized because there were very few people providing the data, the data were still inconsistent. Also, due to the lack of precise definitions for ratings, inconsistency was possible not only amongst data providers but also from project to project and from year to year. That is, because of changing perceptions, similar projects may have been given different ratings. Nevertheless, these data were used by the SEL in a variety of experiments, two of which are detailed below.

Early Uses of Subjective Measures

One early experiment using subjective measures was the development of a meta-model for software development resource expenditures.[4] The goal of the experiment was to develop a cost model that included subjective process measures. First, the subjective data from the SEL database were converted from the 0-5 scale to a binary (high/low) scale for use in the experiment. Second, the researchers selected one manager who was familiar with all the projects as a source for establishing consistency across the projects.

Using data from 17 projects, the researchers developed a baseline cost model that related effort to LOC. They examined the impact on cost of 71 different subjective measures to determine if any of them showed a significant relationship to the cost of the project. No significant correlation was found. The data proved to be too detailed to really determine if there was any impact. While the researchers were able to find some correlation between certain measures and cost, it was not consistent. The researchers then applied a grouping technique to the measures, converting the 71 measures into three groups. This allowed them to build new, broader-based subjective measures. Using these three measures they built a new cost model which they later confirmed against new projects that were similar to those in the data set.

Two main points emerge from reviewing this experiment:

- *Be wary of "looking for correlations."* While these researchers found some correlations when using the detailed data, they proved to be inconsistent. In almost any experiment using subjective data some correlations may exist, but they must be repeatable to be significant.
- *Collecting lots of data does not guarantee lots of results.* In this experiment the vast amount of data collected had to be converted to a much less detailed set.

In a second experiment using subjective measures, SEL researchers sought to determine the effect of modern programming practices (MPPs) on productivity and reliability.[5] Again, the subjective measures data in the SEL database were used after being converted to a binary scale and combined into groups. However, the grouping method used in this experiment differed from the method used in the previous experiment. Various subjective measures were combined with quantitative data to predefine MPPs such as structured coding and tool use. Then, analyzing data from 22 projects, the researchers tested the effects of MPPs on productivity and reliability. No correlation was shown on productivity, while quality of documentation, amount of quality assurance, and quality of code reading did have an impact on error rate. Unfortunately, these results were never confirmed over other data sets.

Major lessons on subjective measures from this study are:

- *Detailed subjective data probably are not useful.* Having over 300 different subjective measures actually proved to be less useful than having fewer, more general categories of subjective information.
- *To validate results using subjective data, confirm the results across multiple data sets.*

Refining SEL Subjective Data Collection

In 1987, the SEL (recognizing the difficulty with collecting and using over 300 detailed subjective measures) set out to significantly reduce the data set. Based on the experience of other researchers and the specific experience of the SEL, a new set of 36 measures was defined. These data continue to be collected today.

Subjective measure data are now provided by project leads. At the end of each project, the project lead completes a questionnaire that uses a 1-5 scale. (The questionnaire is included as an appendix.) The opinions of the project lead are presumed to be accurate; no other validation or cross-checking of the data is done. This data collection policy still allows bias and potential inconsistency within the data as people with different perspectives and experiences might give the same project different scores. Two experiments using the newer subjective data are discussed below.

Recent Experiences with Subjective Measures

Recently, a study was conducted in which the 36 subjective measures were applied to a basic cost model. This was done as part of a larger effort to build a specific cost model for the SEL environment.[6] In this study the researchers used the measures as they were recorded in the SEL database. They developed a basic cost model and then attempted to improve that model by adding various subjective measures. On the initial data sets used, some of the measures did appear to improve the cost models, but when the researchers tried to validate the models using different data sets (from similar projects) they were unable to duplicate the results. In fact, they found similar improvements in the models when they substituted random data for the actual subjective measures data. Given these results, the researchers concluded that the current subjective data should not be used as a factor in projecting cost.

Two lessons learned from this experience:

- *Collecting data on a 1-5 scale is probably not optimal.* Distinguishing each rating, for example a "2" versus a "3," is difficult. In the past, when these data have been used in analysis they have been converted to a binary scale. The scale should be reduced either when the data are collected or when they are used.
- *Results should be confirmed over multiple data sets.* This has been pointed out before, but it bears repeating. In too many instances researchers have come to conclusions based on one set of projects without checking out the results on other similar projects.

Another study was conducted (specifically for this report) with the goal of determining the impact of subjective measures on effort, errors, and changes. Data were converted to a binary scale. Also, the analytic method used assumed that the 36 measures were not independent. (The previous study did not address the dependency of the data.) For any set of projects, a linear model was built relating the size of a project to a particular measure, such as changes. Then a set of subjective measures that may have had an impact on the chosen measure was identified. From that set, the factors that were most likely to have had an impact and those that best represented the dependent set of measures were added to an enhanced linear model. Attempts to validate these models against multiple similar data sets showed little or no consistency.

Based on this study and the others discussed, it appears that even the conservative use (i.e., using a binary scale and incorporating data dependency) of the subjective measures data collected by the SEL is of questionable value. While previous analyses of the data showed some promise, recent experiences have been less successful.

MISUSES AND USES OF SUBJECTIVE MEASURES

Based on these findings, SEL researchers have questioned the value of collecting these data. Although the data may not be viable for rigid statistical analyses, they can be important tools for environment characterization and research planning purposes. When working with subjective measures, the following guidelines should be considered:

- *Be cognizant of the data collection mechanism and the extent to which the data are validated.* Make no assumptions concerning the accuracy and validity of the data.
- *When defining subjective measures for collection, less is usually better.* Collecting a wide variety of data without a plan for their use is pointless.
- *Use subjective measures to spot trends and set goals for more detailed experiments.* General subjective measures can be a good place to start when setting goals for research. This is probably the best way to use loosely defined, nonvalidated subjective measures such as those collected by the SEL.

Given the somewhat limited usefulness of the SEL's subjective measure data, the SEL might be expected to abandon collection of subjective data. Subjective information, however, is important for understanding an environment and it provides a context for data analysis. When designing experiments or studies, a researcher needs to examine subjective information about a project to decide if that project is appropriate for inclusion in a particular study. That information might, however, be more likely found in project documents (e.g., lessons learned reports) than in ranked questionnaire responses.

Rather than abandon subjective measure data collection, the SEL needs to define a set of subjective measures that accurately captures the critical elements of the local environment. From there, a set of goals for the subjective measures must be identified and a set of questions generated that precisely defines the measures for the local environment. The last step would be to develop a methodology for collecting and validating the data. If such steps are taken, the validity of the subjective measures data could be improved and their usefulness in the SEL's ongoing process improvement program could be reexamined.

REFERENCES

1. Walston, C., and C. Felix, "A Method of Programming Measurement and Estimation," *IBM Systems Journal* 16, Number 1, 1977

2. Boehm, B. W., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981

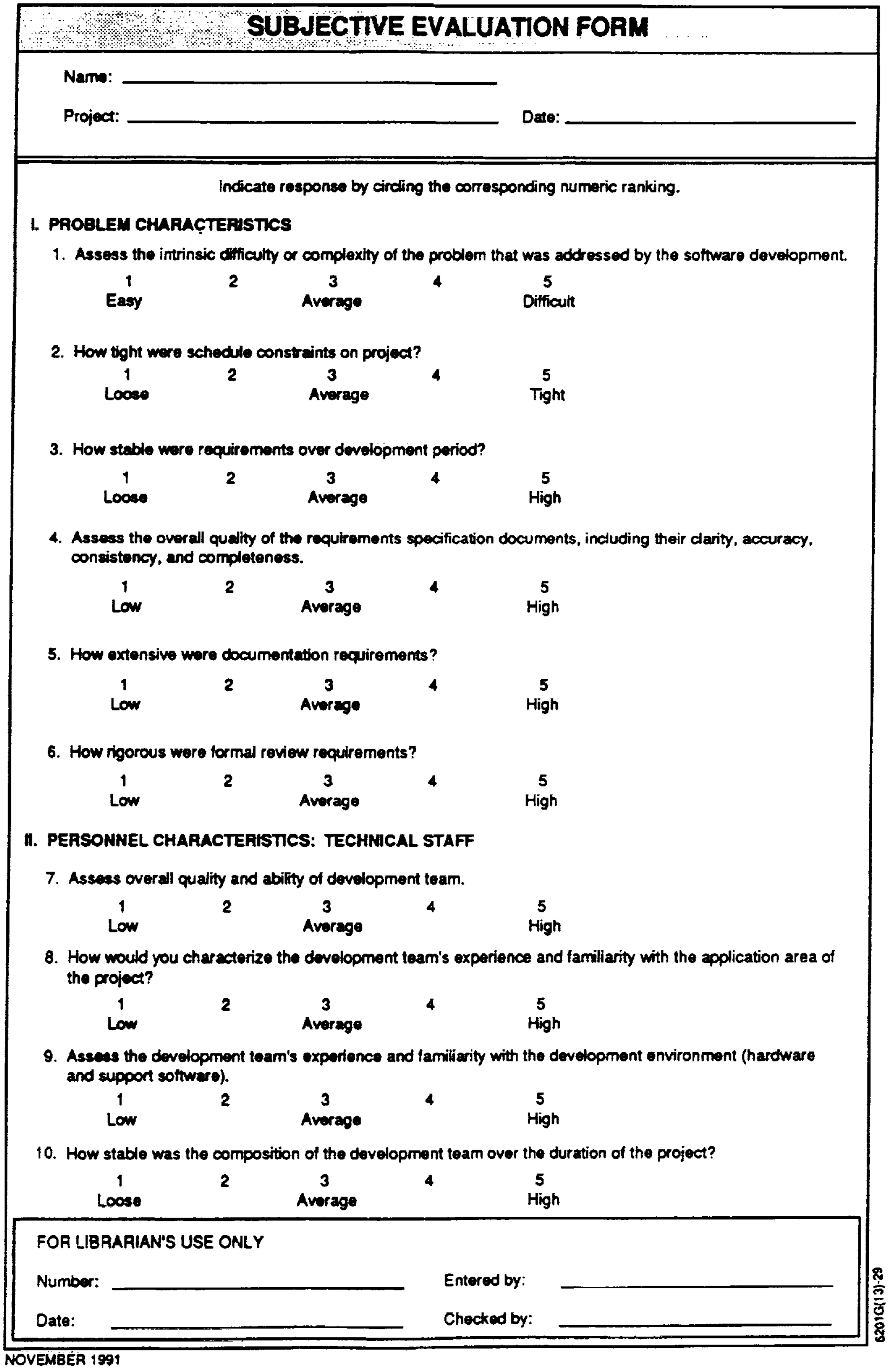
3. Heller, G., J. Valett, and M. Wild, *Data Collection Procedures for the Software Engineering Laboratory (SEL) Database*, SEL-92-002, Software Engineering Laboratory, Greenbelt, Maryland, 1992

4. Basili, V. R., and J. W. Bailey, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*, IEEE Computer Society Press, New York, New York, 1981

5. Card, D. N., F. E. McGarry, and G. T. Page, "Evaluating Software Engineering Technologies," *IEEE Transactions on Software Engineering*, July 1987

6. Condon, S., et al., *Cost and Schedule Estimation Study Report*, SEL-93-002, Software Engineering Laboratory, Greenbelt, Maryland, 1993

APPENDIX—SEL SUBJECTIVE DATA COLLECTION QUESTIONNAIRE



SUBJECTIVE EVALUATION FORM

Name: ____________________

Project: ____________________ Date: ____________________

Indicate response by circling the corresponding numeric ranking.

I. PROBLEM CHARACTERISTICS

1. Assess the intrinsic difficulty or complexity of the problem that was addressed by the software development.

1	2	3	4	5
Easy		Average		Difficult

2. How tight were schedule constraints on project?

1	2	3	4	5
Loose		Average		Tight

3. How stable were requirements over development period?

1	2	3	4	5
Loose		Average		High

4. Assess the overall quality of the requirements specification documents, including their clarity, accuracy, consistency, and completeness.

1	2	3	4	5
Low		Average		High

5. How extensive were documentation requirements?

1	2	3	4	5
Low		Average		High

6. How rigorous were formal review requirements?

1	2	3	4	5
Low		Average		High

II. PERSONNEL CHARACTERISTICS: TECHNICAL STAFF

7. Assess overall quality and ability of development team.

1	2	3	4	5
Low		Average		High

8. How would you characterize the development team's experience and familiarity with the application area of the project?

1	2	3	4	5
Low		Average		High

9. Assess the development team's experience and familiarity with the development environment (hardware and support software).

1	2	3	4	5
Low		Average		High

10. How stable was the composition of the development team over the duration of the project?

1	2	3	4	5
Loose		Average		High

FOR LIBRARIAN'S USE ONLY

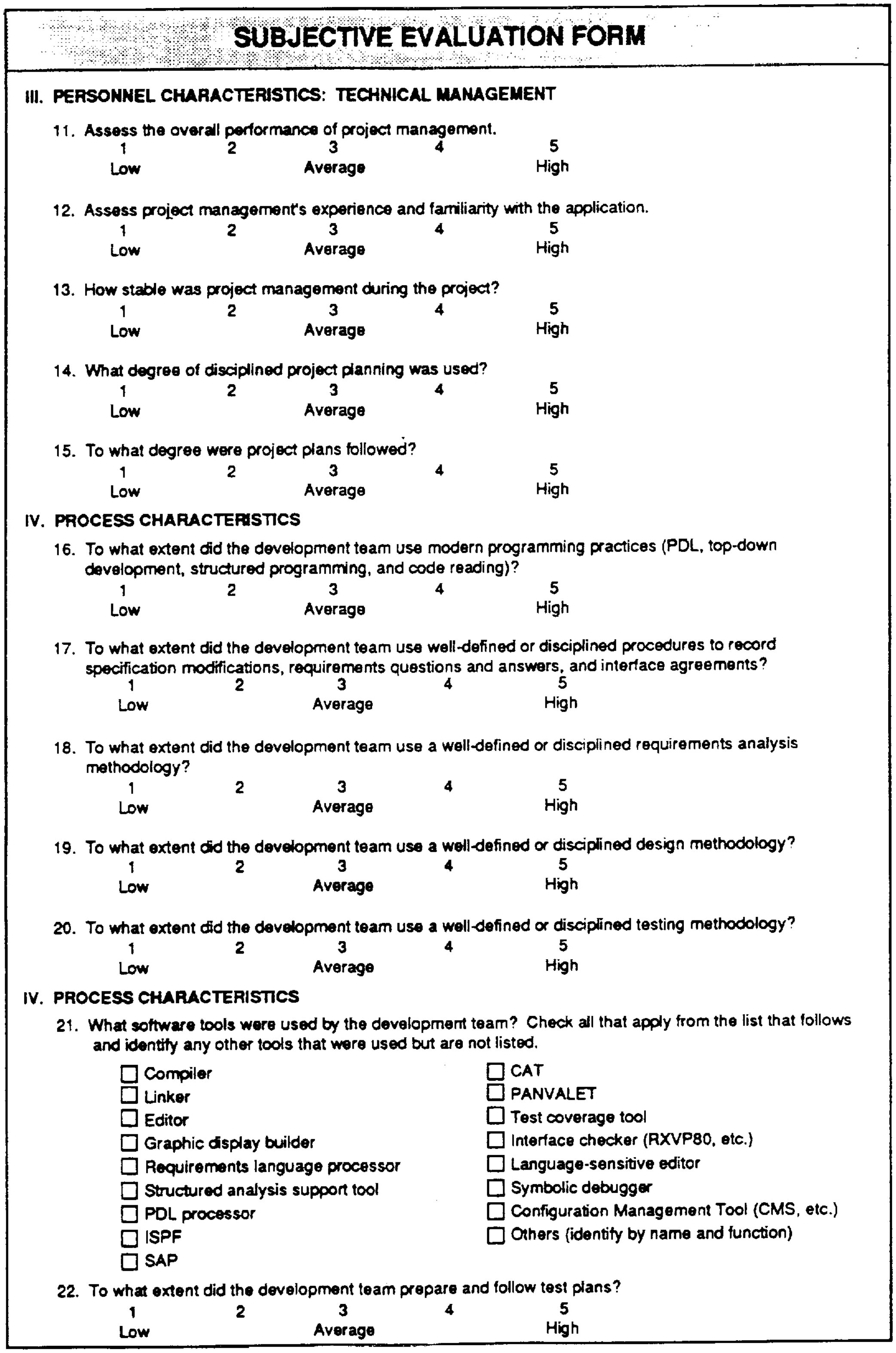
Number: ____________________ Entered by: ____________________

Date: ____________________ Checked by: ____________________

8201G(13)-29

NOVEMBER 1991

APPENDIX—SEL SUBJECTIVE DATA COLLECTION QUESTIONNAIRE



SUBJECTIVE EVALUATION FORM

III. PERSONNEL CHARACTERISTICS: TECHNICAL MANAGEMENT

11. Assess the overall performance of project management.
1 2 3 4 5
Low Average High

12. Assess project management's experience and familiarity with the application.
1 2 3 4 5
Low Average High

13. How stable was project management during the project?
1 2 3 4 5
Low Average High

14. What degree of disciplined project planning was used?
1 2 3 4 5
Low Average High

15. To what degree were project plans followed?
1 2 3 4 5
Low Average High

IV. PROCESS CHARACTERISTICS

16. To what extent did the development team use modern programming practices (PDL, top-down development, structured programming, and code reading)?
1 2 3 4 5
Low Average High

17. To what extent did the development team use well-defined or disciplined procedures to record specification modifications, requirements questions and answers, and interface agreements?
1 2 3 4 5
Low Average High

18. To what extent did the development team use a well-defined or disciplined requirements analysis methodology?
1 2 3 4 5
Low Average High

19. To what extent did the development team use a well-defined or disciplined design methodology?
1 2 3 4 5
Low Average High

20. To what extent did the development team use a well-defined or disciplined testing methodology?
1 2 3 4 5
Low Average High

IV. PROCESS CHARACTERISTICS

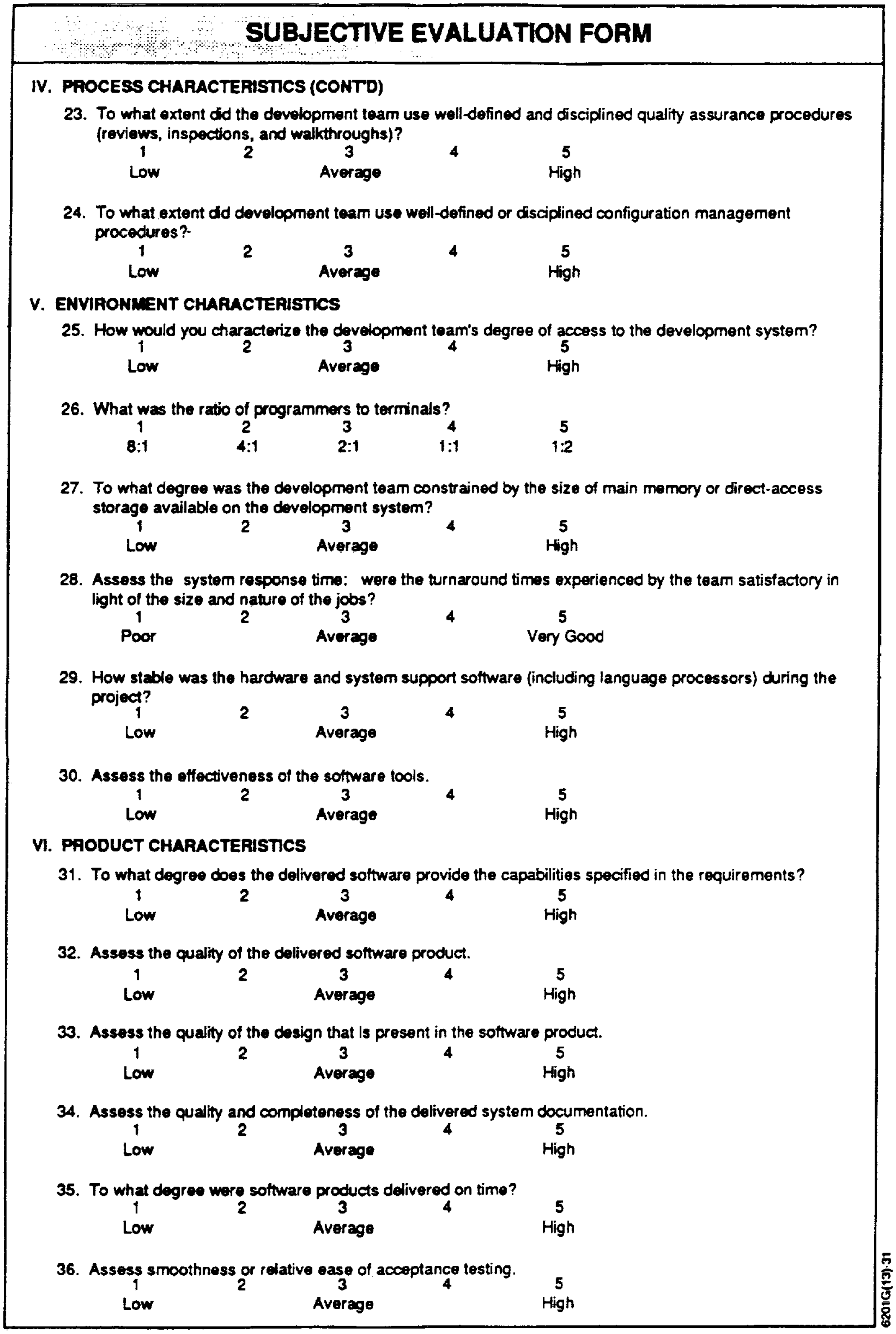
21. What software tools were used by the development team? Check all that apply from the list that follows and identify any other tools that were used but are not listed.

□ Compiler
□ Linker
□ Editor
□ Graphic display builder
□ Requirements language processor
□ Structured analysis support tool
□ PDL processor
□ ISPF
□ SAP
□ CAT
□ PANVALET
□ Test coverage tool
□ Interface checker (RXVP80, etc.)
□ Language-sensitive editor
□ Symbolic debugger
□ Configuration Management Tool (CMS, etc.)
□ Others (identify by name and function)

22. To what extent did the development team prepare and follow test plans?
1 2 3 4 5
Low Average High

8201G(13)-30

APPENDIX—SEL SUBJECTIVE DATA COLLECTION QUESTIONNAIRE



SUBJECTIVE EVALUATION FORM

IV. PROCESS CHARACTERISTICS (CONT'D)

23. To what extent did the development team use well-defined and disciplined quality assurance procedures (reviews, inspections, and walkthroughs)?

1	2	3	4	5
Low		Average		High

24. To what extent did development team use well-defined or disciplined configuration management procedures?-

1	2	3	4	5
Low		Average		High

V. ENVIRONMENT CHARACTERISTICS

25. How would you characterize the development team's degree of access to the development system?

1	2	3	4	5
Low		Average		High

26. What was the ratio of programmers to terminals?

1	2	3	4	5
8:1	4:1	2:1	1:1	1:2

27. To what degree was the development team constrained by the size of main memory or direct-access storage available on the development system?

1	2	3	4	5
Low		Average		High

28. Assess the system response time: were the turnaround times experienced by the team satisfactory in light of the size and nature of the jobs?

1	2	3	4	5
Poor		Average		Very Good

29. How stable was the hardware and system support software (including language processors) during the project?

1	2	3	4	5
Low		Average		High

30. Assess the effectiveness of the software tools.

1	2	3	4	5
Low		Average		High

VI. PRODUCT CHARACTERISTICS

31. To what degree does the delivered software provide the capabilities specified in the requirements?

1	2	3	4	5
Low		Average		High

32. Assess the quality of the delivered software product.

1	2	3	4	5
Low		Average		High

33. Assess the quality of the design that is present in the software product.

1	2	3	4	5
Low		Average		High

34. Assess the quality and completeness of the delivered system documentation.

1	2	3	4	5
Low		Average		High

35. To what degree were software products delivered on time?

1	2	3	4	5
Low		Average		High

36. Assess smoothness or relative ease of acceptance testing.

1	2	3	4	5
Low		Average		High

6201G(13)-31

The (Mis)use of Subjective Process Measures in Software Engineering

Jon. D. Valett
NASA/GSFC

Steve Condon
CSC

G218 001

Categories of Measurement Data

Quantitative	Characteristics	Subjective
- Effort	- Programming Language	- Team experience
- LOC	- Platform	- Requirements stability
- Computer use	- Application	- Degree of MPP
●	●	●
●	●	●
●	●	●

Subjective Measures –
those that are based on the opinion of individuals

How Should Subjective Measures be Used in Software Engineering?

G218.002

Need for Subjective Measures

- Help to Quantify the Software Process
  - Does team experience impact productivity?
  - Do Modern Programming Practices (MPPs) impact the development process and product?
- Improve Models of Software Process and Product
  - Error Rate = X * Developed LOC - Y * Quality of Code Reading
  - Intermediate COCOMO
- Define Software Domains
  - Are projects that use structured analysis different from those that don't?

G218.003

Subjective Measures

- There are many subjective measures
  e.g.
  - Team experience
  - Management stability
  - Machine availability
  - Quality of tool set
  - Schedule constraint
  - Product complexity

  etc.
- There have been many proposed uses -
  - Walston and Felix
  - COCOMO
  - Other Cost Models
  - Domain Analysis

G218.004

The SEL and Subjective Measures

Beginning in 1977 the Software Engineering Laboratory (SEL) began collecting subjective measures

Philosphy -

Validate models of other researchers

Fully characterize the environment

Sumary

What Data?	Collect Everything (over 300 individual measures)
Who Provides?	Managers rate
How Collected?	After project completion Use 0-5 scale
How Clarified/Validated	None

G218.005

Use of Subjective Measures

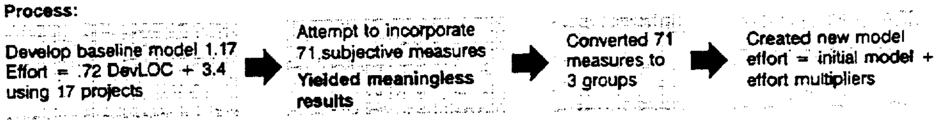
"The Meta-Model for Software Development Resource Expenditures*"

Goal:
Develop a cost model that incorporates subjective process measures

Subjective Measures:

- Converted to binary scale
- Validated measures using 1 manager as source
- Converted detailed data into three groups

Process:



Develop baseline model 1.17
Effort = .72 DevLOC + 3.4
using 17 projects

Attempt to incorporate
71 subjective measures
Yielded meaningless results

Converted 71
measures to
3 groups

Created new model
effort = initial model +
effort multipliers

Result:

Model confirmed using new projects similar to those in data set

Lessons:

- If you look hard enough you may find some correlations
- A lot of data does not generate a lot of results

* Bailey and Basili 1981

G218.006

Use of Subjective Measures

"Evaluating Software Engineering Technologies*"

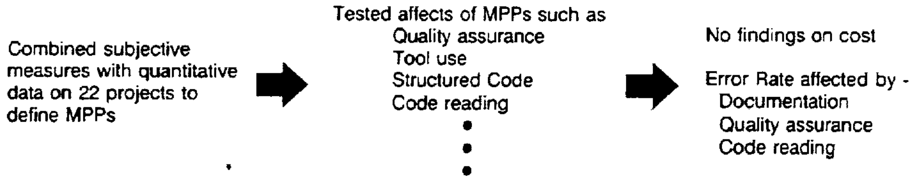
Goal:

Do MPPs affect productivity and reliability?

Subjective Measures:

- Converted to binary scale
- Combined data into groups

Process:



Combined subjective measures with quantitative data on 22 projects to define MPPs

Tested affects of MPPs such as
Quality assurance
Tool use
Structured Code
Code reading
•
•
•

No findings on cost

Error Rate affected by -
Documentation
Quality assurance
Code reading

Result:

Not confirmed over other data sets (within the same domain) - conclusions questionable

Lessons:

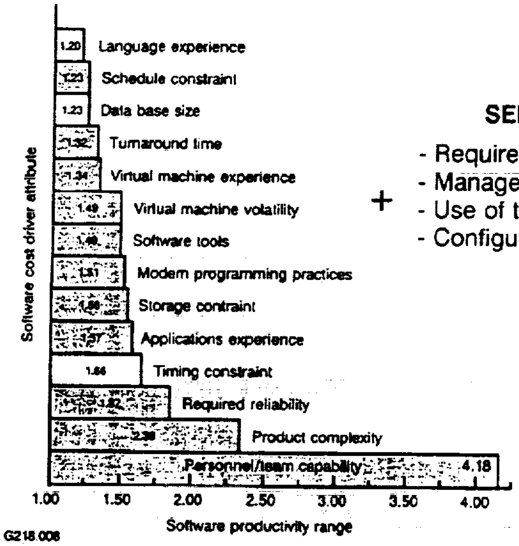
- Confirm results over multiple similar data sets
- A lot of data does not generate a lot of results

[*] Carc, McGarry, Page 1987

G218.007

Reducing the Measure Set

Boehm's Software Engineering Economics



▨ - Relevant to the SEL

G218.008

+

SEL Experience

- Requirements Stability
- Management Experience
- Use of test plans
- Configuration management

•
•
•

= 36 measures

Current Subjective Measures (1987 - Present)

Philosphy -

Determine the impact of key measures on the software process and product

Characterize the environment

Sumary

What Data?	36 General subjective measures
Who Provides?	Project leads
How Collected?	After project completion Use 1-5 scale Survey form
How Clarified/Validated	None

The SEL continues to collect high level subjective measures

G218.009

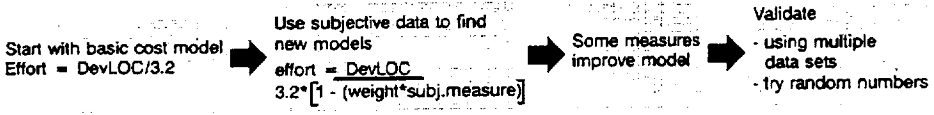
Use of Subjective Measures

"Cost Estimation Study"

Goal:
Improve a basic cost model using subjective measures

Subjective Measures:
- Used the 1-5 ratings
- Used multiple data sets

Process:



Start with basic cost model
Effort = DevLOC/3.2

Use subjective data to find new models

$$\text{effort} = \frac{\text{DevLOC}}{3.2^{*}[1 - (\text{weight}^{*}\text{subj.measure})]}$$

Some measures improve model

Validate
- using multiple data sets
- try random numbers

Results:
- Enhanced models inconsistent over multiple data sets
- Random numbers improved models as well as the real data

Lessons:
- Tend toward conservative use of measures (1-5 scale too detailed)
- Carefully evaluate all results

* Condon, Regardie 1993

G218.010

Use of Subjective Measures

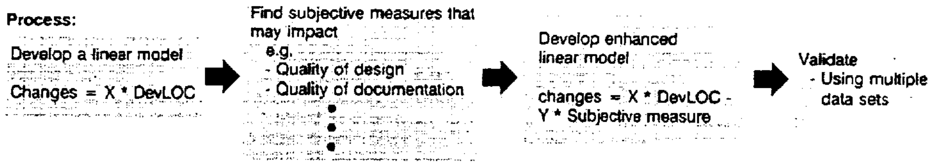
"Impact of Subjective Measures on Effort, Errors, and Changes"

Goal:

Are there subjective measures that impact effort, errors, and changes?

Subjective Measures:

- Converted to binary scale
- Assumed dependency in the data
- Used multiple data sets



Process:

Develop a linear model

Changes = X * DevLOC

Find subjective measures that may impact

e.g.

- Quality of design
- Quality of documentation

•
•
•

Develop enhanced linear model

changes = X * DevLOC - Y * Subjective measure

Validate

- Using multiple data sets

Result:

Little or no consistency found among data sets

Lessons:

- Even conservative use of data is questionable
- The 36 measures are not independent

G218.011

Misuses of Subjective Measures

- Don't search for correlations, because you will find at least one
- Don't collect too much data without understanding how to use it
- Don't go beyond the validity and consistency of your data
- Don't rely on on-line data - except to spot trends/set goals

The measures contain no miracle answers. They are only one tool.

G218.012

Subjective Measures ≠ Subjective Information

- Subjective Information Provides Context for Analysis
  - Lessons learned documents
  - Project annotations
- Set Goals for Subjective Information
- Subjective Information Transformed into Subjective Measures by
  - Local definitions
  - Using consistent data collection methods

Subjective information is critical to understanding an environment, but don't think it is easy

G218.013

ANALYSIS OF A SUCCESSFUL INSPECTION PROGRAM

S8-61
12690
p- 23

N94- 36492

Ray Madachy
Linda Little
Sylvia Fan

Software Engineering Process Group
Litton Data Systems
Agoura Hills, CA

ABSTRACT

Litton Data Systems has institutionalized the inspection process, and achieved dramatic results in terms of defect prevention and cost savings thus far. Additionally, several findings have been gleaned from an analysis to optimize the process. Over 300 inspections have been performed over the last two years on many types of documents, and this paper describes some quantitative results to-date from the initial "champion" project.

BACKGROUND

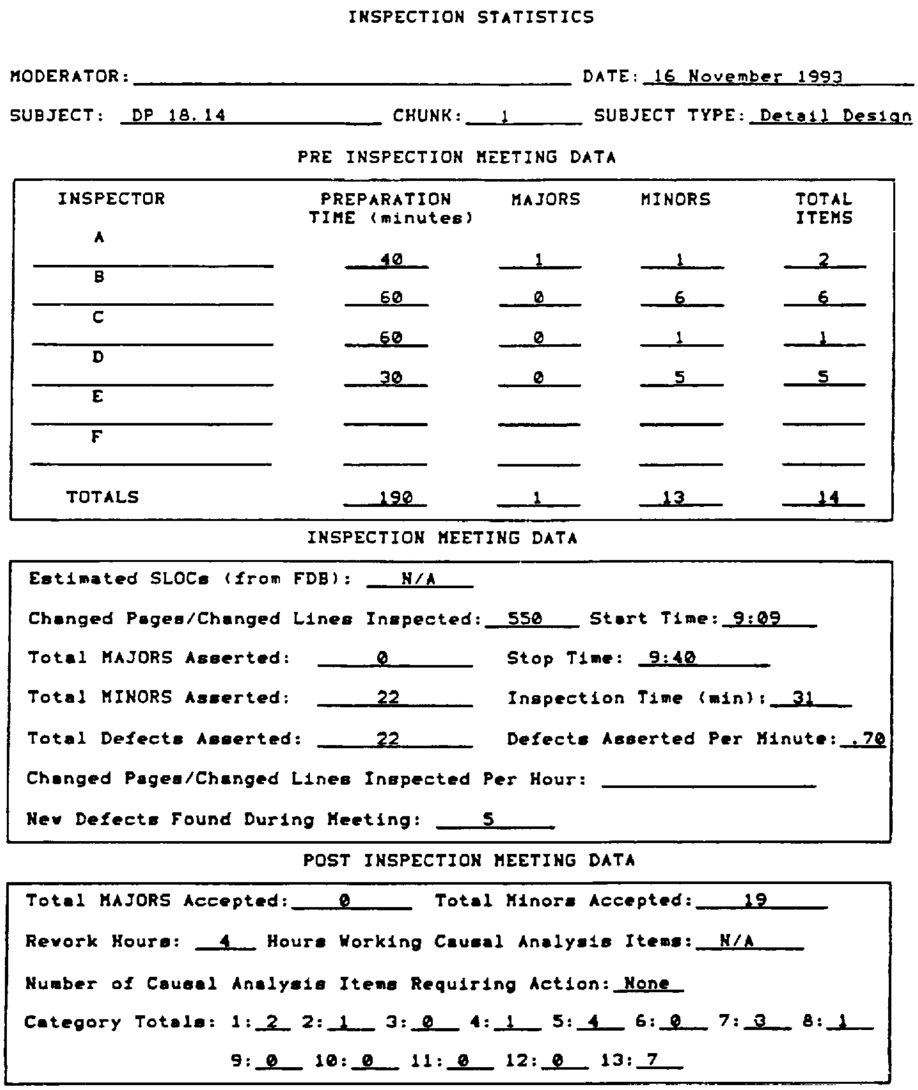
Litton was first trained in inspections by Tom Gilb in 1989. His method differs from Fagan's [Gilb 88], and Litton has subsequently modified Gilb's method for in-house. The success of our program owes much to strong executive support. Inspections are now the cornerstone of our peer review process.

Over 400 software personnel have been trained in inspections, and inspections are now being used on four major development programs. Our software director set project goals to save at least 50% of integration effort by spending more effort during design and coding for inspections. Thus far, we appear to be achieving this goal.

Unique properties of the Litton inspection process include no "reader" role, no discussion on defect category during inspection, a routing process for inspection results, no time limit on causal analysis and the use of a Software Engineering Process Group (SEPG) Peer Review Coordinator. A standard reporting form, as shown in Table 1, has been devised for collecting the inspection data.

Though project management has collected some high-level inspection statistics, the SEPG instituted an inspection database as part of its metrics program to evaluate process improvement. Data from the form in Table 1 goes into the database, and is regularly entered at the end of each week. The database was used for this analysis, and validated against high-level project management data. The provision on the form for defect categories supporting causal analysis is a recent addition, so little data has been collected for defect category analysis up to this point. The following sections describe some results-to-date of our analysis of the inspection data.

Table 1: Typical Data Sheet



INSPECTION STATISTICS

MODERATOR: ______ DATE: 16 November 1993

SUBJECT: DP 18.14 CHUNK: 1 SUBJECT TYPE: Detail Design

PRE INSPECTION MEETING DATA

INSPECTOR	PREPARATION TIME (minutes)	MAJORS	MINORS	TOTAL ITEMS
A	40	1	1	2
B	60	0	6	6
C	60	0	1	1
D	30	0	5	5
E				
F				
TOTALS	190	1	13	14

INSPECTION MEETING DATA

Estimated SLOCs (from FDB): N/A

Changed Pages/Changed Lines Inspected: 550 Start Time: 9:09

Total MAJORS Asserted: 0 Stop Time: 9:40

Total MINORS Asserted: 22 Inspection Time (min): 31

Total Defects Asserted: 22 Defects Asserted Per Minute: .70

Changed Pages/Changed Lines Inspected Per Hour: ______

New Defects Found During Meeting: 5

POST INSPECTION MEETING DATA

Total MAJORS Accepted: 0 Total Minors Accepted: 19

Rework Hours: 4 Hours Working Causal Analysis Items: N/A

Number of Causal Analysis Items Requiring Action: None

Category Totals: 1: 2 2: 1 3: 0 4: 1 5: 4 6: 0 7: 3 8: 1
9: 0 10: 0 11: 0 12: 0 13: 7

ANALYSIS

This analysis concerns both optimization of the inspection process, as well as performing a cost/benefit analysis to determine how much extra effort is used during design and coding for inspections and how much is saved during testing and integration. This effect on project effort is shown in Figure 1 from [Fagan 86].

2

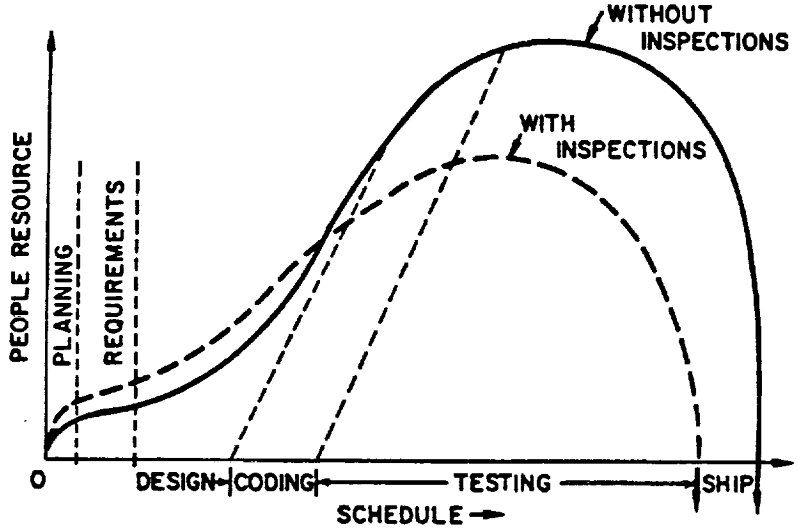


Figure 1: Effect of Inspections on Project Effort (from [Fagan 86])

The following formulations are used in this analysis:

defects found = items from preparation + new items

inspection effort = preparation effort + meeting effort + rework effort

defect removal effectiveness = defects found / inspection effort

finding rate = defects / meeting time

inspection rate = inspected pages / meeting time

meeting effort = (meeting time) * (# personnel involved).

Preparation effort is the total effort for all inspectors. A major defect is defined as an error that would lead to a trouble report during testing and integration. A new item is one found during the inspection meeting that was not identified by any inspectors during pre-inspection preparation. We decided to separate new items discovered at the inspection meeting from defects noted during preparation, as we have observed that certain practices increase the new item finding rate and wish to investigate further.

Several types of documents are inspected: requirements (both requirements description and requirements analysis); design (top-level and detailed design); code; and change requests. Summary statistics are shown in Table 2. The total inspection effort was distributed as follows: preparation effort - 27%, inspection meeting effort - 39% and rework effort - 34%. The last column in Table 2 represents the defect removal effectiveness. As seen, the effectiveness decreases for later documents.

Table 2: Summary Statistics

Subject Type	Total Defects	Total Majors	Inspection Effort	# Pages	LOC Inspected	Major Defects/ Inspection Effort
REQUIREMENT DESCRIPTION	460	72	78	179	0	.923
REQUIREMENT ANALYSIS	2165	177	483	1065	0	.366
HIGH LEVEL DESIGN	2199	188	655	1592	0	.287
DETAILED DESIGN	1550	127	610	1387	19007	.208
Subtotal	6374	564	1826	4223	19007	.309
CODE	4272	432	1742	5047	149361	.248
CHANGE REQUEST	814	27	309	1579	0	.087
Grand total	11460	1023	3877	10849	168368	.264

When the defect density for these document types are ordered by activity, the results show that the defects steadily decrease since the predecessor artifacts were previously inspected. This is shown in Figure 2, overlayed with similar results from JPL [Kelly-Sherif 90]. The trend seems to corroborate the previous results. Code is not shown because of inconsistencies in reporting the document size. These results strongly support the practice of inspecting documents early as possible in the life cycle.

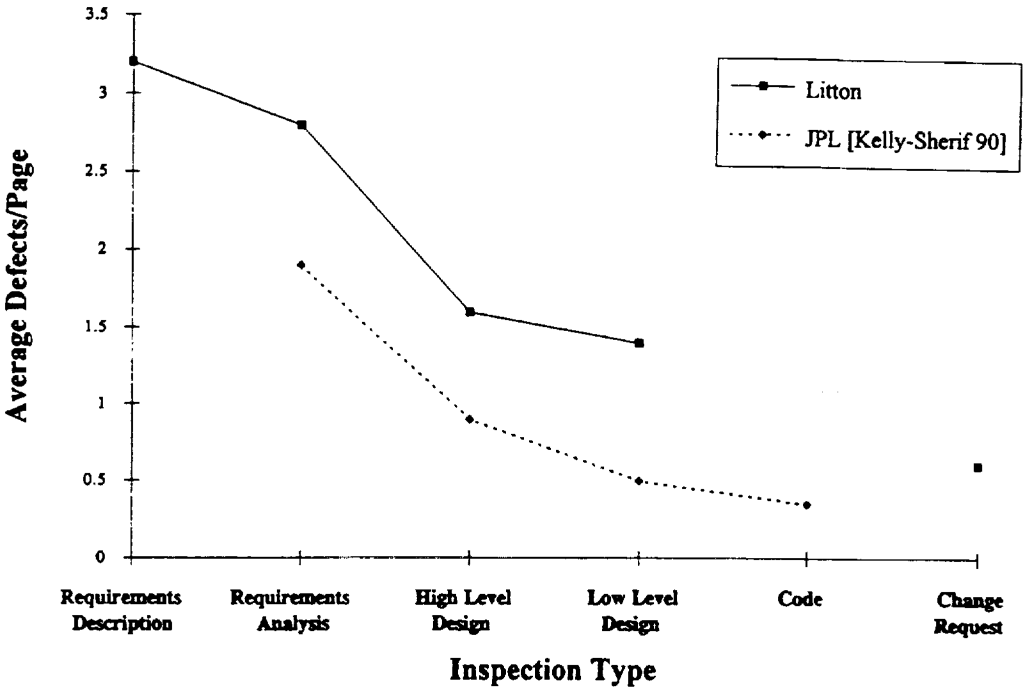


Figure 2: Defect Density per Subject Type

Project Effort for Inspections

We tracked the inspection effort as a portion of the total software development effort over the last year. The effects of schedule pressure were seen on inspection data, much as it is observed for staff coding and integration efforts before a "drop dead" date. This trend is shown in Figure 3, where the percent of project effort dedicated to inspections is plotted. The monthly inspection effort profile shows extreme peaks right before two Technical Interface Meetings where the customer evaluates the inspected documents, and right before a Preliminary Design Review with the customer. For this time period of regular inspections, an average of 2.9% of effort was spent on inspections. Both preparation time and inspection time increased during the peaks, but preparation time increased much more severely. The relatively small increase in actual inspection time indicates that the meeting process remained under control, instead of moderators drastically slowing the pace to find more defects. These dynamic effects on effort due to schedule pressure affect the long term averages and short-term project behavior, and should be kept in mind when planning effort or evaluating project trends since process stability is affected.

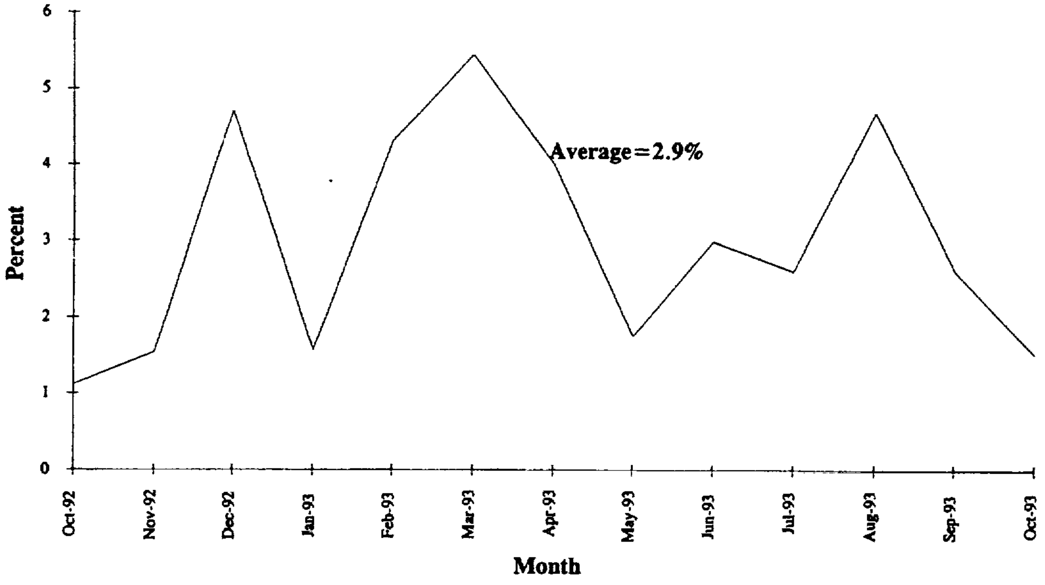


Figure 3: Percent of Project Effort for Inspections

Based on statistics on the inspection effort and knowledge about the process, a bottoms-up inspection costing algorithm has been devised. It identifies effort for pre-inspection, inspection and post-inspection activities based on the type and length of the inspected document(s). The algorithm is being included in a cost model used in the Division.

Return on Investment

The following return on investment (ROI) method of tracking inspection success calculates the difference of testing time saved and inspection effort for each meeting [Grady 92], [Rodriguez 91]. It uses the formulation

ROI= (found defects) * (average effort to fix defect in test) - inspection effort for each inspection meeting, using major defects only. The rationale for equating test defects with design defects follows from the previous definition of a major defect. At Litton, our historical data on the product line shows an average of 17.6 person-hours is spent to fix defects during testing. Using this value, the ROI for each inspection is shown in Figure 4. Overall, the total return from these inspections has been 14,210 person-hours of effort, with an average return of 63.4 person-hours per inspection. Out of 223 inspections, 139 have provided savings.

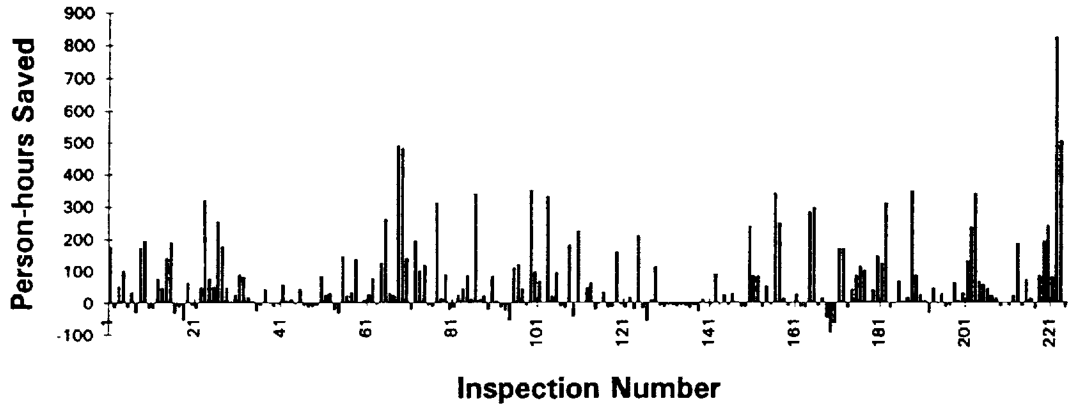


Figure 4: Return on Investment per Inspection

Statistics have been kept for several years on the number of trouble reports encountered during integration and the associated costs to fix them for this particular product line. When comparing trouble report data before and after inspections were introduced, there is a 76% reduction in trouble report density. This appears to be on the high end of reported results for defect reduction. Using the historical data on average efforts to fix inspection defects and trouble reports, about 573 labor-hours per KSLOC have been saved.

Process Control

Figures 5-7 show control charts for defect finding rate, design document inspection rate and code inspection rate. The bands shown on them represent the average values plus or minus a standard deviation for the upper and lower control limits. The overall items/minute for this project is apparently on the low end of the industry standard. The variances of inspection rates are higher relative to the variances of defect finding efficiency due to aforementioned dynamic schedule effects and other phenomena such as "process tweaking" and new personnel.

In Figure 5, it appears that the defect finding rate seems to have come down since the beginning of the program. This trend of project evolution could be due to the earlier documents having higher defect densities per Figure 2. In Figure 6, note that there seems to be a relatively sudden ending to the activity near 5/93. This corresponds to the date when coding started in earnest, and much design documentation was completed at that time.

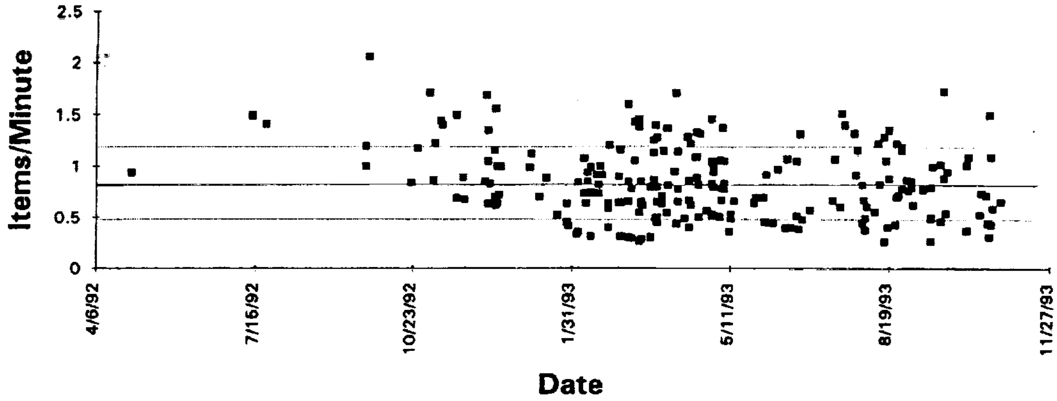


Figure 5: Defect Finding Rate Control Chart

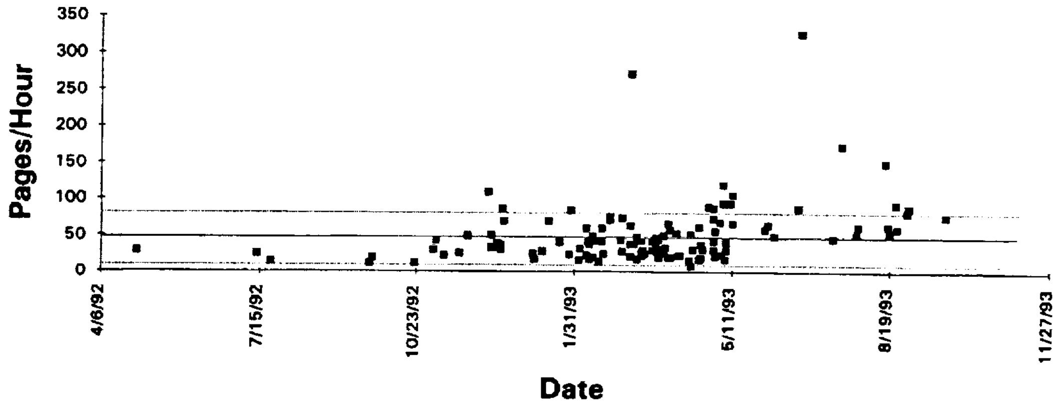


Figure 6: Document Inspection Rate Control Chart

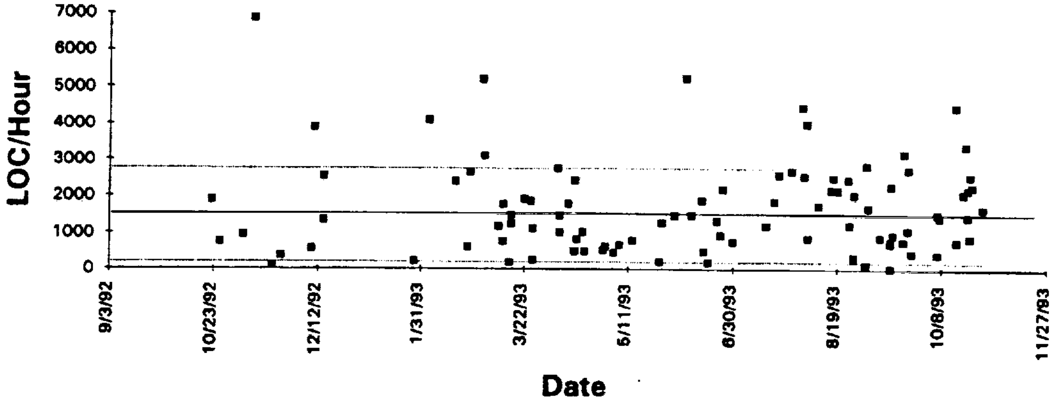


Figure 7: Code Inspection Rate Control Chart

When analyzing the data for adherence to process control limits for inspection rate and item finding rate, several outlying data points were identified. Upon further investigation, it was seen that there was a single moderator who was not particularly well-suited to the task. This moderator had been previously identified as one who rushed through the documents too fast, and the analysis confirmed that perception.

Along these same lines of inquiry, we wanted to see if outlying moderators could be detected by looking at individual performance. Figure 8 shows the average items found per minute for all moderators, and they all are in the same approximate range. This depiction showed some disparate ranges between moderators earlier in the program, thus we feel that the process has stabilized among moderators over time. This provides confidence that the process is relatively independent of individual moderators used and shows the benefits of good training.

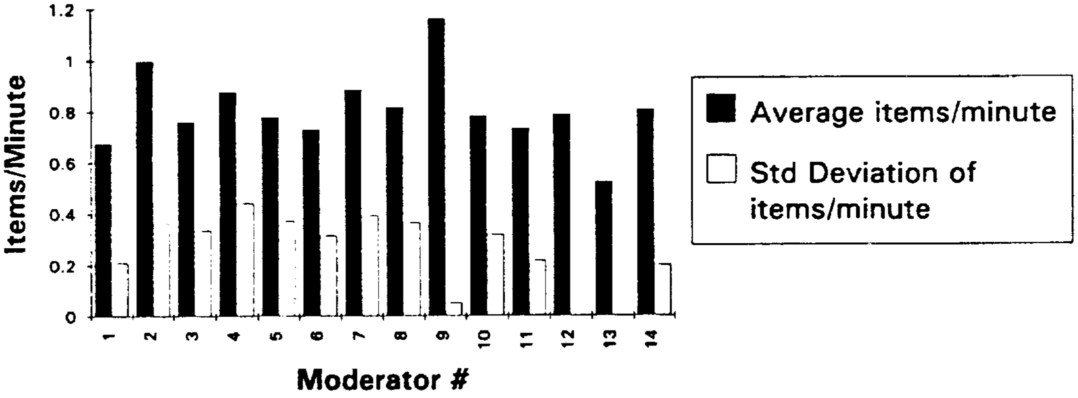


Figure 8: Moderator Finding Efficiency

The inspection rate is an important parameter to optimize. Going too slow may waste time, but going too fast will miss defects. Figure 9 shows the average defect density for different ranges of inspection rate. Note that we have normalized the defects found by the document size. As seen, going faster than about 50 pages per hour seems to substantially decrease the defects found. The overall average is 48 pages per hour, though we are currently trying to slow down the rate at meetings to be closer to 30-40 pages per hour.

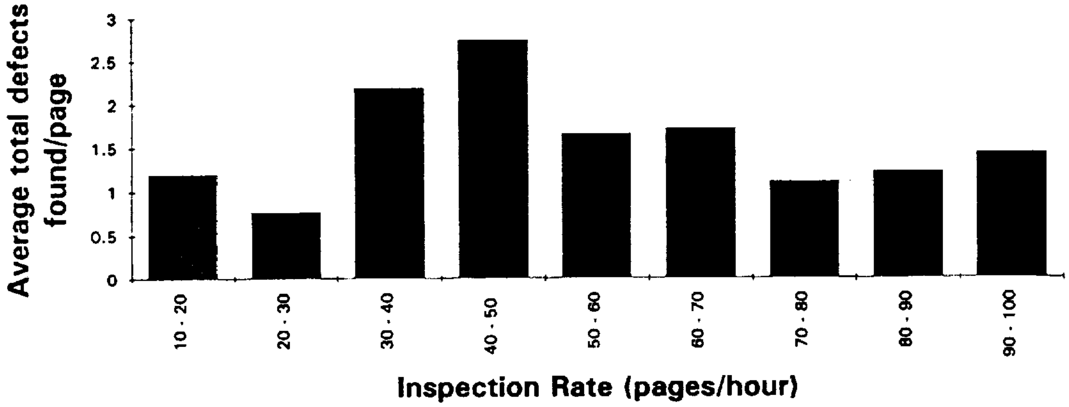


Figure 9: Effect of Inspection Rate on Defects Found

We also wish to know the optimal number of inspectors to maximize the defect removal effectiveness. Other studies have shown that 4-5 inspectors is the optimal number [Grady 92], [Gilb 88], and our data also supports this number. Figure 10 shows the average defect removal effectiveness for the number of inspectors. From our data, the optimum does not appear quite as clear-cut for major defects alone.

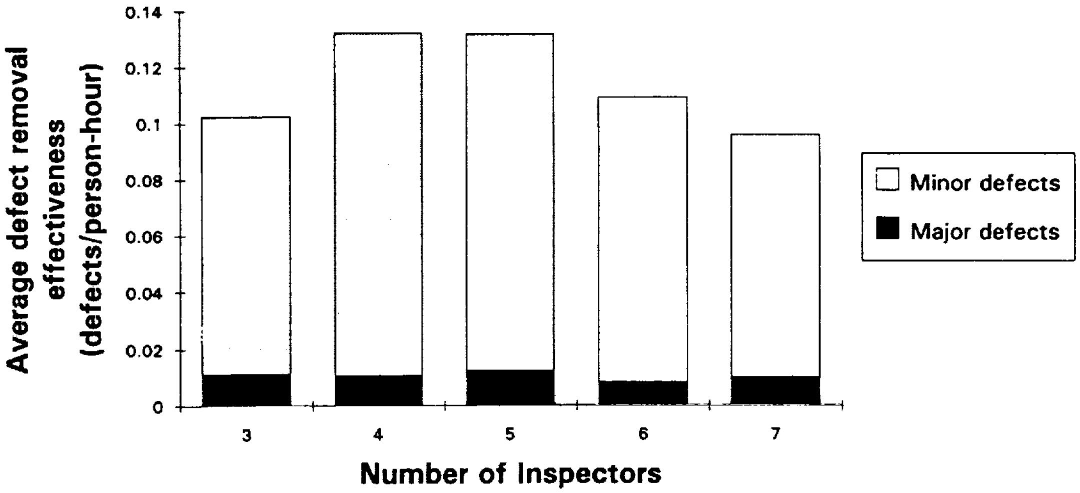


Figure 10: Average Defect Removal Effectiveness vs. Number of Inspectors

Yet another process parameter to optimize is the ratio of preparation time to inspection time. Grady and others [Grady 92] indicate an optimum value greater than 1.75, with some sites averaging about 1.5. Figure 11 shows our results. The optimum ratio appears to be somewhere between .5 and 2.0, with our average ratio being 1.4.

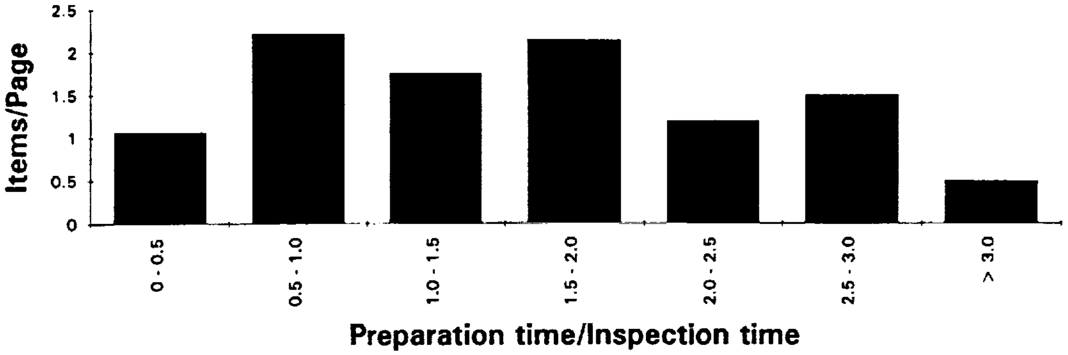


Figure 11: Defects Found vs. Preparation/Inspection Time

One counter intuitive result not previously reported in the literature is a high correlation (.8) between the preparation time (averaged over the inspectors) and new items/page or new items/KSLOC found during the inspection. Instead of catching less new defects during inspection after more thorough preparation to identify defects before the meeting, the inspectors are more familiar with the subject matter and thus able to find even more new items during the inspection meeting. A scatterplot of this data for all non-code documents is shown in Figure 12.

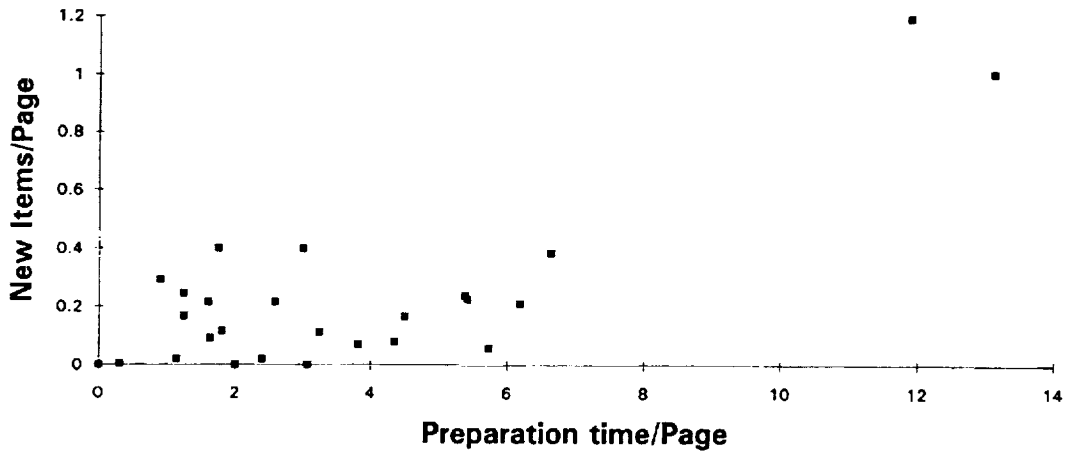


Figure 12: Effect of Preparation Time on New Items Found

As expected, there were also high correlations between preparation time and total items found (pre-inspection and new items) and inspection time versus both total items and new items. These relationships are more stable for design documents as opposed to code documents, due to the reduced clarity and understanding of program code.

Resulting Defect Density During Integration

Inspections are expected to severely reduce the number of problems encountered during testing and integration activities. Though this project is not 100% complete, data from the first couple of builds supports this hypothesis. Figure 13 shows the resulting defect density during integration, as the trouble report density running average by build. The first 10 builds were before inspections started, and the last two are for the current project within the same product line after inspections were mandated. Other project environmental factors are virtually identical except for the use of inspections. We are confident that something is helping to reduce the trouble report density.

Attempts were also made to perform a t-test on individual modules to determine if there are significant differences in defect density during testing due to inspection. The metrics tracking procedures did not lend themselves to such analysis due to intractable mappings between design documents and implemented code functions, actual code sizes could not be mapped at a low level to what was being inspected, and the inability to distinguish new development from modified code.

This experience was a lesson learned. In order to evaluate new techniques in the future for process improvement, the metrics procedures have to be restructured on the program, so that individual modules can be tracked throughout the lifecycle. Recommendations for the changes are being documented.

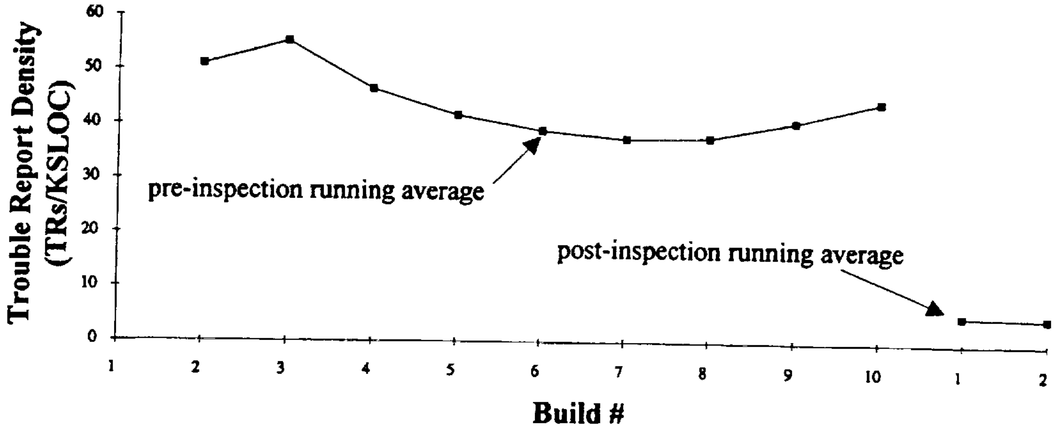


Figure 13: Defect Density During Integration

CONCLUSIONS AND FUTURE WORK

Though this initial major project using an inspection-based process is not complete, the preliminary results indicate a large return on investment. Since inspections began, inspectors have increased their effort and authors are producing higher quality documents, indicating buy-in to the new process.

Some process stabilization occurred during the first year of practice, and the teaching method and the process itself has been modified based on the statistical results. Inspections are being used on more ongoing projects, and the results appear to be repeatable within the company. The process is now mandated on all new projects.

This analysis has helped to identify areas of improvement for software metrics collection. This impetus will lead to revised procedures to enable more thorough analysis of process improvement activities.

Analysis of inspection data will continue in order to understand and account for the confounding factors of inspectors and authors, to continue identifying optimal practices, to perform more detailed cost/benefit analysis and to investigate other related process issues. Analysis of variance will be performed to determine the contribution of different process parameters to overall defect removal effectiveness.

With the recent enhancement to the data form for defect category information, defect metrics will be collected to support causal analysis activities. Additionally, a system dynamics model of an inspection-based process is under development, and will be calibrated to Litton data to assist in process improvement activities.

BIBLIOGRAPHY AND SELECTED NOTES

[Ackerman et al. 84] Ackerman AF, Fowler P, Ebenau R, *Software inspections and the industrial production of software*, in "Software Validation, Inspections-Testing-Verification-Alternatives" (H. Hausen, ed.), New York, NY, Elsevier Science Publishers, 1984, pp. 13-40

Describes inspections as performed at Bell Laboratories and discusses use of inspections in conjunction with other verification and validation techniques.

[Boehm 81] Boehm BW, *Software Engineering Economics*. Englewood Cliffs, NJ, Prentice-Hall, 1981, pp. 383-386

Discusses error removal production functions for inspection, unit test and other error removal techniques. Points out difficulty of overlap between methods for removing different classes of errors.

[Buck-Dobbins 84] Buck R, Dobbins J, *Application of software inspection methodology in design and code,* in "Software Validation, Inspections-Testing-Verification-Alternatives" (H. Hausen, ed.), New York, NY, Elsevier Science Publishers, 1984, pp. 41-56

[Fagan 76] Fagan ME, *Design and code inspections to reduce errors in program development*, IBM Systems Journal, V. 15, no. 3, 1976, pp. 182-210

The original article by Mike Fagan that introduced the IBM inspection experience.

[Fagan 86] Fagan ME, *Advances in software inspections,* IEEE Transactions on Software Engineering, V. SE-12, no. 7, July 1986, pp. 744-751

A more recent article by Fagan provides additional evidence of inspection benefits over the years, indicating slight front-end loading of the development effort and significant reduction in testing and rework effort.

[Freedman-Weinberg 82] Freedman D, Weinberg G, *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Prgoram, Projects and Products*, Little Brown, 1982

Good material on the human and organization aspects of inspections.

[Gilb 88] Gilb T, *Principles of Software Engineering Management*. Addison-Wesley, Wokingham, England, 1988, pp. 205-226, 403-422

Gilb originally taught the inspection method in-house, which was attended by high-level engineering management. Their strong support of the method led to our inspection-based process. This book provides ample detail to start an inspection progam.

[Grady 92] Grady R, Caswell D, *Practical Software Metrics for Project Management and Process Improvement* Prentice-Hall, Englewood Cliffs, NJ, 1992

Good current book on process improvement metrics with relatively brief but worthwhile treatment of inspections. Has an illustrative complete example of calculating inspection savings and cost/benefit on page 180.

[Kelly-Sherif 90] Kelly J, Sherif J, *An analysis of defect densities found during software inspections*, Proceedings of the Fifteenth Annual Software Engineering Workshop, Goddard Space Flight Center, 1990

[Radice-Phillips 88] Radice RA, Phillips RW, *Software Engineering - An Industrial Approach*, Englewood Cliffs, NJ, Prentice-Hall, 1988, pp. 242-261

A good overall treatment and summary of how to do inspections, by someone who helped pioneer inspections at IBM.

[Remus 84] Remus H, *Integrated software validation in the view of inspections /reviews,* in "Software Validation, Inspections-Testing-Verification-Alternatives" (H. Hausen, ed.), New York, NY, Elsevier Science Publishers, 1984, pp. 57-64

[Rodriguez 91] Rodriguez S, *SESD inspection results*, April 1991

The ROI tracking method was used at Hewlett-Packard.

[Scott-Decot 85] Scott B, Decot D, *Inspections at DSD - automating data input and data analysis*, HP Software Productivity Conference Proceedings, 1985, pp. 1-79 - 1-80

[Weller 93] Weller E, *Three years worth of inspection data*, IEEE Software, September 1993, pp. 38 - 45

Weller published a previous article in Crosstalk on the first two years of data at Bull HN Information Systems, and this article improves upon it.

ANALYSIS OF A SUCCESSFUL INSPECTION PROGRAM

Ray Madachy
Linda Little
Sylvia Fan

Litton Data Systems
Agoura Hills, CA

Presented at the Eighteenth Annual Software Engineering Workshop
NASA Goddard Space Flight Center
December 1, 1993

Litton
Data Systems

Outline

- Introduction and background
- Defect density versus inspection subject
- Inspection effort
- Return on investment
- Process control
- Defect density during integration
- Conclusions and future work
- References

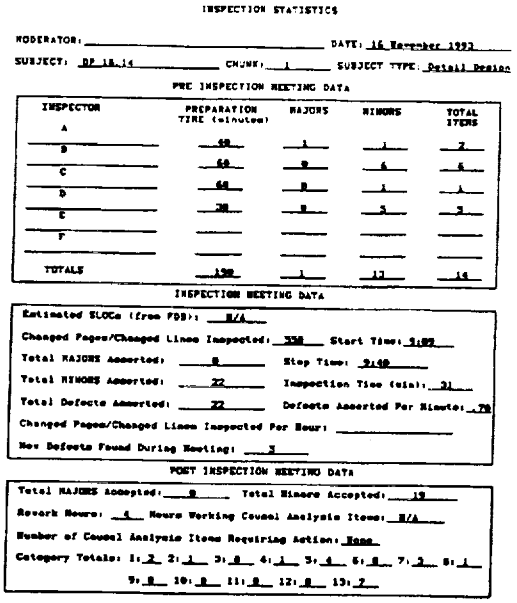
Litton
Data Systems

Unique Properties of Litton Inspection Process

- No "reader" role (Fagan).
- No discussion on defect category during inspection.
- Routing process.
- No time limit on causal analysis.
- SEPG Peer Review Coordinator serves as moderator.

Litton
Data Systems

Typical Data Sheet



INSPECTION STATISTICS

MODERATOR: ______ DATE: 16 November 1993
SUBJECT: DP 18.14 CHUNK: 1 SUBJECT TYPE: Detail Design

PRE INSPECTION MEETING DATA

INSPECTOR	PREPARATION TIME (minutes)	MAJORS	MINORS	TOTAL ITEMS
A	40	1	1	2
B	60	0	6	6
C	60	0	1	1
D	30	0	5	5
E				
F				
TOTALS	190	1	13	14

INSPECTION MEETING DATA

Estimated SLOCs (from PDB): N/A
Changed Pages/Changed Lines Inspected: 558 Start Time: 9:09
Total MAJORS Asserted: 0 Stop Time: 9:40
Total MINORS Asserted: 22 Inspection Time (min): 31
Total Defects Asserted: 22 Defects Asserted Per Minute: .70
Changed Pages/Changed Lines Inspected Per Hour: ______
New Defects Found During Meeting: 3

POST INSPECTION MEETING DATA

Total MAJORS Accepted: 0 Total Minors Accepted: 19
Rework Hours: 4 Hours Working Causal Analysis Items: N/A
Number of Causal Analysis Items Requiring Action: None
Category Totals: 1: 2 2: 1 3: 0 4: 1 5: 4 6: 0 7: 3 8: 1
9: 0 10: 0 11: 0 12: 0 13: 7

Litton
Data Systems

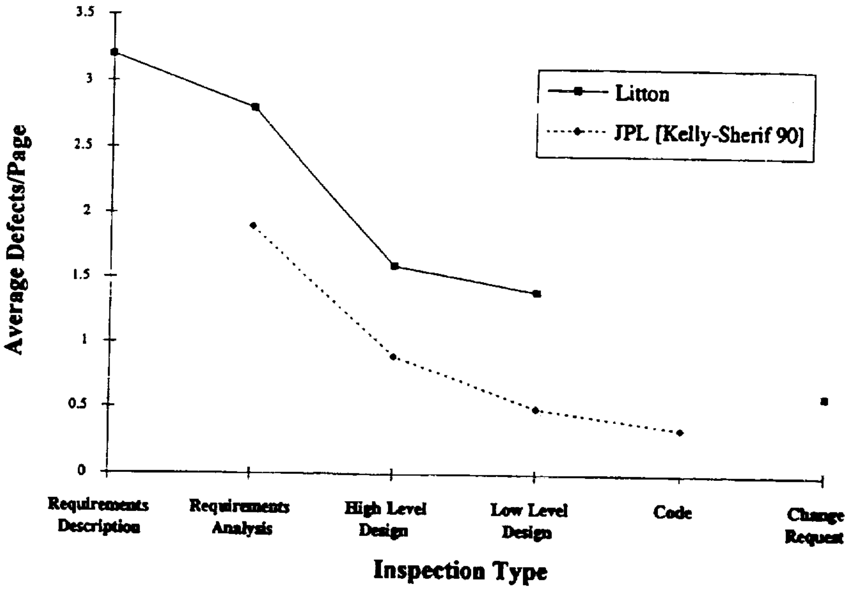
Summary Statistics

			Inspection		LOC	Major Defects/
Subject Type	Total Defects	Total Majors	Effort	# Pages	Inspected	Inspection Effort
REQUIREMENT DESCRIPTION	460	72	78	179	0	.923
REQUIREMENT ANALYSIS	2165	177	483	1065	0	.366
HIGH LEVEL DESIGN	2199	188	655	1592	0	.287
DETAILED DESIGN	1550	127	610	1387	19007	.208
Subtotal	6374	564	1826	4223	19007	.309
CODE	4272	432	1742	5047	149361	.248
CHANGE REQUEST	814	27	309	1579	0	.087
Grand total	11460	1023	3877	10849	168368	.264

Litton

Data Systems

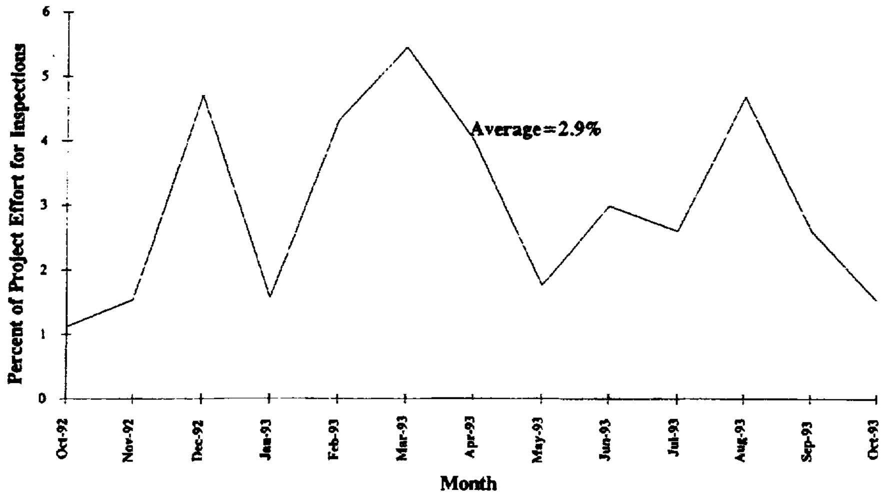
Defect Density per Subject Type



Litton

Data Systems

Inspection Effort



Litton
Data Systems

Return on Investment

- For each inspection, ROI = (test effort saved) - (inspection effort)

where

test effort saved =

(# major defects found)*(average effort to fix defect during test)

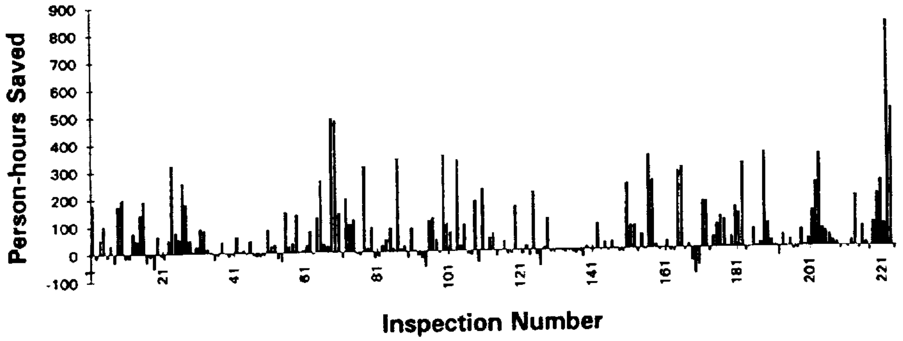
inspection effort = preparation effort + meeting effort + rework effort

= total preparation effort
+(meeting time) * (# personnel involved in meeting)
+ rework effort

Litton
Data Systems

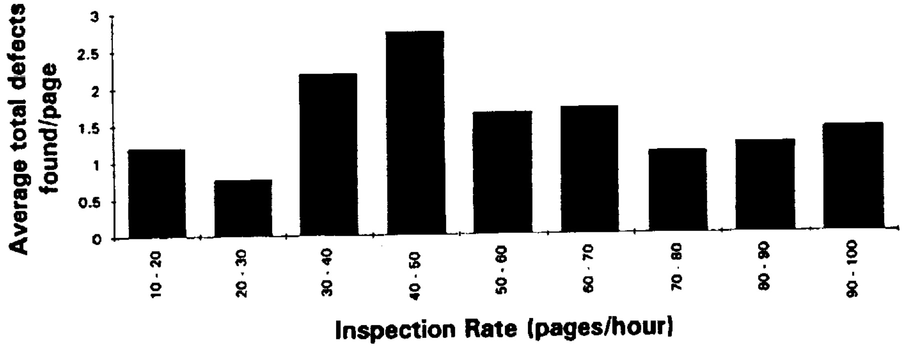
Return on Investment

total return = 14210 person-hours
average inspection savings = 63.4 person-hours
139/223 inspections saved time



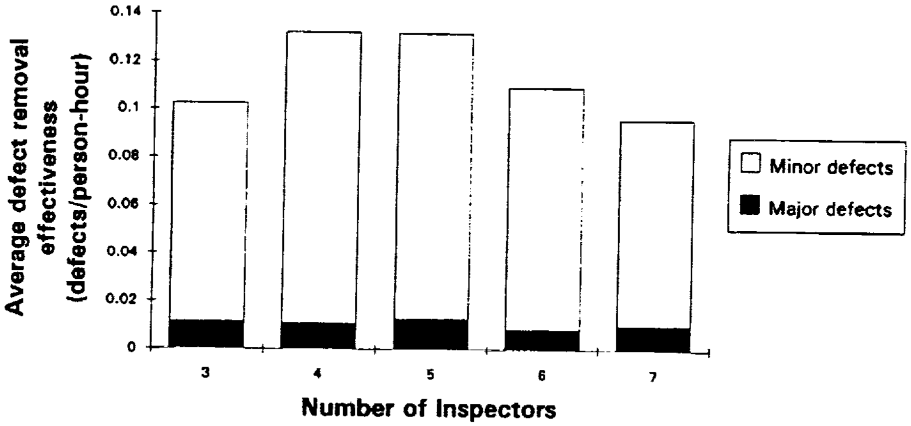
Litton
Data Systems

Effect of Inspection Rate on Defects Found



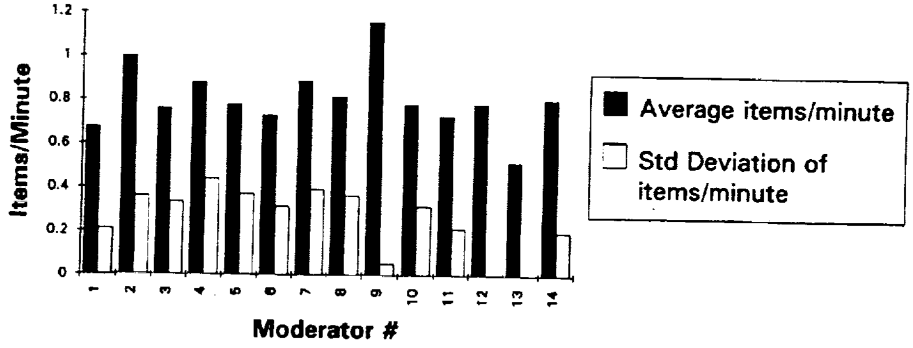
Litton
Data Systems

Defect Removal Effectiveness vs. Number of Inspectors



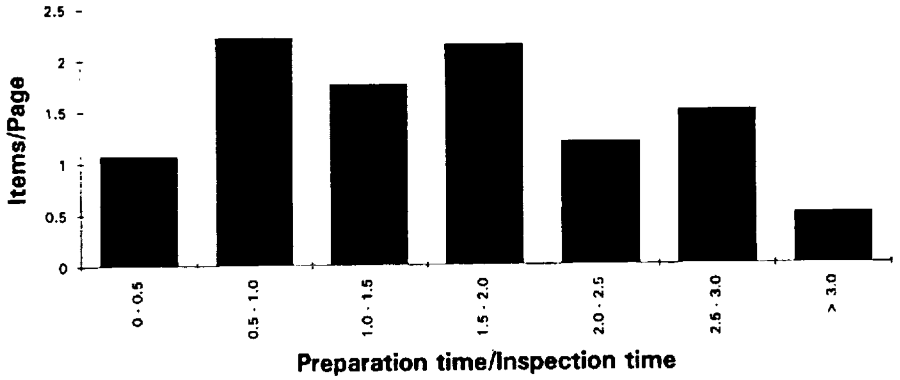
Litton
Data Systems

Moderator Finding Efficiency



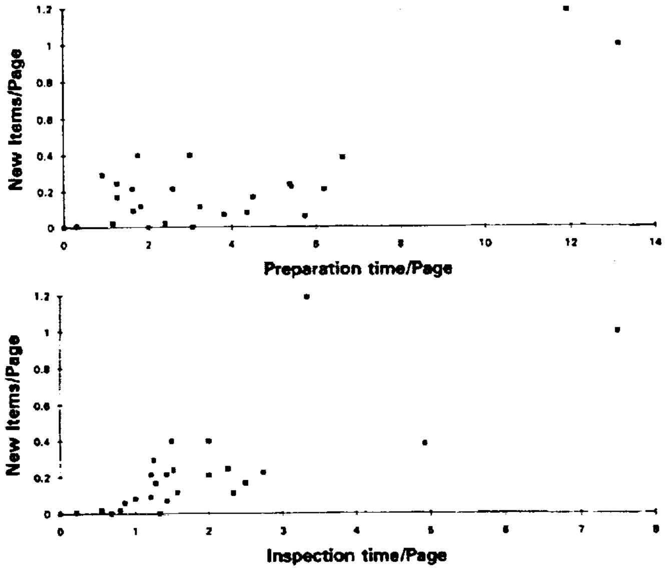
Litton
Data Systems

Effect of Preparation/Inspection Time Ratio on Defects Found



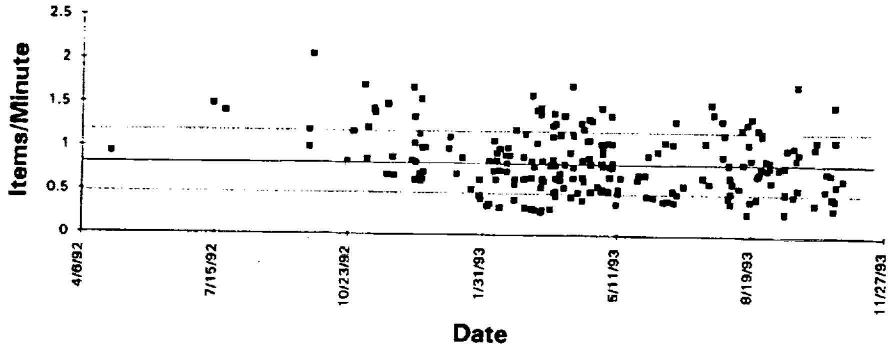
Litton
Data Systems

Effect of Preparation and Inspection Time on New Items Found



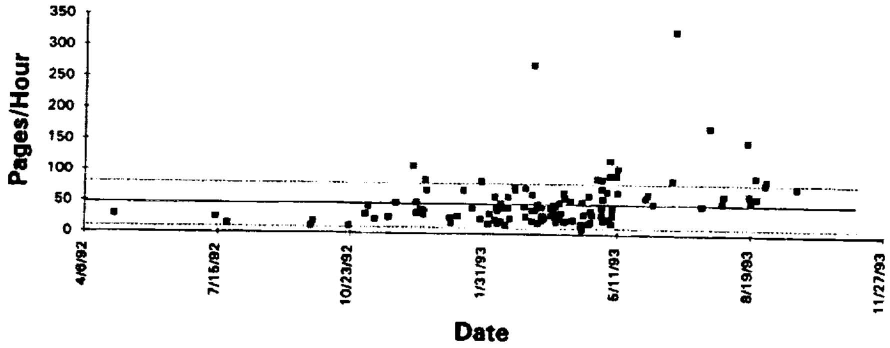
Litton
Data Systems

Defect Finding Rate Control Chart



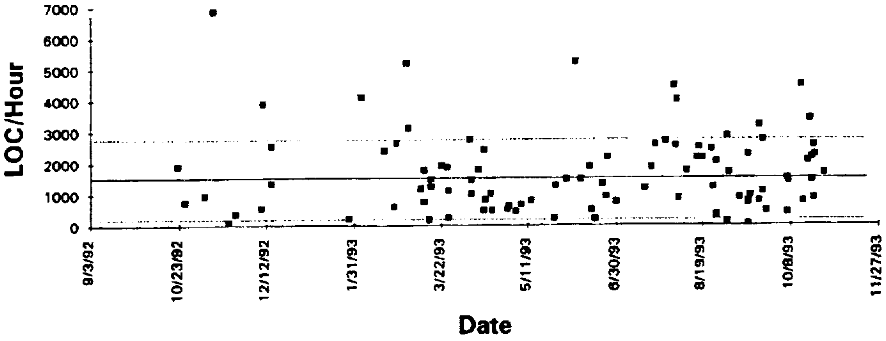
Litton
Data Systems

Document Inspection Rate Control Chart



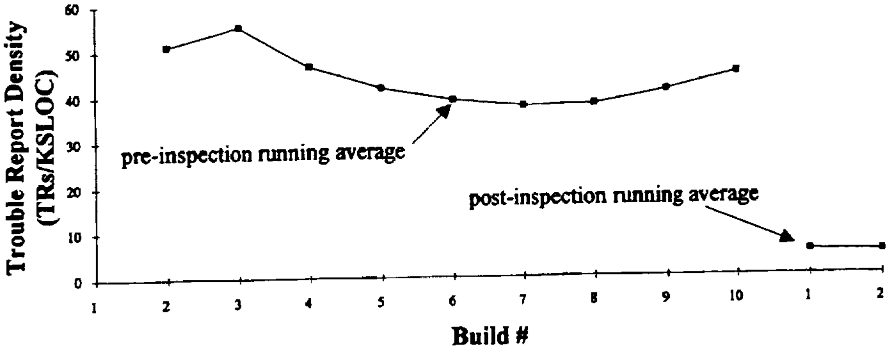
Litton
Data Systems

Code Inspection Rate Control Chart



Litton
Data Systems

Resulting Defect Density During Integration



Litton
Data Systems

Conclusions and Future Work

- Inspections are a worthwhile investment.
- Peer review coordinator essential to keeping process under control.
- Strong correlation between pre-inspection effort and new items found.
- Inspections appear to affect downstream artifacts and eventual system integration.
- Inspectors and authors have improved since inspections began.
- Some stabilization observed during first year of practice.
- Improved teaching method and changed process based on statistical results.
- Inspection analysis has provided impetus for improved metrics tracking procedures.
- Further analysis desired.
  - understand and account for confounding factors
  - defect category metrics and causal analysis
  - process control and optimization
  - ANOVA, other

Litton
Data Systems

References

Fagan ME, *Design and code inspections to reduce errors in program development,* IBM Systems Journal, V. 15, no. 3, 1976, pp. 182-210

Gilb T, Principles of Software Engineering Management, Addison-Wesley, Reading MA, 1988, pp. 205-226, 403-422

Grady R, Caswell D, *Practical Software Metrics for Project Management and Process Improvement* Prentice-Hall, Englewood Cliffs, NJ, 1992

Remus H, *Integrated software validation in the view of inspections /reviews,* in "Software Validation, Inspections-Testing-Verification-Alternatives" (H. Hausen, ed.), New York, NY, Elsevier Science Publishers, 1984, pp. 57-64

Weller EF, *Three years worth of inspection data,* IEEE Software, September 1993, pp. 38-45

Litton
Data Systems

N94- 36493

18th Annual Software Enginnering Workshop
Lessons Learned Applying CASE Methods/Tools To Ada Software Development Projects

59-61
12691
p. 52

December 1, 1993

Maurice H. Blumberg
(301)240-6018
blumberm@wmavm7.vnet.ibm.com
Dr. Richard L. Randall
(719)554-6597
randallr@wmavm7.vnet.ibm.com

STARS Project
IBM Federal Systems Company
800 N. Frederick Ave.
Gaithersburg, Md. 20879

Abstract

This paper describes the lessons learned from introducing CASE methods/tools into organizations and applying them to actual Ada software development projects. This paper will be useful to any organization planning to introduce a software engineering environment (SEE) or evolving an existing one. It contains management level lessons learned, as well as lessons learned in using specific SEE tools/methods. The experiences presented are from Alpha Test projects established under the the STARS (Software Technology for Adaptable and Reliable Systems) project. They reflect the frontend efforts by those projects to understand the tools/methods, initial experiences in their introduction and use, and later experiences in the use of specific tools/methods and the introduction of new ones.

Preface

This paper was prepared by the Maurice H. Blumberg and Dr. Richard L. L. Randall of the IBM Federal Systems Company, located at 800 North Frederick Avenue, Gaithersburg, MD 20879. Many thanks to fellow IBMers, Terry Snyder and Ron Backus of the GPS project and Frank Hyson and Neal Walters of the ACE project, for their inputs on lessons learned from their projects.

The following trademarks are used in this paper.

AIX, RISC System/6000, PS/2, and IBM are trademarks of the International Business Machines Corporation.

Rational, R1000, R300C, and Rational Environment are trademarks of Rational Corporation.

AdaMAT is a trademark of Dynamics Research Corporation.

Interleaf is a trademark of Interleaf, Inc.

Teamwork is a trademarks of Cadre Technologies Inc.

DocEXPRESS, DoDEXPRESS, and Methods_Help are trademarks of ATA Inc.

ObjectMaker (Adagen) is a trademark of Mark V Ltd.

STATEMATE is a trademark of i-Logix, Inc.

RTrace is a trademark of Protocol Company.

PVCS is a trademark of INTERSOLV, Inc.

CCC is a trademark of Softool, Inc.

LOGISCOPE is a trademark of Verilog, Inc.

Windows is a trademark of Microsoft Corporation.

Vax is a trademark of Digital Equipment Corporation.

Keywords: Software Engineering Environment, CASE, STARS, Methods, and Lessons Learned.

Table of Contents

Introduction

The objective of the Software Technology for Adaptable, Reliable Systems (STARS) Program is to develop, engineer, and integrate technologies that, when employed in the development of DoD software systems, will improve quality and predictability, and reduce the cost of development. STARS believes these improvements will primarily result from applying the "megaprogramming" paradigm which involves designing and building systems based upon tailorable reusable components, improvements in the software process, and through technology support for the development process. The STARS solution will embody these concepts with megaprogramming processes supported by software engineering environments (SEE).

STARS will accelerate a transition to a megaprogramming paradigm by demonstrating the benefits of megaprogramming on real DoD projects. This is being accomplished by a STARS Demonstration Activity, which was initiated in the 3Q92 and involves multiple "demonstration projects" in different application domains. The demonstration projects will formally begin their performance phase in 4Q93.

The STARS program is evolving and instantiating SEE solutions to support the demonstration projects. Prior to 4Q93, the STARS SEE solutions will evolve from the 1991-1992 "alpha versions" to integrated versions in 4Q93, running on various hardware platforms. In 1991 the IBM STARS team defined an initial generic instantiation of a SEE solution, designated as the IBM STARS Alpha SEE.

During 1Q91, IBM created the organization and structure to support "Alpha Test" projects, developing Ada software. The purpose of Alpha Test projects is to:

- Gain early experience and feedback in the use of the IBM STARS Alpha SEE Solutions
- Provide vehicle for early technology transfer of IBM STARS capabilities
- Be a precursor for STARS Demonstration Projects in defining:
    - A technology transfer process
    - How to support projects using a SEE
    - How to capture lessons learned information

IBM STARS Alpha SEE Solution

The IBM STARS Software Engineering Environment (SEE) is a combination of hardware platforms and software tools which support Ada software development from requirements analysis through code generation, testing and maintenance. The SEE is adaptable, i.e., it is tailorable to a "SEE Solution" which meets the specific needs of a project.

The IBM STARS Alpha SEE is based on IBM's AIX CASE Solutions. These solutions consist of IBM and IBM Business Partner products that support the software development process through software engineering methodologies, distributed workstation-based environment, and open system applications. The AIX CASE Solutions provide an open framework and a set of solutions and products supported across the range of the RISC System/6000 family. The IBM STARS team incorporated value add efforts from STARS into solutions where applicable.

The initial IBM STARS Alpha SEE solution was assembled from IBM AIX CASE Business Partners and other AIX CASE vendors. The figure below depicts the major hardware and software components.

```
                             --------   o AIX
                            | RISC   |  o AIX CASE Tools (e.g.,Teamwork)
                            | System/|  o Publishing Tools (e.g., Interleaf)
                            |  6000  |
                             --------
                                |
                                |
           ----------       -------------      ------------   o Design Facility
  o AIX | Xstation |      |   Local     |    | Rational   |  o Ada Language
        |   120    |----- |   Area      | ----| 300C or    |    Development
        |   130    |      |  Network    |    |   1000 ·   |    Environment
           ----------       -------------      ------------   o AdaMAT
                                |
                                |
                             --------
                            |        | o AIX
                            |  PS/2  | o AdaMAT Metrics Display Tool
                            |        |
                             --------
```

The software tools are integrated at varying levels within the initial solutions, e.g., there is a Rational/Teamwork interface which allows Teamwork analysis diagrams and text to be imported into Rational and a Teamwork/Interleaf interface for generating specifications and design documents.

The IBM STARS Team supported the Alpha Test projects in modifying the solution and adapting it to fit the project's process and methods.

IBM STARS Alpha Test Projects

The IBM STARS team has established three Alpha Test projects which are using the capabilities of the IBM STARS Alpha SEE to develop Ada software. The IBM STARS team has provided support to these Alpha Test projects and has collected feedback from them on their experiences with their SEE solutions.

The current IBM STARS Alpha Test projects are as follows:

- Global Positioning System (GPS)
- Ada CASE Engineering (ACE)
- Forward Area Air Defense (FAAD) Electronic Support Measure (ESM) Non-cooperative Target Recognition (NCTR) System (FAADS)

The table below provides a summary of the major tools used by the Alpha Test projects, categorized by system life cycle activities (see Appendix A for a description of these tools).

Life Cycle Activity	GPS	ACE	FAADS
Analysis	Teamwork	Teamwork STATEMATE	N/A
Design	Adagen Rational	Adagen Rational	Teamwork/Ada & DSE
Implementation	Rational	Rational	TLD Ada Compiler
Document Generation	Teamwork DocEXPRESS Interleaf Rational	Rational	Teamwork DocEXPRESS Interleaf
Reverse Engineering	Adagen	Adagen	Adagen
Requirements Traceability	Rational	Rqt RTrace RTM Rational	Manual

Table 1. Alpha Test Projects SEE Tool Usage

The Alpha Test projects are using a wide range of SEE tools and methods covering the entire system life cycle. However, to provide some focus to the Alpha Test efforts, each project was asked to concentrate on a specific aspect of the lifecycle. GPS is focusing primarily on system engineering and software design, ACE on requirements traceability and software design, and FAADS on software design and reusability.

Each of the Alpha Test projects are described in the sections that follow. The descriptions include their hardware/software configuration, tool usage, and lessons learned from introducing and using CASE tools/methods. The "introduction" lessons learned reflect, for the most part, the frontend efforts to understand the tools/methods and some early experiences in the use of them. The "using" lessons learned reflect, for the most part, later experiences in the use of the specific tools/methods and the introduction of new ones.

Lessons Learned Introducing and Using CASE Tools

Global Positioning System (GPS)

Description

The Global Positioning System is a space based navigation system consisting of a constellation of Space Vehicles (SVs) and a ground support system. The GPS project is responsible for the hardware and software development of the ground support system. This includes software to generate the navigation data, upload the SVs, process telemetry data, and in general, provide commanding and control of the SVs. Project responsibilities also include the maintenance and upgrade of hardware at the remote tracking stations and master control station. The GPS project is in its 13th year of development and follow-on contracts. The system consists of approximately 1 million SLOC, mostly in JOVIAL.

The current GPS effort includes the development of two new Computer Software Configuration Items (CSCIs), requiring approximately 32 KSLOC of Ada. These CSCIs are being developed on RISC System/6000 and workstations; they will also be run operationally on RISC System/6000 and Rational hardware.

Initially, the IBM STARS Alpha SEE was used to support the Ada software design, Software Design Document (SDD) generation, and Ada code development for the CSCIs. Subsequently, the SEE was also used to develop a Software Requirements Specification (SRS) using OOA and the SEE Tools, Teamwork and Interleaf. Future plans include use of Teamwork/Ada for high-level software design.

GPS Hardware/Software Configuration

The GPS hardware/software configuration is depicted in the figure below.

GPS Hardware/Software Configuration

```
                              --------   o AIX
                             | RISC  |   o Teamwork
                             | System/|  o ObjectMaker (Adagen)
                             |  6000 |   o Interleaf
                              --------
                                  |
                                  |
                                  |
            ----------       -------------        ------------    o Design Facility
     o AIX |          |     | Local Area  |      |  Rational  |   o Ada Language
           |   PS/2   |---- |   Network   |      |    300C    |     Development
           |          |     | (Token Ring)|      |            |     Environment
            ----------       -------------        ------------    o Remote Compile
                                  | |                   |            Integrator
                                  | +-------------+     |
                                  |               |     |Ethernet
                             --------              --------
            o DOS           |        |            | RISC   | o DASD for Rational
            o Windows 3.0   |  PS/2  |            | System |
            o Adagen        |        |            |  6000  |
            o FTP (TCP/IP)   --------              --------
```

GPS SEE Tool Usage

The IBM STARS Alpha SEE tools currently being used by GPS are Teamwork (and DocEXPRESS), ObjectMaker (formerly known as Adagen), and Rational. The use of these tools is described below.

Teamwork

GPS initiated an effort to use OOA methods and Teamwork to generate an SRS for the Onboard Processor CSCI (Computer Software Configuration Item). Because of funding and scheduling issues, the system engineer (SE) responsible for the SRS could not take any formal classes on OOA or Teamwork. This provided an opportunity to determine the effectiveness of learning a CASE tool and the methods it supports without formal classroom training. The training approach developed by STARS for the GPS system engineer was as follows:

- Go thru on-line Teamwork/SA tutorial - 1 day
- Go thru "Strategies for Real Time Systems Specification" book by Derek Hatley - 3 days
- Go thru "Object-Oriented Analysis: Modeling the World in Data" and "Object Lifecycles: Modeling the World in States" books by Sally Shlaer and Steve Mellor - 3 days
- Go thru on-line training tools, DoDEXPRESS and Methods_Help - 1 1/2 days
- Experiment with Teamwork - 2 days
- Bring in Cadre system engineer (no charge) for a Q & A session - 1 day

GPS also used the DocEXPRESS tool which generates a DoD-STD-2167A compliant SRS from the Teamwork developed OOA model. The DocEXPRESS vendor, ATA Inc., was funded to enhance the tool to generate requirements traceability matrices as part of the SRS.

Teamwork is also being considered by the GPS software engineering team to support an object based Ada software engineering methodology. This includes use of Teamwork/Ada for creating Ada Structure Graphs as well as compilation dependency and Buhr diagrams. The interface between Teamwork and Rational is also being studied to determine how far to go with Teamwork before migrating to the Rational.

ObjectMaker (Adagen)

ObjectMaker (Adagen) was originally used by GPS for early conception of design, preparation of high level architecture and design overview material. Later on, it was used for reverse engineering of Ada code to diagrams for inclusion in the SDD (Software Design Document). Specifically, for conceptual design, Adagen was used to draw bubble charts which showed relationships between objects and message flow. In addition, Booch class category charts were used for CSC (Computer Software Component) evolution, and Buhr diagrams were used for CSU (Computer Software Unit) evolution and interaction. Reverse engineering of Ada code, using Adagen, was employed to ensure that the Ada graphical diagrams in the SDD were consistent with the Ada source code. The diagrams generated by Adagen were compilation dependency diagrams, which showed the "withing" relationship between Ada packages and Buhr diagrams, which graphically depicted the contents of each Ada package.

Rational

The Rational design facility and environment was originally used by GPS for development of PDL for preliminary and detailed design, generation of SDDs, code development, some unit testing, and requirements traceability to code. The Rational Design Facility was further customized to provide the capability to include Ada package specifications in-line in the Software Design Document. Additionally it provided for Appendices to address what Non-Developed Software (NDS) is being used in the system. In addition, the newly available Remote Compilation Integrator (RCI) was used to allow source code on the Rational to be compiled on the RISC System/6000. The RCI was used in conjunction with Rational's Configuration Management Version Control.

GPS Lessons Learned from Introducing CASE Tools

The lessons learned by the GPS project from introducing the IBM STARS Alpha SEE are described below:

- Significant start up preparation and cost for a new Ada project

  For GPS, an existing project transitioning to Ada, many new things needed to be learned by the software developers, including a new language, a new set of tools and methods, and a new process (i.e., DoD-STD-2167A). This required considerable training costs and a significant learning curve for the project team.

- Customization of SEE tools requires significant resources

  Several of the tools used by GPS required customization, e.g., Rational needed customization to produce an SDD which conformed to the customers requirements, provide requirements traceability, and imbed diagrams from Adagen. Additional customization of Rational was required to extract PDL for the SDD. This customization required several labor months of effort, with additional customization still required.

- Choose a project methodology and train developers early

  The decision to use Ada as the programmming language was made well after the start of the current GPS effort. Thus, choice of a method and tools, and all the startup costs mentioned above, were not part of the initial project planning and required significant adjustments to the original plans and schedules.

- Have engineers use object oriented analysis for specifications

The GPS system engineers used functional decomposition methods to generate their SRS, while the software engineers used object-based design methods for the Ada software design. The use of an object oriented approach to defining requirements would reduce the effort required to transition between the specification and design phases.

- Use Ada as the design language

  Using Ada as the design language provides a compilable design which can be checked for completeness/consistency of interfaces, data definitions, etc.

- Preserve ability to extract PDL after code completion

  The ability to extract PDL from the code (e.g., for inclusion in an SDD) allows design and code to be maintained in a single place and decreases configuration management requirements.

- Agree with customer on diagramming and PDL techniques early into a project

  As mentioned above, choice of Ada and associated design methods and tools were made after the start of the current GPS effort. As a result, several iterations with the customer were required to gain agreement on issues regarding the formats and styles of Ada diagrams (e.g., Buhr diagrams) and PDL.

- Plan for a target code version and unit tests on target

  The GPS Ada source code was developed, compiled, and partially unit tested on the Rational hardware. Since the target machine is the RISC System/6000, provision was made for comprehensive unit testing on the target machine, even though the Ada code is "compatible". In addition, configuration management of source and target software needed to be provided for.

- Need additional personnel roles

  New project roles are required as a result of introducing SEE methods and tools into a project. In addition to a methods/tools "guru(s)" for consultation, GPS required the following additional personnel roles:

  - Rational System Administrator
  - Rational CMVC/RCF Administrator
  - Rational Design Facility Customizer
  - Adagen Support Expert

  Note: Multiple roles can be played by one person.

GPS Ada Lessons Learned from Using CASE Tools

The lessons learned on the GPS project from using the IBM STARS Alpha SEE tools/methods are described below:

Teamwork Lessons Learned

- An understanding of the basic capabilities of Teamwork was gained without formal classroom training.

  The Teamwork tutorial which provides hands-on training is adequate for new users to learn the basic capabilities of Teamwork. Building data flow diagrams, entity relationship diagrams, and defining entities, data flows, processes, stores, their corresponding attributes is relatively straight-forward. Teamwork/Ada was also fairly easy to use without formal training. The various user's guides, provided for each of the editors, are also well-written and provide detailed information on the use of the editors.

- Formal classroom training is required for understanding object-oriented methods.

Learning any new methodology is a challenge. However, object-oriented methods require a totally different approach and way of thinking that is quite different from traditional structured analysis (SA) and design methods. For example, most system engineers are so ingrained in using some form of SA for requirements analysis that without formal training, they will have a difficult time understanding OOA and transitioning from SA to OOA methods.

- Generating an SRS which satisfies the customer's DoD-STD-2167A DIDs (Data Item Descriptions) requires considerable tailoring of the Teamwork provided templates.

  The Teamwork document generation capability is powerful, but fairly complex. Tailoring the SRS and extracting the specific data requires significant time and effort. Also, the lack of a table building capability contributes to the difficulty in generating documents. The use of DocEXPRESS simplified generation of 2167A compliant documentation from Teamwork, but it required considerable enhancements (by the DocEXPRESS vendor) to generate an SRS which satisfied the customer's DoD-STD-2167A DIDs (e.g., provide requirements traceability matrices).

Adagen Lessons Learned

- An understanding of the capabilities of Adagen was gained without formal classroom training.

  Adagen was fairly easy to learn and has a well-designed user interface. The Adagen tutorial provides hands-on training and is adequate for new users to learn the basic capabilities.

- Reverse engineering of Ada code to create design diagrams for documentation and communication of design is very effective.

  The reverse engineering of Ada code ensured that the Ada graphical diagrams in the SDD were consistent with the Ada source code. However, some manual editing of the diagrams generated by Adagen was required.

Rational Lessons Learned

- Expense and overhead of supporting the Rational development environment is high.

  The Rational development environment has served the GPS project well in developing and testing of Ada code. However, Rational is somewhat inflexible in its ability to increase the number of users. Tokens are also fairly expensive and cannot be leased. Upgrading the Rational hardware to support additional users can also become an issue.

- A significant effort was required to customize the Rational Design Facility (RDF).

  The customization of the RDF to include Ada package specifications in-line in the Software Design Document and provide appendices for NDS required several labor-months to complete.

- Extensive training was required to fully utilize Remote Compilation Integrator.

  Extensive training was required to fully understand the implications of using the Remote Compilation Integrator, in conjunction with Rational's Configuration Management Version Control.

- Unit testing within Rational of Ada code using multiple COTS tools requires a significant effort.

  The Rational debugger works well for code which is machine independent and does not call the operating system or other COTS software. Unfortunately, the GPS application is dependent on the operating system and on a COTS GUI package. The Remote Procedure Call (RPC) capability of Rational would enable GPS to use the debugger through more of the unit testing cycle. However, a significant amount of resources is required to implement and support RPC.

Integration Lessons Learned

- Integrating tools to build an environment to support the entire life cycle is difficult.

  Although each of the GPS CASE tools performed well in its intended phase of the life cycle, their underlying data representations are not easily shared across other tools or products. Incorporating changes in one phase into the products of a previous or future phase is difficult if another tool is required to produce those products.

- No CASE tool currently addresses the need for page integrity.

  As software is modified to incorporate enhancements, the supporting document's page numbers cannot change. Add pages are used to accomplish this. Many CASE tools can generate a document but they are not good publishing tools. As a result, none support page integrity. It is hoped that Interleaf, which is a publishing tool, will address this issue. Unfortunately, using another tool adds to the problem of configuration management across tools.

Ada CASE Engineering (ACE)

Description

The Ada CASE Engineering (ACE) project was established at the end of 1988 to perform ongoing evaluations of tools and methods that can improve the process of developing Ada software, from proposal activity through maintenance. To accomplish this, a CASE tool environment laboratory was set up at FSC Manassas and investigations of various methods which could be successfully used with these tools were performed (this included conducting pilot projects).

Since 1988, the ACE project has played a lead role in infusing new tools and methods into FSC systems engineering and software development, with the goal of improving the productivity and quality of Ada software. This has included providing a variety of training classes for both tools and methods. ACE has also supported new Ada projects that use CASE tools and the Rational Ada development environment.

In 1992 the ACE project has continued to investigate the application of CASE tools and methods for Ada systems development. Early this year the project supported the IBM Manassas' efforts to evaluate the Integrated CASE (I-CASE) RFP from the U. S. Air Force. The I-CASE RFP was reviewed and found to contain more than 900 requirements. The large number of requirements gave ACE an opportunity to put the draft RFP under the RTrace requirements tool so that they could better manage the total scope of the requirements.

The ACE project continues to strongly focus on CASE tools hosted on the IBM RISC System/6000. A major emphasis has been on the requirements definition and modeling tool, STATEMATE, from i-Logix, the requirements traceability tool, RTrace, from Protocol as well as other tools. Efforts are on-going for investigating reuse techniques including the formulation of an object-oriented domain model for an existing project. ACE is using STATEMATE, RTrace, ObjectMaker, Rational and other tools to develop a SEE.

ACE Hardware/Software Configuration

The ACE hardware/software configuration is depicted in the figure below.

```
            ACE Hardware/Software Configuration

                    --------   o AIX
                   |  RISC  |  o STATEMATE
                   | System/|  o IBM Ada Compiler
                   |  6000  |  o Interleaf
                    --------   o ObjectMaker
                       |       o Teamwork
                       |       o LOGISCOPE
                       |       o RTrace
 ----------        -------------          ------------  o Design Facility
| Xstation |      | Local Area  |Ethernet|  Rational  | o Ada Language
|   120    |-----|   Network    |--------|   R1000    |   Development
|          |      | (Token Ring)|        |            |   Environment
 ----------        -------------          ------------  o AdaMAT
  o AIX                 |                               o Custom Tools
                        |
                    --------
                   |        | o DOS
                   |  PS/2  | o Windows 3.0
                   |        | o Rational Interface
                    --------  o AdaMAT Metrics Display Tool
                              o FTP (TCP/IP)
```

The specific components of configuration are as follows:

- RISC System/6000 Model 530
- Rational R1000, Series 400
- Token Ring and Ethernet LAN
- Xstation terminals for LAN access
- PostScript Printer (IBM 4216)
- STATEMATE, ObjectMaker, Teamwork, RTrace, Ada Compiler, LOGISCOPE, and Interleaf on RISC System/6000
- Rational Design Facility (RDF) on Rational R1000
- Rational interface, AdaMAT Metrics Display Tool (MDT) on PS/2

ACE Tool Usage

The IBM STARS Alpha SEE tools currently being used by ACE are STATEMATE, RTrace, Rational, and ObjectMaker. Other tools used include LOGISCOPE, AdaMAT and the AdaMAT Metrics Display Tool (MDT), text editors, and text postprocessors. The use of these tools are described below.

STATEMATE

STATEMATE, from i-Logix Company, is a graphic modeling tool. It ties together three types of diagrams which can be used to model systems. The Statechart is very much like a state transition diagram. The Activity chart is like a data flow diagram and Module charts show the structural view of a system. The "languages" of the Statechart and the Activity chart are very sophisticated, including time based functions, yet simple to draw and manipulate. STATEMATE includes extensive model checking, DoD documentation generation, model simulation and prototype code generation. The prototype code generated by STATEMATE can be used to drive screen panels, and take panel actions as inputs.

STATEMATE is used by ACE to perform requirements analysis and modeling of the problem domain. This methodology uses a technique for characterizing requirements as objects and accurately models system behavior. STATEMATE performs completeness/consistency checking and captures all definitions of external and internal data. STATEMATE produces the SRS and IRS (Interface Requirements Specification) documents according to DoD-STD-2167A standards. The ACE project customized this documentation facility to better match the object-oriented methodology that it is defining as part of its SEE development effort.

Modeling

The ACE group used STATEMATE to build an object-oriented analysis/design model. The model is built using Module Charts to represent the objects, Statecharts to represent the behavior of each object and the interaction between objects, and an Activity Chart to show the context of the problem. The Statecharts were executed to illustrate performance modeling (run-time overhead and processing times), event-response scenarios and error recovery scenarios. The STATEMATE model and timing chart outputs will be documented in Version 2.1 of the model software specification that the project is writing.

Rapid Prototyping in Ada

There was a two week effort to evaluate the Ada prototyping capabilities of STATEMATE. A Traffic Light Model and Panel were successfully built and executed. The last obstacle regarding the prototype code driving the display panel was solved, by using the "hooks" option. STATEMATE code generation does not normally generate hooks for states, activities, etc. unless specifically asked for.

The Ada code for the system (1 Statechart, 1 Activity chart, 8 states and one panel) amounted to about 1500 SLOC of Ada. The Ada code generated models the states and events defined in the system, and drives a simple panel.

RTrace

The RTrace requirements traceability tool currently is used by ACE as a stand-alone package to identify requirements from a customer A-Spec and to build a requirements data base. These requirements are then allocated to various system objects and components. This allocation may be used to do impact analysis if, for example, a requirement should change. This tool has extensive reporting capabilities, and some report formats were customized to better support technical and project management activity.

RTRACE is currently hosted on a SUN or DEC platform. Protocol has recently ported RTRACE to the RISC System/6000.

Rational

The Rational design facility and environment is used by ACE to produce various DoD-STD-2167A documents including preliminary SDDs and IDDs from the preliminary design activity, and final SDDs and IDDs (Interface Design Document) from the detailed design activity. Rational's automated document generation facility allows documents to be produced from one common evolving source. An interface from Rational to ObjectMaker is also used to allow diagrams generated by ObjectMaker to be included in the appropriate documents. Rational is used for design, code development, unit testing, and requirements traceability to code.

ObjectMaker

ObjectMaker is used by ACE to support the development of high level graphical Ada design constructs that result from the STATEMATE model. ObjectMaker is then used to develop Ada code design diagrams (Buhr diagrams), and from these diagrams, generate Ada skeletal code. After code completion, ObjectMaker is used to perform reverse engineering to create accurate Ada graphical diagrams from the Ada code, for reviews and incorporation into the SDD. The diagrams generated by ObjectMaker are compilation dependency diagrams, which show the "withing" relationship between Ada packages and Buhr diagrams, which graphically depict the contents of each Ada package.

LOGISCOPE

LOGISCOPE is used to perform metrics analysis of Ada code to enhance reliability, maintainability, and portability. By quantifying the Ada software quality, LOGISCOPE identifies potential problems in the Ada code, provides test coverage and complexity metrics, and addresses performance issues. LOGISCOPE is also used to provide graphical reports of the metrics.

AdaMAT

AdaMAT is an Ada metrics tool that runs on the Rational. Like LOGISCOPE, it checks on the conformance of Ada code to a wide variety of quality indicators. AdaMAT provides a number of reports and generates data that may be further analyzed off-line with a PS/2 based tool.

ACE Lessons Learned from Introducing CASE Tools

The lessons learned by the ACE project from introducing the IBM STARS Alpha SEE are described below:

- The single most important key to the success of a project is still to understand the problem thoroughly.

  There is still no substitute for sound systems and software engineering. SEE tools and methods only provide support to this process by providing a structured approach to recording and checking the results of sound engineering, and a means for communicating the results more clearly to others.

- Adequate training in tools/methods must be provided.

  SEE tools/methods require significant training to learn. Lack of adequate training can lead to misuse of tools, causing a negative impact on a project, and resulting in tools becoming expensive "shelfware".

- New methods and tools require considerable time to learn and this time must be allocated to a project schedule.

  In addition to proper training, SEE tools/methods require considerable hands on use before developers are proficient in their application to real problems. This learning curve must be accounted for, especially in the front end costs of a project.

- Tools require considerable lead time before they are operational.

  Significant customization of tools to a project's specific needs and documentation of detailed project standards and procedures are required to make SEE tools "operational". In addition, bridges between tools, e.g., Adagen and Rational need to be developed.

- New methods/tools need to have a strong project advocate.

  Because of the significant startup costs mentioned above, and the the long lead times required before new methods/tools begin to make an impact, a strong project advocate is needed to maintain the project commitment until the benefits of the tools/methods are realized.

- A project should have a 'toolsmith' who can customize tools to the project when necessary.

  In addition to the initial customization of tools, there is an ongoing requirement for customization to tailor tools to the changing needs of a project.

- Consider whether a tool/method might not 'scale up' to a large project.

Many tools/methods/notations look very good when applied to relatively simple problems, but yield very complex and difficult to understand results when applied to large, fairly complex problems. In evaluating tools/methods, system developers need to look beyond the simple examples that are used to demonstrate the application of those tools/methods and determine if they will scale up to their specific problem domain.

- Having tools available in the office via networking is a productivity enhancer.

  Having access to SEE tools from office desktop computers, provides convenient access to these tools, while taking advantage of existing computing resources.

- New tools and methods should not be seen as a panacea.

  This is another way of emphasizing that there is still no substitute for sound systems and software engineering.

ACE Lessons Learned from Using CASE Tools

The ACE project has recently been developing a software engineering environment to be used by an upcoming Ada project at Manassas. This tool's environment will provide life-cycle support from requirements tracing and capture through software design and development, testing and maintenance activities. The lessons learned are from the experience in developing this SEE and from other activities.

STATEMATE

- The STATEMATE panel generator and Ada Prototyper provide an interesting and informative view into a model.

  These tools have great potential for modeling network and processor performance, timing, concurrent processing, error paths and event-response. They are also good for demonstrating user interfaces, but the generated code models STATEMATE states/events/etc, and is not likely to be useful for real code design/development.

- The language and semantics of STATEMATE require a steep learning curve.

  STATEMATE is a powerful tool that requires a fair amount of time to fully understand. For example, it was very time consuming to figure out the best way to model the first object under STATEMATE; however, once this was understood it became fairly mechanical to add new objects.

  The ACE estimate for training is a minimum of one week of hands-on training, followed by three weeks of hands-on prototyping with available expert consulting. The training and consulting could be provided internally as sufficient skills are developed and a course/prototyping exercise is developed.

- Definition of STATEMATE naming conventions is very important.

  Good naming conventions are important not only for STATEMATE modeling and prototyping activities, but for all CASE tool efforts. In the STATEMATE object model, each object service needs two conditions and two events associated with it as well as data-items to define simulated processing times. All these items require names. Some of these items need to be global in scope, some are of local scope - they can be defined either way. (In this sense STATEMATE is somewhat like Ada - you can overload a name in the proper context).

RTrace

- RTrace supports 2167A requirements traceability.

  Based on the testing done, the review of the documentation and the support received from Protocol Company, ACE recommended using RTRACE for projects requiring 2167A traceability. RTrace can also be used for proposal requirements management, e.g., tracing RFP requirements to proposal sections and responsible authors.

- RTrace is easy to use and should not require formal education.

  The current version of RTRACE has a fairly straight-forward and menu oriented, key-intensive user interface. Protocol is planning to have a future Motif user interface. Protocol should help set up the first project's database. It will save a lot of time later if all the "objects" and their relationships are defined completely and correctly before the requirements are loaded. Examples of RTRACE objects are: Configuration Items, Functions and Sub-Functions, responsible engineer, test drop, build number, test procedures, etc.

- RTRACE is not integrated with any other CASE tool.

  Because RTrace is a standalone tool, this means all updates must be done manually. Updating data from tool to tool will be easier in the future windows environment.

- Projects using RTRACE will need a "guru" to support users.

  Projects using RTRACE can use co-ops or junior level people to parse existing documents to enter into the RTRACE database. After that, knowledgeable engineers, programmers, and testers must assign characteristics and link objects to each requirement being traced. A tools "guru" is needed to customize reports and do some tool administration functions.

- The current release of RTRACE does not support any automated configuration management or version control.

  Some form of version control is planned for future releases. This should provide the potential for integration with configuration management tools.

Miscellaneous Lessons Learned

- Many tools need to be customized before they can be used on a project.

  Most CASE tools, such as STATEMATE and Rational, require customizing in order to support the desired methodology and unique requirements of a project (including the customer's requirements). Thus, for most projects a toolsmith is a necessity.

- Many new tools have a significant learning curve.

  If the users do not have adequate learning time and motivation, this will likely kill the tool's chance of acceptance with users and management. If possible, new tools should be chosen to operate in the same manner as earlier tools they replace. This will give the user the sense that their previous skill with the older tool has not been wasted.

- Educating the user group is an important part of introducing a new toolset.

  Once a toolset and methodology are selected, an educational plan (schedules, preparation, funding) needs to be addressed as early as possible that will support this methodology. In the beginning, tools environments may succeed by virtue of attracting enthusiastic individuals. Ultimately a good teaching method is needed to extend tool use to those who would rather keep the status quo.

- Anticipated users of a toolset should have training and access to the toolset prior to its needed use on a project.

  If possible, users should gain familiarity with a toolset before the demands of the project are felt. There should be some opportunity to experiment with the tool on a prototype or small pilot project before using it on the actual project.

- Bringing users onto a technology transition oriented project such as the ACE project, before their real use of the tool is required, eases the learning process and makes the user more receptive.

The ACE project found that training new users by participating in the ACE project was a very effective way of producing enthusiastic users, but is limited in terms of how many users can be trained this way. Another approach is to seek out receptive individuals who express an interest and enthusiasm for working with new tools and methods. Obviously lead positions need to be filled with such persons.

- A course in how to write a good specification can improve the quality of specifications.

  The ACE project developed a 2-hour model SRS writing course that teaches the fundamentals of "good" SRS writing techniques. This course is taught just before the students are to begin developing real SRSs and the techniques are fresh in mind. A class needs to be timed for optimal effectiveness. If it's given too soon, the motivation may not be there, and retention may be a problem. Waiting too late may overload users with too much last minute information. The lesson is to have material on the shelf ready to go at the optimum time. A good teaching technique is to use illustrated examples of the principles being taught. Continuous process improvement and defect analysis techniques should be applied to a course, after student critiques are received, to improve the course for next time.

FAADS

Description

The FAADS contract, which was started on April 1, 1991, is responsible for the development of hardware and software for a passive ESM system to support tactical forward area defensive weapons platforms in detecting airborne threats and cueing weapons operators. Magnavox is the prime contractor and the IBM Federal Sector Company in San Diego is the software developer.

Although this program is informally known as FAADS, it is actually only a portion of a much larger program, the Forward Area Air Defense System. The specific portion under contract is the AN/VSX-2 program, also known as the Non-Cooperative Target Recognition program, or NCTR-1. IBM is under contract with Magnavox Electronic Systems Company to develop a portion of AN/VSX-2.

The FAADS Software will be developed in two phases, with a Model I consisting of one CSCI developed in the first phase' and a Model II consisting of three CSCIs developed in the second phase. There is planned heavy reuse of Ada code from model I to Model II. The Model I software architecture is based on an existing Ada system consisting of approximately 15 KSLOC of Ada code. Model I software will consist of approximately 12 KSLOC of Ada code. The software will run on 1750A processors; Ada compilations, which were originally being done on a uVax II, are now being performed on the RISC System/6000.

FAADS Hardware/Software Configuration

The FAADS hardware/software configuration is depicted in the figure below.

FAADS Hardware/Software Configuration

```
                    -------     --------  o AIX
                   |PostScr|    -------- | o Teamwork
                   |Printer|---| RISC  || o DocEXPRESS
    -------         -------    | System || o Interleaf
   |Printer|                   | 6000  |  o ObjectMaker (Adagen)
    -------                     --------  o TLD Ada Compiler
       |                           |
       |                       ---------       --------
  ----------                  | LAN     |     |        |     --------
 | Vax      |-----------------|(Token   |-----|  PS/2  |----| 1750   |
  ----------                  | Ring)   |     |        |    | Target |
       |                       ---------       --------      --------
    ------                         |
   |Tape  |                        |
    ------                         |
                               --------
  ---------                   |        |  o DOS
 | IBM     |------------------|  PS/2  |  o MS Windows&OS/2
 | WAN     |                  |        |  o FTP (TCP/IP)
  ---------                    --------   o XVision (XServer)
```

The specific components of configuration are as follows:

- Two RISC System/6000 Model 320s
- Token Ring LAN
- PS/2 Workstations for LAN access
- PostScript Printer
- Teamwork, DocEXPRESS, Interleaf, and Adagen on RISC SYSTEM/6000
- Ada TLD Compiler on RISC SYSTEM/6000 and Vax

FAADS Tool Usage

The STARS Alpha SEE tools currently being used by FAADS are Teamwork, ObjectMaker (Adagen), DocEXPRESS, Interleaf, and PVCS. A member of the FAADS team, who had some prior background in tools and methods, served as a SEE tools/methods consultant.

Teamwork

Originally, IBM planned to use Teamwork/SA for software requirements analysis, but since the prime contractor retained this responsibility, this was not possible. IBM did use Teamwork/SA on a very limited basis to analyze the SSDD (System/Segment Design Specification) and the SRS early in the program. This effort yielded some useful feedback to the prime contractor as to omissions and inconsistencies.

In another change to the original plans, IBM elected to use Teamwork/Ada rather than Adagen to support software design, for these reasons:

- Teamwork had built-in multi-user support
- Teamwork had built-in document production support

- Cadre delivered a new Teamwork module called the Ada Design Sensitive Editor (DSE) which was integrated with the Teamwork/Ada graphical design tool and which supported code generation.

The FAADS software engineers are actually doing their development (i.e., the Ada design and Ada code generation) from within Teamwork/Ada. They are doing the design by creating Buhr diagrams and automatically generating the Ada skeleton code from the diagrams. They are using the DSE to create the detailed Ada code. The DSE does not allow code generated from the diagrams to be changed, without first changing the associated diagram and regenerating the Ada code. This assures that the code and the design documentation are always in synch.

The SDD documentation generation is being done with DocEXPRESS, a third party tool which is integrated within Teamwork. DocEXPRESS uses Teamwork's Documentation Production Interface (DPI) to generate 2167A-compliant documents.

Budgetary constraints prevented formal tool training. To partially offset this deficiency, the following measures were taken:

- Some team members attended the two week Paul Ward Real-time CASE curriculum. Although this class concentrated on method, it did utilize the diagrammatic conventions supported by Teamwork.
- The SEE Consultant (SC) provided a two-day informal hands-on training class, which included an introduction to the Teamwork environment.
- Team members referred to the Teamwork users manuals, including the limited amount of tutorial material.
- The SC circulated among the team and provided case-by-case suggestions.
- On two occasions, Cadre made one-day site visits, which included question and answer sessions and hands-on demonstration sessions.

ObjectMaker (Adagen)

Originally, IBM intended to use Adagen for software design. The reasons for shifting to Teamwork are discussed above.

Since FAADS future phases involved reuse of existing Ada code modules, some experimentation was conducted to evaluate Adagen's reverse engineering capability. The diagrams produced were of limited value by themselves, and since Teamwork had been chosen to support design, no attempt was made to modify them to make them usable. Future experimentation is planned with both Adagen's and Teamwork/Ada's reverse engineering capabilities.

DocEXPRESS

The FAADS documentation strategy called for producing documentation from the Teamwork design model, with minimal additional text publication work. DocEXPRESS supports this strategy by smoothing the interface between the Teamwork Document Production Interface (DPI) and the chosen text publishing software (Interleaf or FrameMaker).

The SC attended a three-day course offered by ATA (the vendor for DocEXPRESS) on 2167A software analysis and design. This course included an introduction to using DocEXPRESS, and was sufficient for the SC to set up tailored support for FAADS. Users required very little understanding of DocEXPRESS, since it was designed to be relatively transparent.

Interleaf

FAADS users had very little need to work within Interleaf, since most of the work was done by Teamwork and DocEXPRESS. None of the team received any relevant training.

The SDD is the only document produced with Interleaf (other project deliverables have been produced using Bookmaster, Word for Windows, and other tools). Interleaf was chosen for the SDD because it was one of two publication systems supported by Teamwork/DPI.

PVCS

PVCS was chosen to support configuration management aspects of the project. PVCS includes support for version management and configuration building. To date, only the latter capability is in use. None of the team received any PVCS training.

FAADS Lessons Learned from Introducing CASE Tools

The lessons learned by the FAADS project from using the IBM STARS Alpha SEE during the S-Increment are described below:

- Significant frontend budget allocation required for training and tools procurement and for installation and maintenance of SEE tools and network.

  Introduction of a new operating system (AIX), tools, and methods required considerable training costs and a significant learning curve for the project team, which are often underestimated in the original budget allocations.

- Strong management commitment and vision essential.

  Management must provide leadership and vision, as new (and often immature) SEE tools and methods are introduced into a project, to ensure that any initial negative (often valid) reactions are overcome and the necessary adaptations are made.

- New project roles required, e.g., system administrator for LAN and AIX, toolsmith for customizing and supporting use of tools.

  This is an especially difficult problem for a small site with limited access to support personnel.

- Single, integrated desktop access to SEE tools important for productivity and use of existing assets.

  A single virtual desktop access to all heterogeneous software/platforms preserves access to familiar tools, while taking advantage of existing site assets (rather than buying additional workstations or X terminals).

- Immaturity of methods (e.g., Buhr notation) and tools in design of large scale systems.

  Many of the notations currently being used for Ada design are relatively immature and evolving. As a result, problems occur when trying to scale up these notations to large systems, e.g., Ada design diagrams become very complex and difficult to understand.

- Immaturity of methods and tools in reverse engineering and reuse.

  Because of its immaturity, reverse engineering is an art, requiring careful tailoring of directives and manual post-processing. In addition, very little training is available on reverse engineering tools.

FAADS Lessons Learned from Using CASE Tools

The lessons learned on the FAADS project from using the IBM STARS Alpha SEE tools/methods are described below:

Teamwork Lessons Learned

- Some Teamwork tools/methods training would have been beneficial.

  Although the training described in the prior section was useful, users failed to pick up numerous time-saving techniques that would have been covered in Cadre's tool training classes. Although it is difficult to quantify, the training might have ended up paying for itself in the long run.

  Even more important than tools training, however, is methods training. Only a few team members received any methods training, and even that training concentrated on the analysis phase -- which was wasted to some extent since IBM ended up not being responsible for the SRS.

  The best possible course would a hybrid method/tool course, in which a specific method is taught using the tool as a hands-on vehicle. A one week Ada Design method course coupled with Teamwork/Ada would have been ideal.

- The value of prior experience.

  As echoed throughout the industry, there is no substitute for prior experience. This holds true with respect to methods, tools, and environments.

  Based on the reports of other projects, IBM had a basic understanding of the impact that would result from the number of changes being introduced for FAADS. These changes included:

  - Migration from a centralized, Vax-based environment to a networked AIX-based environment;
  - Use of several significant new tools;
  - Adaptation of the existing software methods and process to the above.

  The project felt that it would break even, at best, during the incorporation of these changes, but that the investment during the first phase of the contract (FAADS Model I) would result in higher quality for the Model I (which it has) and higher productivity during the subsequent phase (Model II). In retrospect, the impact was underestimated: the number of changes may have been too ambitious for a relatively small project. Unfortunately, it is impossible to quantify the productivity impact, since there have been too many other variables.

  Although the project is not yet complete, it does seem likely that the next phase will realize the improved productivity that was assumed when it was bid. This is due to several factors:

  - Most of the learning curve is over, and the team is acclimated to the adapted software process using the new environment and tools;
  - Many of the problems encountered during the first phase have either been solved or workarounds have been devised;
  - A Teamwork reuse base has been established for the next phase, providing a head-start in developing the three Model II CSCIs;
  - The performance of the Ada compilation system is so much better on the new RISC System/6000 than it was on the previous Vax system that much less time is lost waiting for compilations, simulation runs, and builds.

- Immaturity of Teamwork/Ada and DSE impacted their use.

  Teamwork/Ada, and particularly the DSE, are very new products. In addition, because of their enhanced functionality, the FAADS project opted to introduce these tools during their beta test phases. Both of these facts resulted in significant impacts due to bugs and problems in the use of the tools (e.g., crashes and loss of data).

- Availability of needed resources on configured hardware.

The FAADS project encountered some stability problems because the RISC System/6000 hardware was configured with marginal memory and disk space. Orders placed to remedy this problem could not be filled until late in the program because of high demand for RISC System/6000 hardware. For most of the program, Teamwork was running on machines with only 16MB memory; experience has shown that a minimum of 64 MB is needed for a Teamwork server machine, and that a minimum of 24 MB is needed for a user on a remotely connected RISC System/6000. Currently, the project is using a RISC System/6000 Model 550 for the server machine; this comfortably supports our average of three to four user sessions.

The architecture of the Teamwork product calls for a single model database on a host machine. There are then two methods of using Teamwork from another machine on the network: mounting the Teamwork directories using NFS (and running Teamwork locally), and remotely logging on to the server machine (and using the local machine as an X Server). Using the Model 550 as a host, users have found that the latter method is the most stable; and they pay no response time penalty to use it.

- Teamwork/Ada functionality

  The Teamwork/Ada graphical editor is responsive and robust. The Ada Structure Graph (ASG) notation implemented by the editor (based on the Buhr notation), however, while good at reflecting Ada code structure, has proven insufficient by itself to communicate the software design, and the customer was unhappy with it. The problems were that its notation was unfamiliar and that it does not adequately reflect the following aspects:

  - Data flow

    The notation shows packaging and invocation well, but falls short in showing accesses to data structures, and the functional interfaces among high-level software components.

  - Operational flow

    Using standard ASGs, there is no good way to show how the software components combine to respond to external stimuli (such as an operator action, or the arrival of a signal event).

  To supplement the design documentation, the team added high-level data flows (using Teamwork/SA), and a set of hybrid operational flow diagrams (using Teamwork/Ada).

- Teamwork/DSE functionality

  The users found that the DSE provides significant value added when compared with an ordinary text editor, since it understands Ada syntax. Among other benefits, it automates much of the formatting (reducing keystrokes), identifies syntax errors during text entry, and enhances on-screen readability of the code.

  Teamwork/DSE is integrated with the Teamwork/Ada model and enforces adherence to the design diagrams. Since the SDD documentation is also driven from the same model, this paradigm assures a level of agreement among the documentation, the design model, and the code. Most importantly, using Teamwork/DSE for Ada code development provides the capability to generate both the design documentation and the code from the same design (model) database, helping to assure agreement between them. This is a major strength of using Teamwork/Ada and DSE.

  While problems with the DSE product (bugs and limitations) have hampered full realization of the benefits of the approach, the team generally regards its use as a significant improvement to the Ada development process.

  As of this writing, Cadre has initiated an effort to improve the product. The major problems are:

  - The editor is prone to crashing, causing some loss of data. Users have adopted a practice of saving frequently.

  - Transitioning from the ASG editor to the DSE editor sometimes results in truncation of the Ada source code.

- The "pretty-print" rules adopted by the editor sometimes renders the code much less readable: users want the ability to selectively inhibit the reformatting, or to have more ability to modify the formatting rules.

- Teamwork/DPI Functionality

  Teamwork's Document Production Interface (DPI) is intended to make it possible to generate a document from the Teamwork model database. Most projects suffer from the "multiple, inconsistent databases" problem, and one of Teamwork's central appeals is the advertised ability to produce both documentation and code from a single database. The FAADS team has realized this goal to some extent, but several major problems remain. Some of these problems are addressed by the new Teamwork/DocGEN module, and the team plans to take advantage of its new features when producing the final version of the SDD.

  The following lists the major DPI problem areas:

  - Text Formatting

    DPI offers little capability to affect the formatting of the published text. Ideally, one would like the ability to enter text using the publication software (Interleaf in this case), since this would allow direct entry of lists, tables, etc., and since it would allow complete control over readability techniques such as italics and underscoring. Instead, the user must use Teamwork's rudimentary text editor and resort to a very limited set of DPI formatting commands.

  - Hierarchical Descent of the Software Structure

    DPI provides a powerful "parse_model" command that allows most of the SDD to be constructed automatically from the model. Parse_model starts from a specified node in the software component hierarchy (in this case a specific Ada Structure Graph diagram) and descends the subtree below this point -- embedding text and other pictures that are attached as notes to the ASG diagrams. The user controls the manner in which these notes are embedded using a format specification file. Unfortunately, there are several notable limitations:

    - ▲ At each level of the hierarchy, the diagrams are introduced in alphabetical order, according to the ASG diagram names; this is almost never the best order in which to introduce them from an understandability standpoint.

    - ▲ Once a parse_model descent is started, it continues to the bottom-most points in the hierarchy. This fact makes it very difficult to adhere to the 2167A DID for the SDD, which calls for a top-level depiction of the software in Section 3 and an intermediate- to low-level depiction in Section 4. This would seem to dictate two different design models, one for the high-level design and one for low-level design. The team decided to stick with the one-database philosophy, but this decision carried with it the need to develop some workarounds to the DPI limitations.

      Unfortunately, the decision also affected the CSC/CSU design structure. In Section 3, the SDD should decompose the software down to the CSC level, but DPI provides no way to cut the parse_model descent short. As a result, Section 3 descends down to the CSU level.

    - ▲ There is no provision for parsing the model multiple times in a single document with different formatting rules. The team was able to develop a work-around for this limitation, but it should be a part of the product.

    Unfortunately, none of the parse_model limitations are addressed in the new DocGEN product component.

  As of this writing the new DocGEN facility is available, and the team plans to take advantage of several of its features to improve the final version of the document:

  - New formatting features are now available, including the ability to construct lists and tables.

- It is now possible to compare earlier versions of documents with newer ones and generate a new document with change bars.

- Model Configuration Management (MCM) functionality

  Teamwork includes some basic CM features, retention of the past 16 versions of each model object, multi-user checkin/checkout of model objects, and the ability to baseline models (and to construct "derivative" models where modifications are separately maintained). These features have proved useful as far as they go: users have been able to work effectively as a team without worrying about losing data due to conflicts.

  Some improvements are needed, however, to fully enable the "single database" strategy (production of documents and code from a single Teamwork model). A standard software CM practice is to control software products in a hierarchical library structure, with higher levels containing previously released software and lower levels containing incremental changes awaiting testing so that they can move to higher levels of control. With this strategy, a method is needed to "promote" components from lower library levels into the higher ones. With conventional files, this can be accomplished by file moves from one directory to another (with suitable controls, of course). Similar functionality is provided with commercial CM tools such as PVCS and CCC.

  Ideally, the project should be able to manage the Teamwork model in the same way so that the Ada code levels can be kept in perfect sync with the corresponding model levels. Unfortunately, MCM offers no way to implement the "promote" function.

  Cadre has been receptive to this problem and is considering ways to address it. Another vendor, Softool, is currently working to tailor its product to Teamwork to provide this functionality in its commercial CM product, CCC.

ObjectMaker (Adagen) Lessons Learned

As mentioned earlier, the project originally intended to use Adagen to support the design process. As Cadre added Teamwork/Ada, and later DSE, the strategy changed to use Teamwork instead of Adagen. As a result, FAADS gained little experience in using Adagen, other than some initial experimentation with its capability to reverse engineer Ada code. It should be noted that Adagen has gone through significant improvements since FAADS early use of it.

DocEXPRESS Lessons Learned

- DocEXPRESS Functionality

  DocEXPRESS simplifies the transition between Teamwork/DPI and publication software (Interleaf or FrameMaker) -- with specific support for producing 2167A documentation. It provides:

  - Additional Teamwork menus for building documents (by itself, DPI must be invoked outside of Teamwork);
  - Predefined templates including boilerplate, to give a headstart for the standard 2167A documents;
  - Consistent formatting among documents, conforming to DID standards as to headings, section and page numbering, etc.

  Unfortunately, DocEXPRESS executes on top of Teamwork/DPI and inherits underlying limitations of that tool. It should be noted that substantial improvements in both DocEXPRESS and Teamwork document generation have been made since the time of this experience. Even with these improvements, however, generation of deliverable-quality documentation remains a significant challenge.

- Doc EXPRESS Documentation and Support

  The users manual is very clear and provides specific, detailed usage instructions.

One of the biggest advantages of using the product is the excellent support offered by the vendor, ATA. ATA personnel have a significant experience base in systems, and specifically in 2167 and 2167A systems. They are also intimately familiar with both Teamwork and the text publication software.

Armed with this experience, they were very responsive in providing product support, including advice on methodology.

Interleaf Lessons Learned

- Publishability of Documents

  In accordance with the "single database" strategy, the team spent minimal time using Interleaf. As discussed elsewhere, however, due to limitations of DPI, the resulting SDD was marginally publishable. It would have been possible to greatly enhance the appearance of the document by using the considerable power of Interleaf, but to do so would have meant introducing multiple databases.

- Tool Integration

  Ideally, tools for modeling (e.g., Teamwork) and documenting (e.g., Interleaf) should be more closely integrated. Several vendors are pursuing this concept. One possible approach would be to allow the Teamwork user to use the publication software while entering text. Another approach would be to give the user transparent navigation between the publication software and the modeling tool.

- Licensing

  There are two means of licensing Interleaf: networked and node-locked. With the networked method, Interleaf can run anywhere on the network, but only a maximum number of users can run it at one time. With the node-locked method, Interleaf only runs on one machine, but any number of users can use it at one time. The cost of the node-locked license is the same as a one-user networked license.

  Because Interleaf is only used for the SDD, and because one user is assigned the task of building the document, it is rare that more than one user is accessing Interleaf at once. Hence, the node-locked method has proven more economical for FAADS.

  Interleaf is installed on the server machine. Users on remote nodes can connect to the product via remote logon, using their local machines as X Servers. Given the limited need for interactive use, response time using this approach has proven adequate.

PVCS Lessons Learned

- Version Manager

  As of this report, the project has not yet incorporated the Version Manager. The site has an established practice of using a multi-level library scheme for controlling incremental software releases. With this scheme, all files are separately maintained, and promotions to higher library levels are accomplished by moving integral files.

  In contrast, the PVCS Version Manager uses monolithic files to store multiple versions of each software component. This method represents a significant departure for the organization, and its incorporation is still under study. Advantages of using the Version Manager would include:

  - More concise storage of multiple versions of source files by using "delta" techniques; and
  - Opportunity to store all archived versions on-line (because of the reduced storage requirements) and to readily reconstruct past archived configurations.

- Configuration Builder

The PVCS Configuration Builder is based on the Unix make, with additional enhancements. PVCS has enabled straightforward automation of the software compilation and build process, including the handling of multi-level libraries.

If the project elects to move to the Version Manager (see above), PVCS includes provisions for integrating the Configuration Builder capabilities with minimal effort.

- Integration

  PVCS by itself provides no capability for integrating Teamwork model artifacts with the code artifacts (the need for doing this has been discussed earlier, in the topic of Teamwork's Model Configuration Management facilities).

  Another vendor, Softool, is developing integrated support for Teamwork and code for its CM tool, CCC.

- Multi-Platform Considerations

  One of the FAADS software CSCs is being implemented in the C language, and it is being developed on an IBM PS/2 since there is no appropriate RISC System/6000 compiler for the processor on which the CSC executes. Although it is desirable to have a single CM database for the entire project, this separation of platforms poses problems.

  At present, the PS/2 software construction process is segregated from the RISC System/6000 process for Ada code. The project is assessing the possibility of integrating the two processes, using NFS on the PS/2 to mount the RISC System/6000 code libraries.

  Should the project also adopt the PVCS Version Manager, Intersolv markets a networked version of PVCS for the PS/2 which would allow common usage of the PVCS database from both platform types.

Miscellaneous Lessons Learned

- The impact of introducing multiple changes/technologies was underestimated (the number of changes may have been too ambitious for a relatively small project). These changes included:
  - Migration from a centralized, Vax-based environment to a networked AIX-based environment.
  - Use of several significant new tools, e.g., Ada, Teamwork, Interleaf.
  - Adaptation of the existing software methods and process to the above.

- One of the greatest performance improvements accrued from upgrading the SEE CPU "horsepower":
  - Ada compilations
  - System builds
  - Test runs using the target computer emulator

- Higher quality was realized in the first phase of the contract (FAADS Model I) and higher productivity is expected for subsequent phase (Model II), due to:
  - Most of the learning curve is over, and the team is acclimated to the adapted software process using the new environment and tools
  - Many of the problems encountered during the first phase have either been solved or workarounds have been devised
  - A Teamwork reuse base has been established for the next phase, providing a head-start in developing the three Model II CSCIs

Summary of Combined Lessons Learned on STARS Alpha Test Projects

In this concluding section, we reflect on all three projects and attempt to distill some of the common lessons learned.

Impediments to Change/Remedial Strategies

All of the projects were groundbreakers, and (to push the metaphor a bit) they all encountered boulders as they were getting underway. The following impediments are representative of the type that may be encountered on any similar project attempting to inject new SEE approaches. For each category, we include some constructive ideas on how a project might attempt to prepare for and offset these impediments.

- Problem: Inertia
  - People are comfortable with the existing process
  - There is a tendency to subvert new methods to old ways of thinking
  - Attitude is all-important

  Strategies:
  - Insure strong support, vision from management and tech leads
  - Enlist early support, involvement from customer
  - Involve people in planning, preparations
  - Develop phased implementation plan, tailored to group
  - Consider formal Technology Transition training
- Problem: Overblown Expectations
  - Marketing hype, overzealous advocates are common
  - Unrealistic hopes lead to disillusionment
  - Unanticipated costs can blossom

  Strategies:
  - Interview teams with real-project experience
  - Try out SEE, tools, methods, process on pilot project
  - Carefully weigh degree of change against cost uncertainties
  - Explicitly plan for each cost category (see checklists, below)
  - Expect no productivity increase on first system

- Avoid "panacea mentality":
    - ▲ New methods and tools are no substitute for domain expertise
    - ▲ Poorly thought-out SEE strategy can degrade effectiveness
- Key: Match Resources to Ambitions
    - Small projects should focus on incremental change
    - Large projects can handle more change, but only with careful planning

Combined Lessons Learned: Planning

Because of the potential risks in introducing new approaches, and because it's vital to get off to as sound a start as possible, planning is especially important. Here are some of the considerations to take into account when planning the project.

- Anticipate Essential Startup Activities
    - Clear identification of methods
    - Methods training
    - Tool evaluation, selection
    - Tool adaptation/integration design & implementation
    - Tool training
- Assess Cost of SEE Realistically, including:
    - H/W components and networking
        - ▲ Consider wiring, installation, checkout, support, maintenance
        - ▲ Insure necessary computing resources & seats to deliver tool capability to users with adequate response time
        - ▲ If possible, install trial configuration before final planning
    - S/W
        - ▲ Be sure number of licenses will support planned roles
        - ▲ Don't forget software maintenance (typically 15%/yr)
    - Adaptation and Integration Expenses

      Note: This can turn out to be extensive:
        - ▲ Each tool typically requires tailoring to fit process
        - ▲ Varying degrees of integration required for tools to work together
        - ▲ Databases typically stretch across heterogeneous platforms
        - ▲ Requirements will continue to emerge throughout project
    - Administration: H/W, Networking, and Tools
    - Plan for Ongoing Roles, such as:
        - ▲ SEE and tool administrators
        - ▲ Adaptation & integration evolution and support

- ▲ Methodology consultants (motivated, knowledgeable, proactive)

- Involve the Customer Early
  - ▲ Agree on approach and required mutual investment
  - ▲ Consider including customers in training sessions
  - ▲ Tailor documentation/deliverable plan (e.g., 2167A)
- Plan for Balanced Learning Approach
  - ▲ Classes (methods, tools, SEE)

    (Note: "Just-in-time" Training is most effective)
  - ▲ Domain-specific workshops, with expert consulting
  - ▲ Pilot project
  - ▲ Hands-on Experience

Combined Lessons Learned: Maintain a Healthy Respect for Murphy's Law

When introducing a significant amount of new technology into a project, it is definitely not the time to utter the phrase "...now, if everything goes well...". The more opportunities for the unexpected to arise, the more opportunities for things to go wrong. Each of the three Alpha projects found ample proof of this principle, and the following examples are cited to provide a flavor.

- Be Wary of Beta Test Versions and Initial Releases

  When you find out the limitations of a tool you've decided to use, you may be anxious to get the next version, perhaps even a Beta (or even an Alpha?). Bear in mind, however, that the inevitable bugs will compound the group's problems of assimilation. If things are bad enough, it might kill the initiative before it gets a real start. Before yielding to the temptation to get the latest and greatest, consider the vendor's past quality record.

- "You only know what you're in for when you're in it"

  This point cannot be overemphasized. Students of calculus often find out the hard way that you don't really learn the subject until you do the problems, and the same principle applies to learning to apply new SEE approaches. All three of the projects rediscovered this as they found that what seemed so smooth in the visionary's pitch (or the vendor's sales literature) had lots of bumps in practice. Experiences from FAAD/NCTR1 are offered as illustrations. These are given without elaboration; interested readers are invited to contact the authors for specifics.

  - Bugs & Kinks

    (e.g., Transition from Teamwork[1] graphical editor to DSE editor sometimes resulted in truncated Ada code files)
  - Gaps

    (e.g., Buhr notation not sufficient for representing the design. This point is discussed in more depth in the next subsection.)
  - Misapprehension of function

[1] Please note: these problems are not meant as criticisms of Teamwork; rather, they are meant to illustrate the problems that can be expected from any set of tools.

(e.g., Built-in Teamwork CM did not mesh with the organization's hierarchical library process)

- Integration with other parts of the SEE

  (e.g., Project legacy documents in Bookmaster, new documents in Interleaf)

Combined Lessons Learned: Technical Tidbits

This subsection lists some of the more interesting and salient technical points that might turn out to be of practical value to new projects.

- Key Objective: Retain Currency of Design as Code Evolves
  - GPS/ACE strategy: PDL maintained in Rational code; diagrams produced from code via reverse engineering
  - FAADS strategy: Teamwork/Ada & DSE: models & code kept in lockstep
- Key Objective: Provide User with Single Desktop Access to SEE Assets
  - GPS, ACE used RISC System/6000 workstations and PS/2s
  - FAADS supplemented workstations with PS/2 with MS-Windows X Server
- Methods Lessons Learned
  - Buhr notation (as per Teamwork/Ada) not sufficient for design
    - ▲ Describes static Ada structure
    - ▲ Additional diagrams needed for
      - △ Interfaces and dynamic behavior
      - △ Operational flows in response to usage scenarios
  - Reverse Engineering Requires Manual Assistance
  - Making diagrams readable
  - Discovering and reflecting interfaces and dynamic behavior
- Documentation Consistently Proved Harder to Produce than Expected

Combined Lessons Learned: Potential Rewards

This section has thus far emphasized caution, and the reader may have begun to conclude that the authors are against the introduction of change. On the contrary, we believe that despite growing pains, the Alpha Projects have shown that the future of automated support of the software process is very promising.

To help make this point, this final subsection is devoted to one of the key positive lessons learned: that there are substantial potential rewards for an organization that manages to weave its way through the obstacle course without crashing.

- Significant Morale Boost
  - Upgraded technology = = > upgraded skills
  - Willingness of management to invest in improving workplace
- More Effective New Process

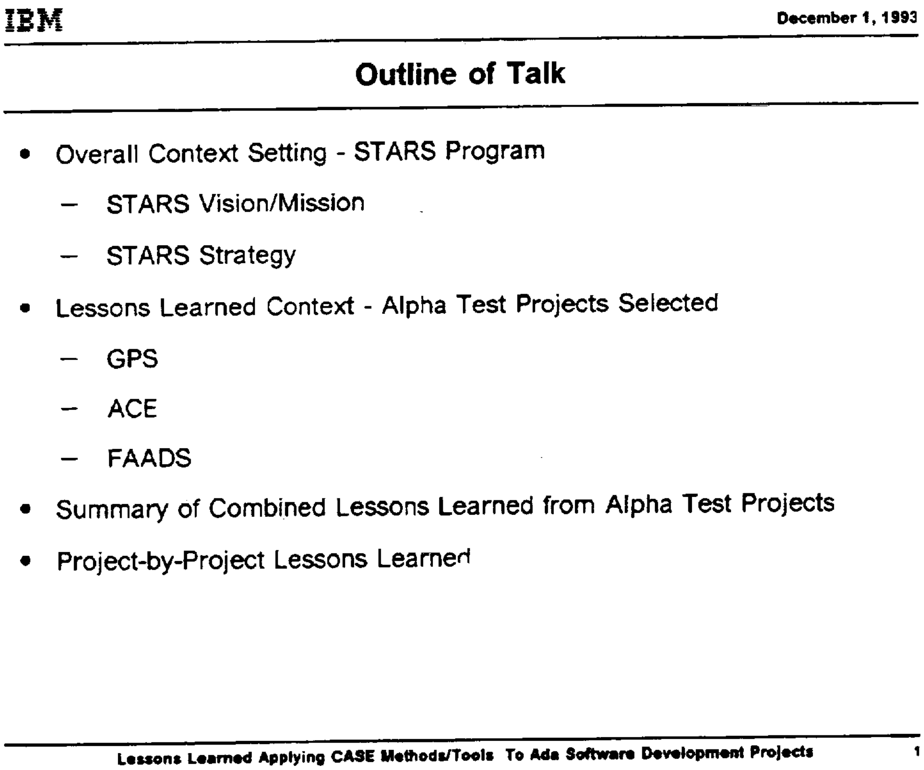
- Better team communication/coordination
- Higher individual and team productivity
- Better quality work products

18th Annual Software Enginnering Workshop
Lessons Learned Applying CASE Methods/Tools To Ada Software Development Projects

December 1, 1993

Maurice H. Blumberg
(301)240-6018
blumberm@wmavm7.vnet.ibm.com
Dr. Richard L. Randall
(719)554-6597
randallr@wmavm7.vnet.ibm.com

STARS Project
IBM Federal Systems Company
800 N. Frederick Ave.
Gaithersburg, Md. 20879

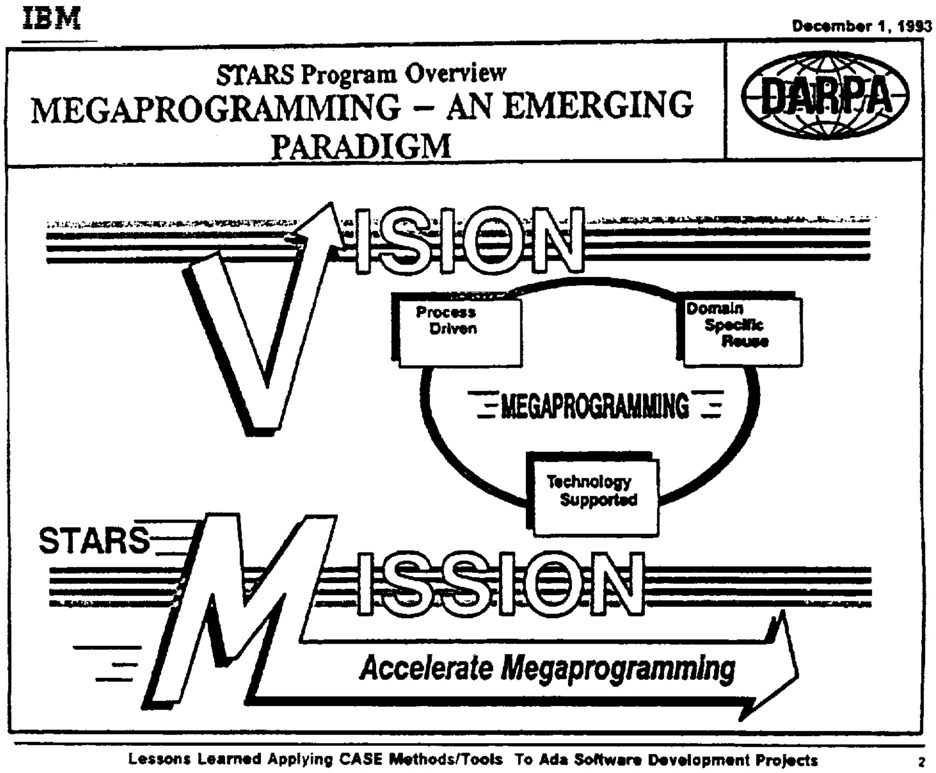


IBM December 1, 1993

Outline of Talk

- Overall Context Setting - STARS Program
  - STARS Vision/Mission
  - STARS Strategy
- Lessons Learned Context - Alpha Test Projects Selected
  - GPS
  - ACE
  - FAADS
- Summary of Combined Lessons Learned from Alpha Test Projects
- Project-by-Project Lessons Learned

Lessons Learned Applying CASE Methods/Tools To Ada Software Development Projects 1



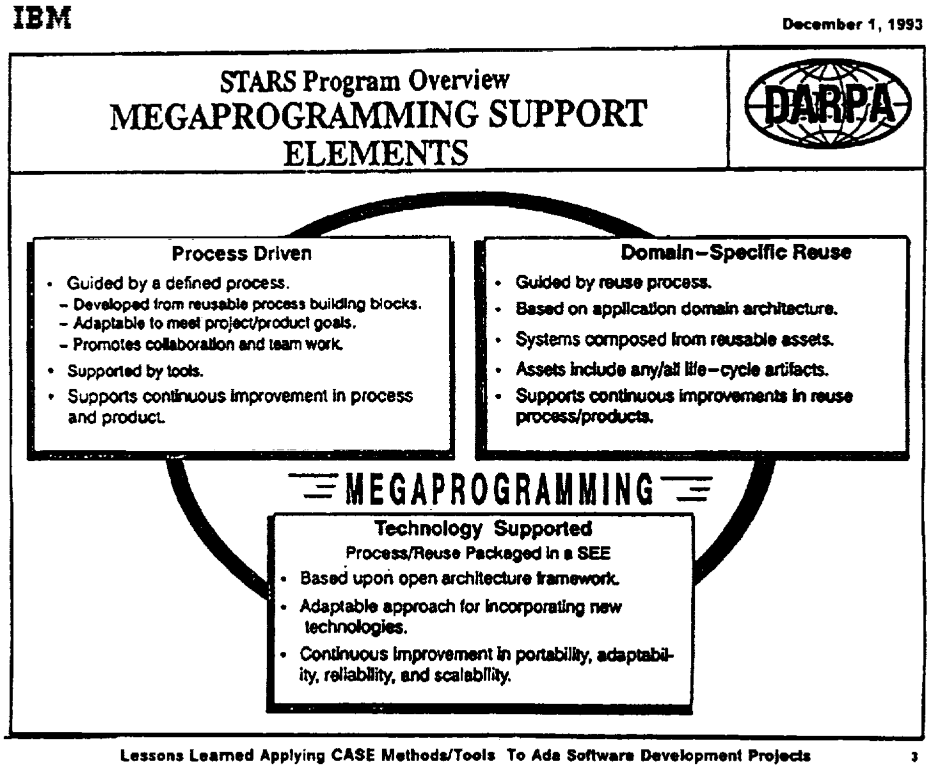
IBM

December 1, 1993

STARS Program Overview

MEGAPROGRAMMING – AN EMERGING PARADIGM

Lessons Learned Applying CASE Methods/Tools To Ada Software Development Projects 2



IBM

December 1, 1993

STARS Program Overview

MEGAPROGRAMMING SUPPORT ELEMENTS

Process Driven

- Guided by a defined process.
  - Developed from reusable process building blocks.
  - Adaptable to meet project/product goals.
  - Promotes collaboration and team work.
- Supported by tools.
- Supports continuous improvement in process and product.

Domain–Specific Reuse

- Guided by reuse process.
- Based on application domain architecture.
- Systems composed from reusable assets.
- Assets include any/all life–cycle artifacts.
- Supports continuous improvements in reuse process/products.

Technology Supported

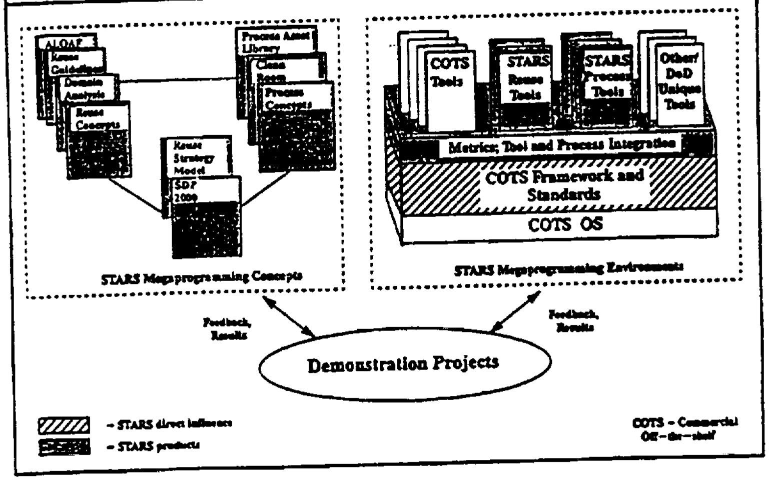
Process/Reuse Packaged in a SEE

- Based upon open architecture framework.
- Adaptable approach for incorporating new technologies.
- Continuous improvement in portability, adaptability, reliability, and scalability.

Lessons Learned Applying CASE Methods/Tools To Ada Software Development Projects 3

STARS Update

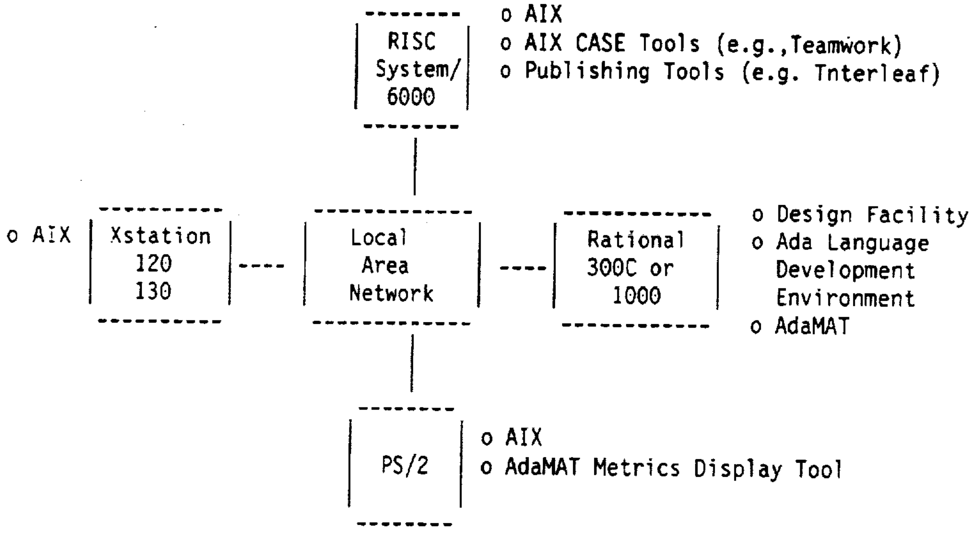
STARS STRATEGY AND PRODUCTS



Purpose of Alpha Test Projects

- Gain early experience and feedback in the use of the IBM STARS Alpha SEE Solutions
- Provide vehicle for early technology transfer of IBM STARS capabilities
- Be a precursor for STARS Demonstration Projects in defining:
  - A technology transfer process
  - How to support projects using a SEE
  - How to capture lessons learned information

IBM STARS Alpha SEE Solution



RISC System/ 6000
- o AIX
- o AIX CASE Tools (e.g.,Teamwork)
- o Publishing Tools (e.g. Tnterleaf)

o AIX

Xstation 120 130

Local Area Network

Rational 300C or 1000
- o Design Facility
- o Ada Language Development Environment
- o AdaMAT

PS/2
- o AIX
- o AdaMAT Metrics Display Tool

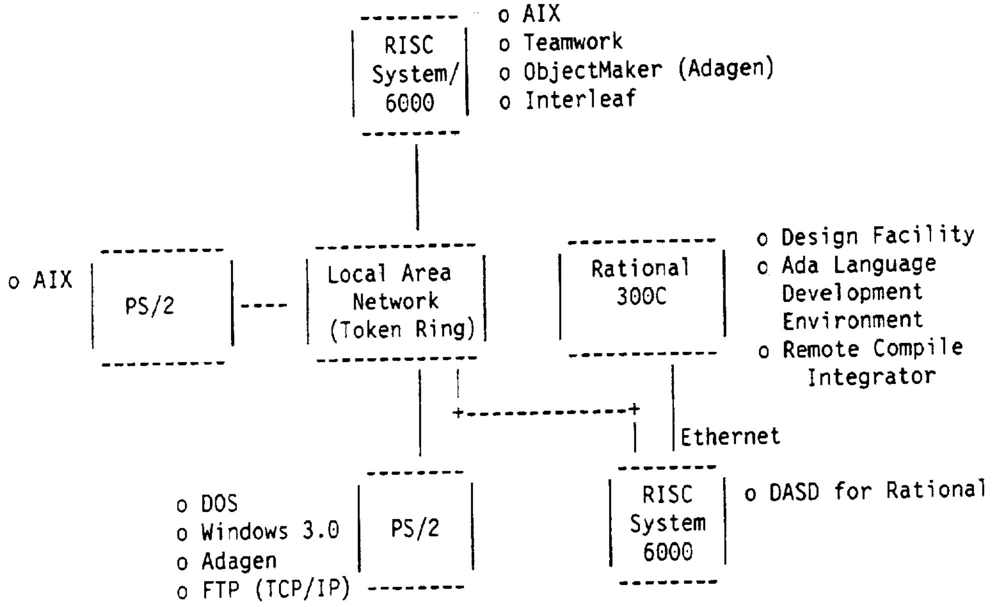
Current Alpha Test Projects

- Global Positioning System (GPS)
- Ada CASE Engineering (ACE)
- Forward Area Air Defense (FAAD) Electronic Support Measure (ESM) Non-cooperative Target Recognition (NCTR) System (FAADS)

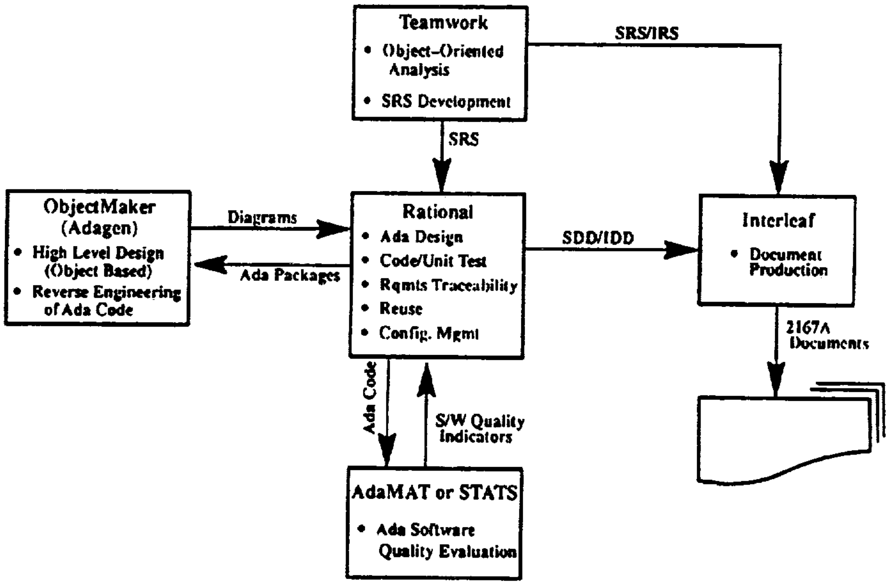
Alpha Test Projects SEE Tool Usage

Life Cycle Activity	GPS	ACE	FAADS
Analysis	Teamwork	Teamwork STATEMATE	N/A
Design	Adagen Rational	Adagen Rational	Teamwork/ Ada & DSE
Implementation	Rational	Rational	TLD Ada Compiler
Document Generation	Teamwork DocEXPRESS Interleaf Rat.	Rational	Teamwork DocEXPRESS Interleaf
Reverse Engineering	Adagen	Adagen	Adagen
Requirements Traceability	Rational	Rqt RTrace RTM Rat.	Manual

GPS Hardware/Software Configuration



GPS Ada Design / Development Environment



Teamwork
- Object–Oriented Analysis
- SRS Development

SRS/IRS

SRS

ObjectMaker (Adagen)
- High Level Design (Object Based)
- Reverse Engineering of Ada Code

Diagrams

Ada Packages

Rational
- Ada Design
- Code/Unit Test
- Rqmts Traceability
- Reuse
- Config. Mgmt

SDD/IDD

Interleaf
- Document Production

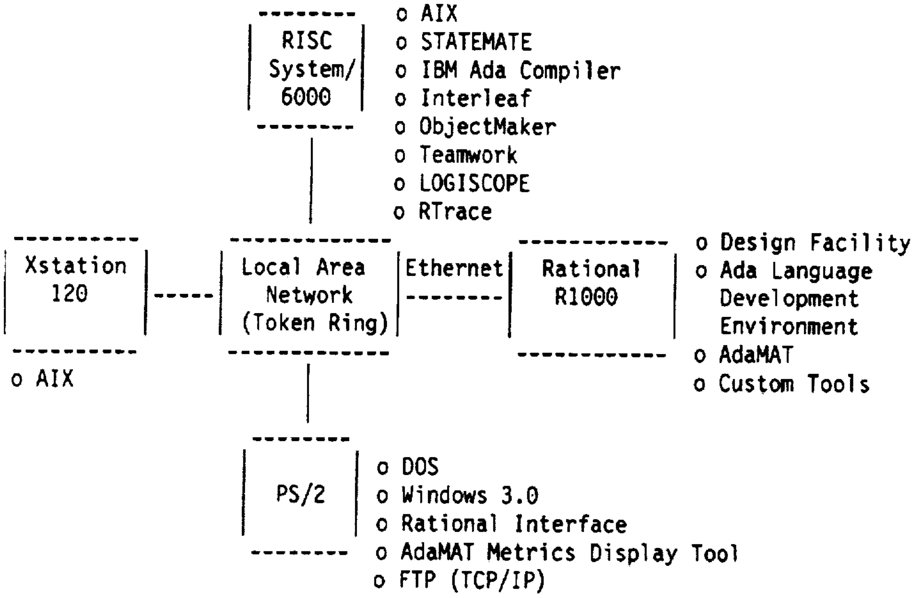
2167A Documents

Ada Code

S/W Quality Indicators

AdaMAT or STATS
- Ada Software Quality Evaluation

ACE SEE



RISC System/6000
- o AIX
- o STATEMATE
- o IBM Ada Compiler
- o Interleaf
- o ObjectMaker
- o Teamwork
- o LOGISCOPE
- o RTrace

Xstation 120
- o AIX

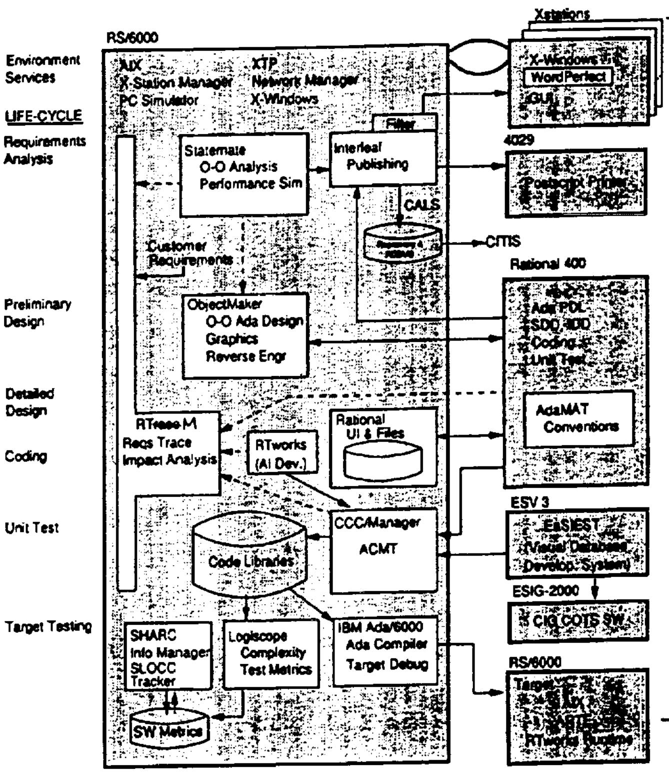
Local Area Network (Token Ring)

Ethernet

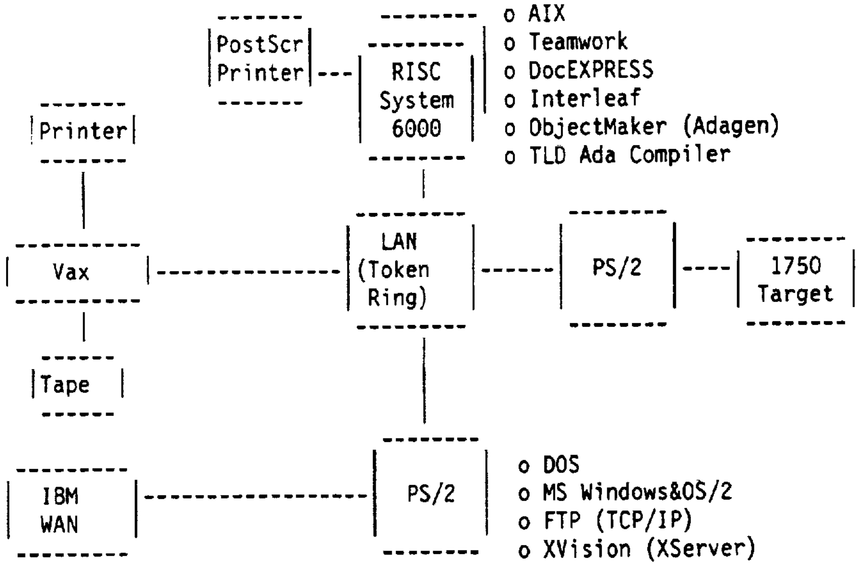
Rational R1000
- o Design Facility
- o Ada Language Development Environment
- o AdaMAT
- o Custom Tools

PS/2
- o DOS
- o Windows 3.0
- o Rational Interface
- o AdaMAT Metrics Display Tool
- o FTP (TCP/IP)

CCTT SOFTWARE DEVELOPMENT ENVIRONMENT

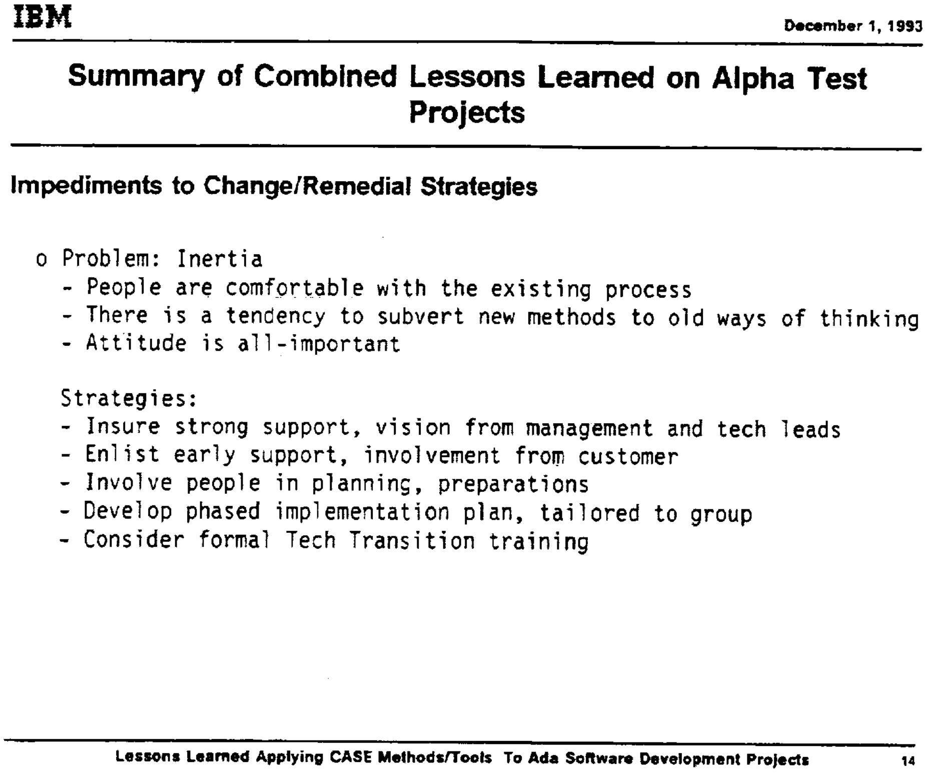


FAADS SEE



o AIX
o Teamwork
o DocEXPRESS
o Interleaf
o ObjectMaker (Adagen)
o TLD Ada Compiler

o DOS
o MS Windows&OS/2
o FTP (TCP/IP)
o XVision (XServer)

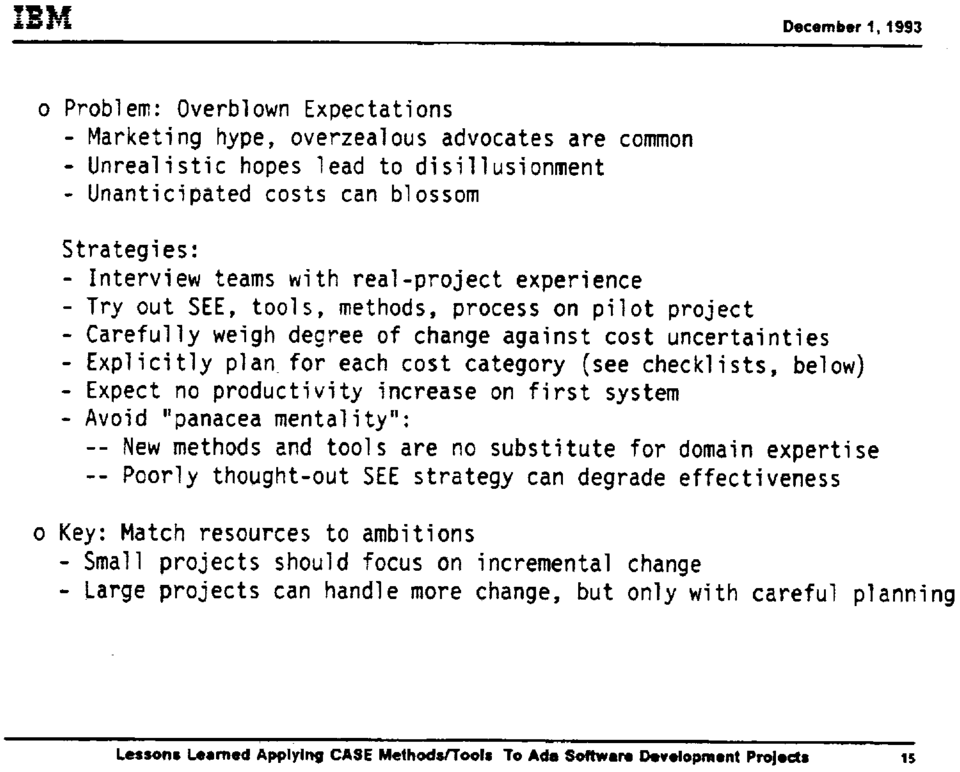


Summary of Combined Lessons Learned on Alpha Test Projects

Impediments to Change/Remedial Strategies

o Problem: Inertia
- People are comfortable with the existing process
- There is a tendency to subvert new methods to old ways of thinking
- Attitude is all-important

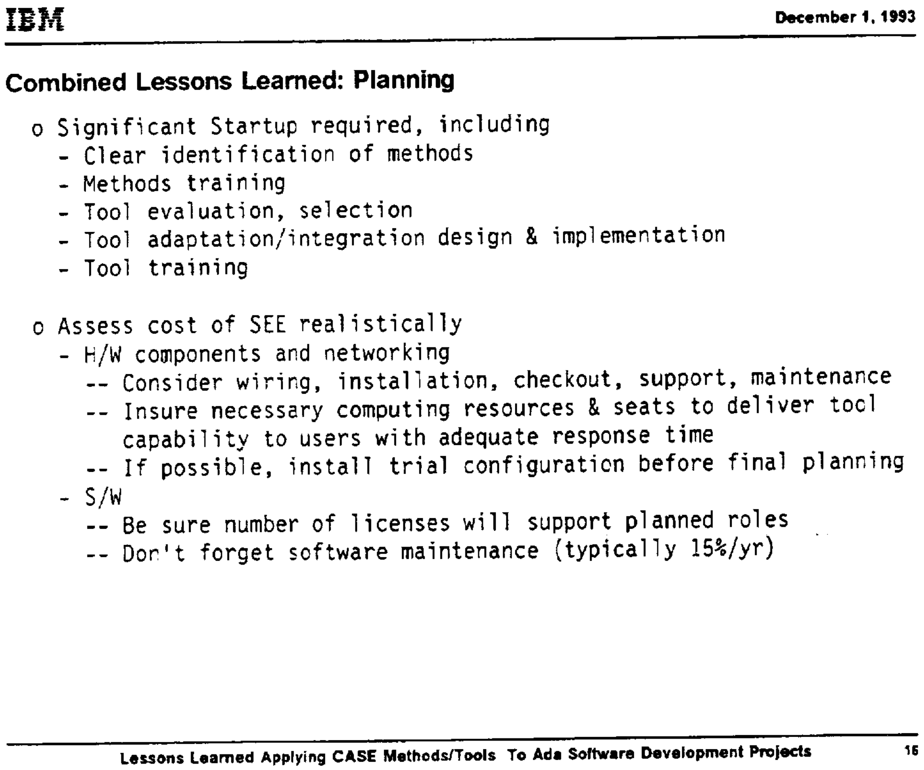
Strategies:
- Insure strong support, vision from management and tech leads
- Enlist early support, involvement from customer
- Involve people in planning, preparations
- Develop phased implementation plan, tailored to group
- Consider formal Tech Transition training



o Problem: Overblown Expectations
- Marketing hype, overzealous advocates are common
- Unrealistic hopes lead to disillusionment
- Unanticipated costs can blossom

Strategies:
- Interview teams with real-project experience
- Try out SEE, tools, methods, process on pilot project
- Carefully weigh degree of change against cost uncertainties
- Explicitly plan for each cost category (see checklists, below)
- Expect no productivity increase on first system
- Avoid "panacea mentality":
  - -- New methods and tools are no substitute for domain expertise
  - -- Poorly thought-out SEE strategy can degrade effectiveness

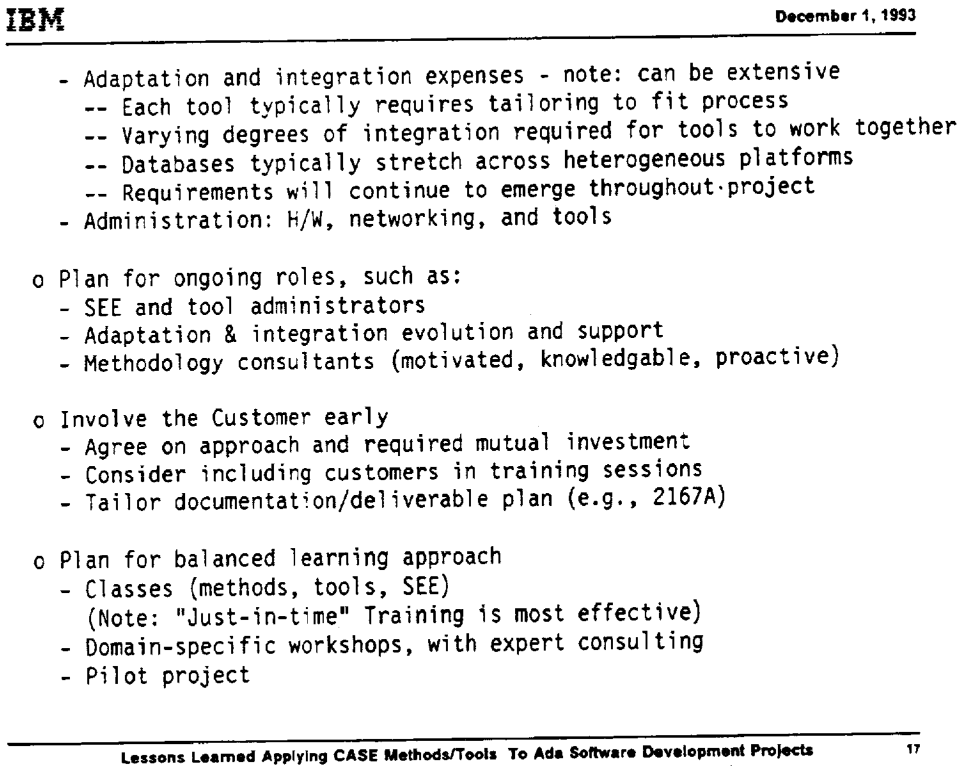
o Key: Match resources to ambitions
- Small projects should focus on incremental change
- Large projects can handle more change, but only with careful planning



Combined Lessons Learned: Planning

- o Significant Startup required, including
  - Clear identification of methods
  - Methods training
  - Tool evaluation, selection
  - Tool adaptation/integration design & implementation
  - Tool training

- o Assess cost of SEE realistically
  - H/W components and networking
    - -- Consider wiring, installation, checkout, support, maintenance
    - -- Insure necessary computing resources & seats to deliver tool capability to users with adequate response time
    - -- If possible, install trial configuration before final planning
  - S/W
    - -- Be sure number of licenses will support planned roles
    - -- Don't forget software maintenance (typically 15%/yr)



- Adaptation and integration expenses - note: can be extensive
  - -- Each tool typically requires tailoring to fit process
  - -- Varying degrees of integration required for tools to work together
  - -- Databases typically stretch across heterogeneous platforms
  - -- Requirements will continue to emerge throughout project
- Administration: H/W, networking, and tools

- o Plan for ongoing roles, such as:
  - SEE and tool administrators
  - Adaptation & integration evolution and support
  - Methodology consultants (motivated, knowledgable, proactive)

- o Involve the Customer early
  - Agree on approach and required mutual investment
  - Consider including customers in training sessions
  - Tailor documentation/deliverable plan (e.g., 2167A)

- o Plan for balanced learning approach
  - Classes (methods, tools, SEE)
    (Note: "Just-in-time" Training is most effective)
  - Domain-specific workshops, with expert consulting
  - Pilot project

Combined Lessons Learned: Maintain a Healthy Respect for Murphy's Law

- Be wary of Beta test versions and initial releases
- "You only know what you're in for when you're in it"

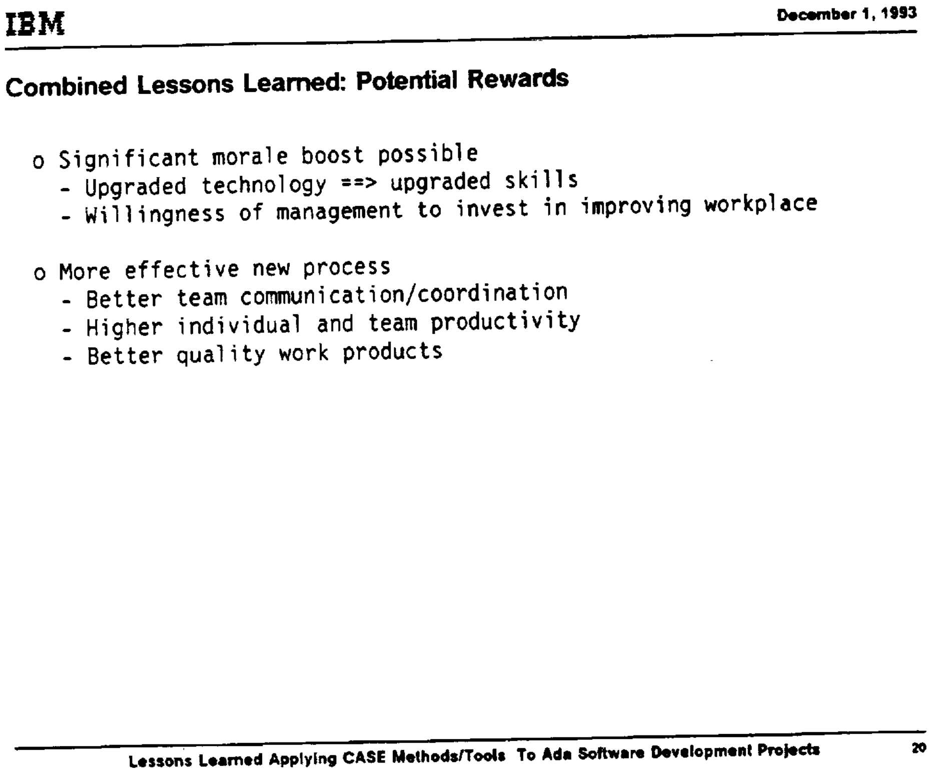
  Examples from FAAD/NCTR1:
  - Bugs & Kinks
    (e.g., Transition from Teamwork* graphical editor to DSE editor sometimes resulted in truncated Ada code files)
  - Gaps
    (e.g., Buhr notation not sufficient for representing the design)
  - Misapprehension of function
    (e.g., Built-in Teamwork CM did not mesh with the organization's hierarchical library process)
  - Integration with other parts of the SEE
    (e.g., Project legacy documents in Bookmaster, new documents in Interleaf)

----

* Please note: these problems are not meant as criticisms of Teamwork; rather, they are meant to illustrate the problems that can be expected from any set of tools.

Combined Lessons Learned: Technical Tidbits

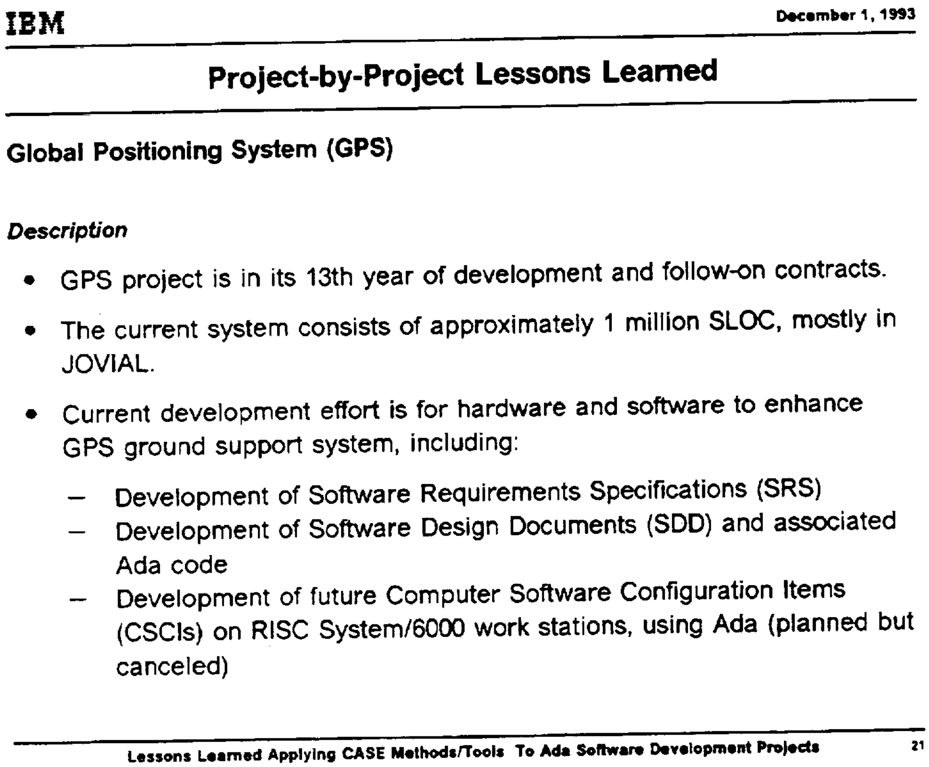
- Key objective: retain currency of design as code evolves
  - GPS/ACE strategy: PDL maintained in Rational code; diagrams produced from code via reverse engineering
  - FAADS strategy: Teamwork/Ada & DSE: models & code kept in lockstep
- Key objective: provide user with single desktop access to SEE assets
  - GPS, ACE used RISC System/6000 workstations and PS/2s
  - FAADS supplemented workstations with PS/2 with MS-Windows X Server
- Methods Lessons Learned
  - Buhr notation (as per Teamwork/Ada) not sufficient for design
    - Describes static Ada structure
    - Additional diagrams needed for
      - Interfaces and dynamic behavior
      - Operational flows in response to usage scenarios
  - Reverse engineering requires manual assistance
    - Making diagrams readable
    - Discovering and reflecting interfaces and dynamic behavior
- Documentation consistently proved harder to produce than expected



Combined Lessons Learned: Potential Rewards

o Significant morale boost possible
  - Upgraded technology ==> upgraded skills
  - Willingness of management to invest in improving workplace

o More effective new process
  - Better team communication/coordination
  - Higher individual and team productivity
  - Better quality work products

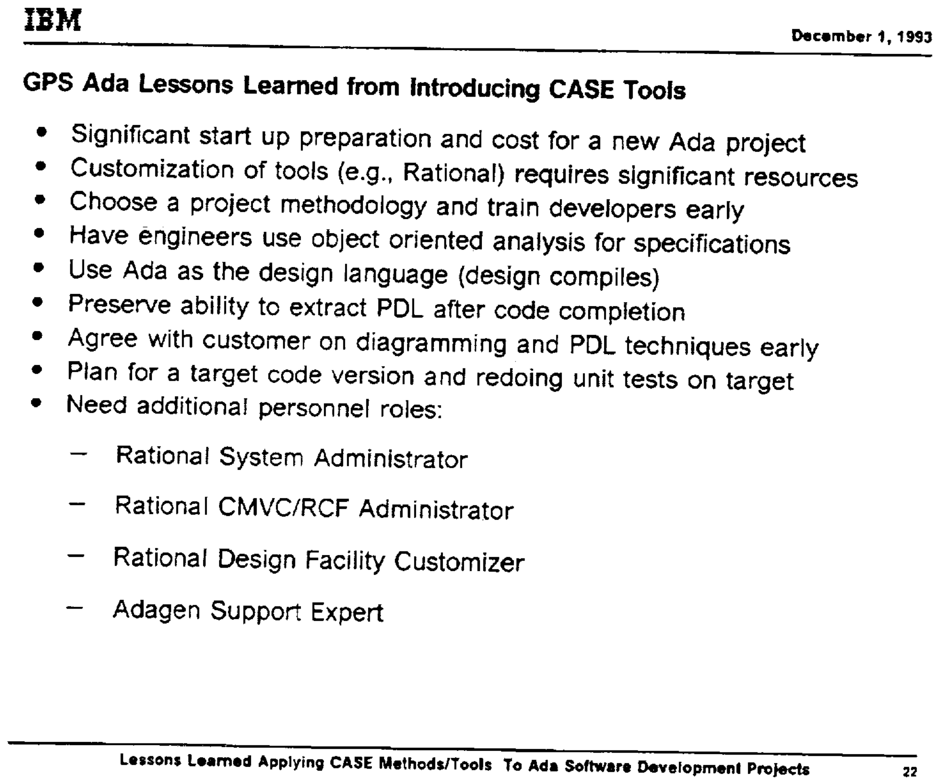


Project-by-Project Lessons Learned

Global Positioning System (GPS)

Description

- GPS project is in its 13th year of development and follow-on contracts.
- The current system consists of approximately 1 million SLOC, mostly in JOVIAL.
- Current development effort is for hardware and software to enhance GPS ground support system, including:
  - Development of Software Requirements Specifications (SRS)
  - Development of Software Design Documents (SDD) and associated Ada code
  - Development of future Computer Software Configuration Items (CSCIs) on RISC System/6000 work stations, using Ada (planned but canceled)



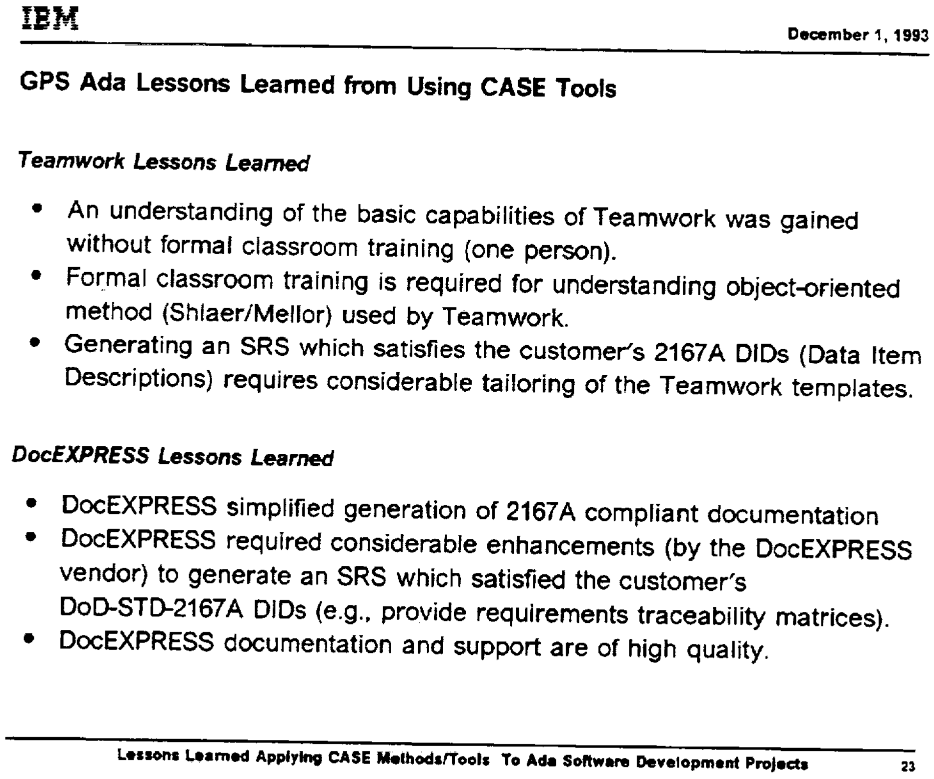
IBM

December 1, 1993

GPS Ada Lessons Learned from Introducing CASE Tools

- Significant start up preparation and cost for a new Ada project
- Customization of tools (e.g., Rational) requires significant resources
- Choose a project methodology and train developers early
- Have engineers use object oriented analysis for specifications
- Use Ada as the design language (design compiles)
- Preserve ability to extract PDL after code completion
- Agree with customer on diagramming and PDL techniques early
- Plan for a target code version and redoing unit tests on target
- Need additional personnel roles:
  - Rational System Administrator
  - Rational CMVC/RCF Administrator
  - Rational Design Facility Customizer
  - Adagen Support Expert

Lessons Learned Applying CASE Methods/Tools To Ada Software Development Projects 22



IBM

December 1, 1993

GPS Ada Lessons Learned from Using CASE Tools

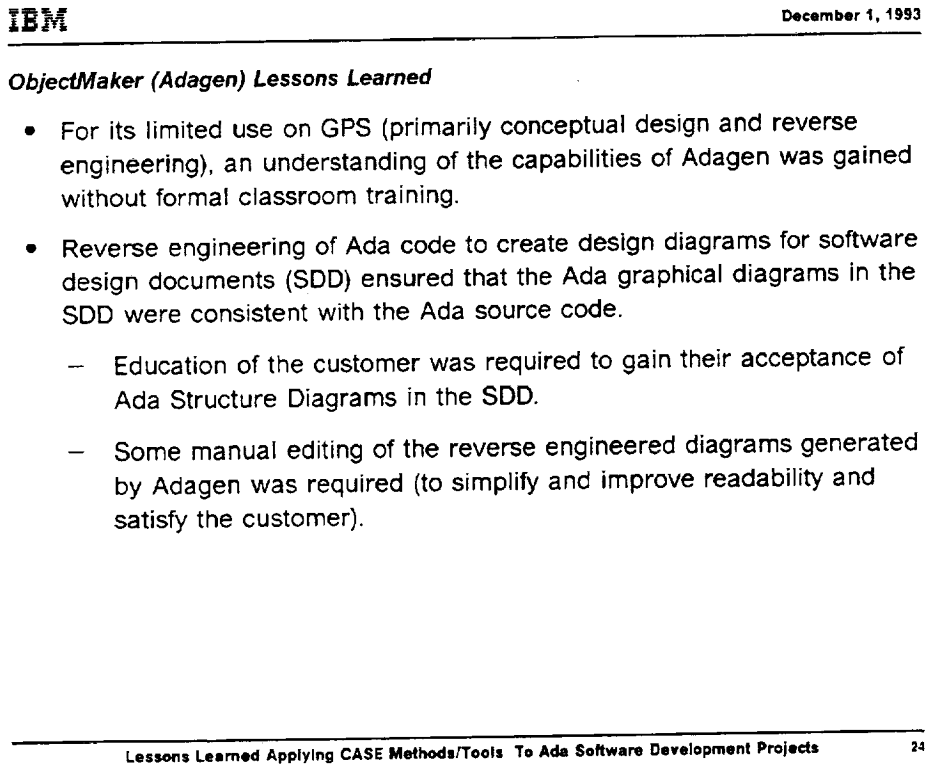
Teamwork Lessons Learned

- An understanding of the basic capabilities of Teamwork was gained without formal classroom training (one person).
- Formal classroom training is required for understanding object-oriented method (Shlaer/Mellor) used by Teamwork.
- Generating an SRS which satisfies the customer's 2167A DIDs (Data Item Descriptions) requires considerable tailoring of the Teamwork templates.

DocEXPRESS Lessons Learned

- DocEXPRESS simplified generation of 2167A compliant documentation
- DocEXPRESS required considerable enhancements (by the DocEXPRESS vendor) to generate an SRS which satisfied the customer's DoD-STD-2167A DIDs (e.g., provide requirements traceability matrices).
- DocEXPRESS documentation and support are of high quality.

Lessons Learned Applying CASE Methods/Tools To Ada Software Development Projects 23

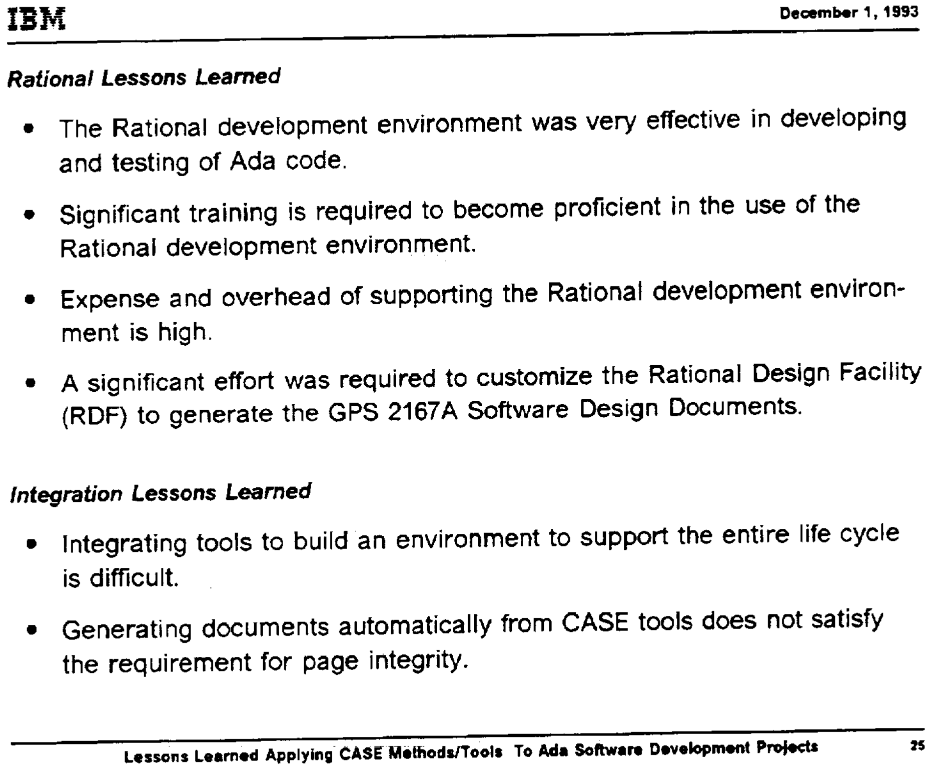


IBM December 1, 1993

ObjectMaker (Adagen) Lessons Learned

- For its limited use on GPS (primarily conceptual design and reverse engineering), an understanding of the capabilities of Adagen was gained without formal classroom training.
- Reverse engineering of Ada code to create design diagrams for software design documents (SDD) ensured that the Ada graphical diagrams in the SDD were consistent with the Ada source code.
  - Education of the customer was required to gain their acceptance of Ada Structure Diagrams in the SDD.
  - Some manual editing of the reverse engineered diagrams generated by Adagen was required (to simplify and improve readability and satisfy the customer).

Lessons Learned Applying CASE Methods/Tools To Ada Software Development Projects 24



IBM December 1, 1993

Rational Lessons Learned

- The Rational development environment was very effective in developing and testing of Ada code.
- Significant training is required to become proficient in the use of the Rational development environment.
- Expense and overhead of supporting the Rational development environment is high.
- A significant effort was required to customize the Rational Design Facility (RDF) to generate the GPS 2167A Software Design Documents.

Integration Lessons Learned

- Integrating tools to build an environment to support the entire life cycle is difficult.
- Generating documents automatically from CASE tools does not satisfy the requirement for page integrity.

Lessons Learned Applying CASE Methods/Tools To Ada Software Development Projects 25

IBM December 1, 1993

Ada CASE Engineering (ACE)

Description

- Internal FSD project
- Setup CASE Tool Environment Laboratory in Manassas
- Performs ongoing evaluations of tools and methods that can improve Ada software development (including maintenance)
- Provides education for tools and methods
- Supports new Ada projects that use CASE tools and Rational Ada development environment
  - Fixed Distributed System (FDS)
  - Advanced Training System (ATS)
  - Global Positioning System (GPS)

IBM December 1, 1993

ACE Lessons Learned from Introducing CASE Tools

- The single most important key to the success of a project is still to understand the problem thoroughly.
- Adequate training in tools/methods must be provided.
- New methods and tools require considerable time to learn and this time must be allocated to a project schedule.
- Tools require considerable lead time before they are operational.
- New methods/tools need to have a strong project advocate.
- A project should have a 'toolsmith' who can customize tools to the project when necessary.
- Consider whether a tool/method might not 'scale up' to a large project.
- Having tools available in the office via networking is a productivity enhancer.
- New tools and methods should not be seen as a panacea.

ACE Lessons Learned from Using CASE Tools

STATEMATE Lessons Learned

- STATEMATE panel generator and Ada Prototyper provide very useful and informative modeling views (e.g., for specification execution (animation), network and processor performance, and user I/F prototyping).
- Language and semantics of STATEMATE require a steep learning curve.
- Definition of STATEMATE naming conventions is very important.

RTrace Lessons Learned

- RTrace supports 2167A requirements traceability.
- RTrace is easy to use and should not require formal education.
- RTRACE is a standalone tool and is not integrated with any CASE tool.
- Projects using RTRACE will need a "guru" to customize reports and perform tool administration functions.
- The current release of RTRACE does not support any automated configuration management or version control.

Miscellaneous Lessons Learned

- Many tools need to be customized before they can be used on a project. This will require a project "toolsmith".
- Most CASE tools have a significant learning curve. Adequate training and learning time are required before users will become proficient in using CASE tools/methods.
- Bringing users onto a technology transition oriented project or a pilot project, before their real use of the tool is required, eases the learning process and makes the user more receptive.
- A course in how to write a good specification improved the quality of specifications produced by the developers.

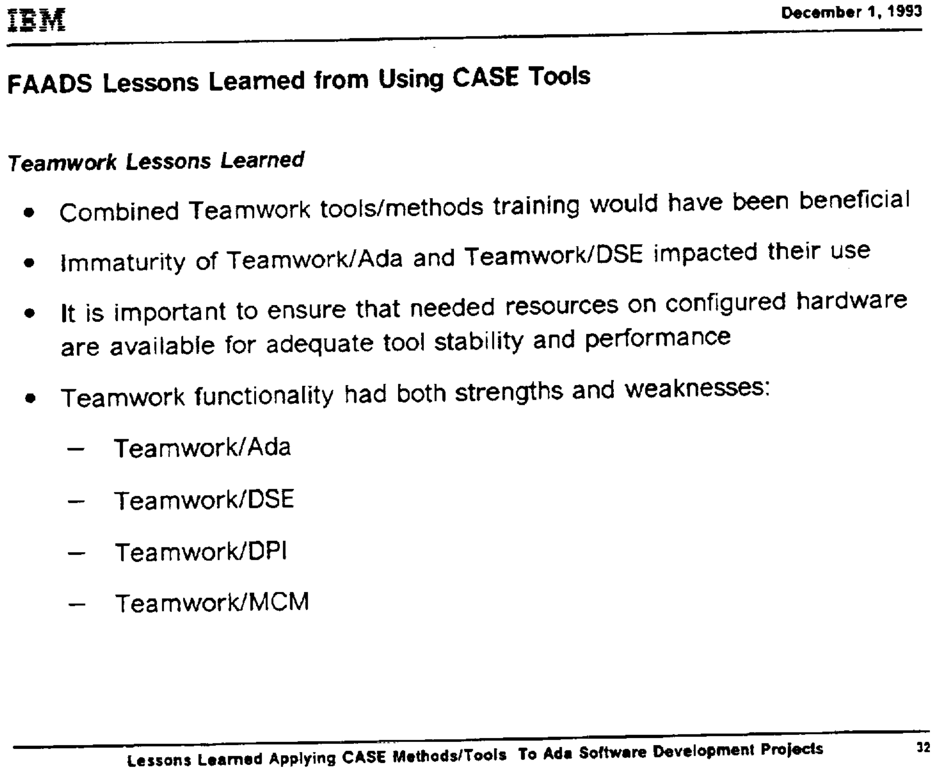
FAADS

Description

- Development of hardware and software for a passive ESM system to support tactical forward area defensive weapons platforms in detecting airborne threats and cueing weapons operators.
- Magnavox is the prime contractor.
- IBM is the software developer.
  - Software to be developed on RISC System/6000 workstations, but will run on 1750A processors.
  - Software to be developed in two phases with planned reuse of Ada code in the second phase.

FAADS Ada Lessons from Introducing CASE Tools

- Significant frontend budget allocation required for training and tools procurement and for installation and maintenance of SEE tools and network.
- Strong management commitment and vision essential.
- New project roles required, e.g., system administrator for LAN and AIX, toolsmith for customizing and supporting use of tools.
- Desktop access to SEE tools important for productivity and use of existing assets.
- Methods and automated support are still immature, e.g.,
  - Graphical design depictions for large Ada systems (Buhr diagrams are not sufficient).
  - Reverse engineering (significant amount of manual work needed to supplement automatically generated diagrams).



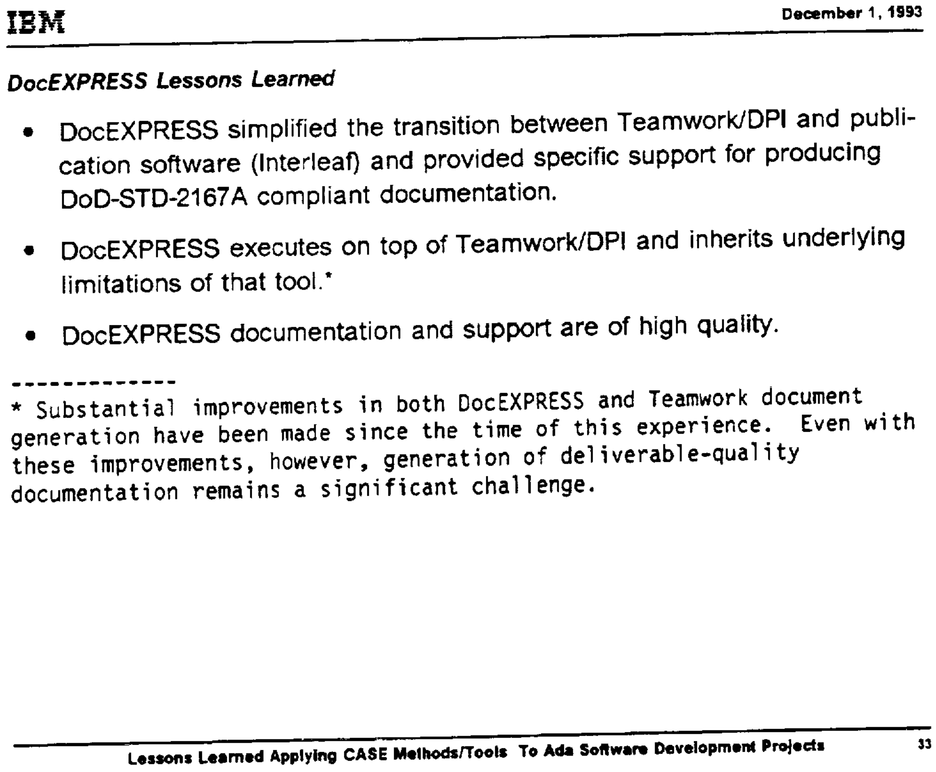
IBM December 1, 1993

FAADS Lessons Learned from Using CASE Tools

Teamwork Lessons Learned

- Combined Teamwork tools/methods training would have been beneficial
- Immaturity of Teamwork/Ada and Teamwork/DSE impacted their use
- It is important to ensure that needed resources on configured hardware are available for adequate tool stability and performance
- Teamwork functionality had both strengths and weaknesses:
  - Teamwork/Ada
  - Teamwork/DSE
  - Teamwork/DPI
  - Teamwork/MCM

Lessons Learned Applying CASE Methods/Tools To Ada Software Development Projects 32



IBM December 1, 1993

DocEXPRESS Lessons Learned

- DocEXPRESS simplified the transition between Teamwork/DPI and publication software (Interleaf) and provided specific support for producing DoD-STD-2167A compliant documentation.
- DocEXPRESS executes on top of Teamwork/DPI and inherits underlying limitations of that tool.*
- DocEXPRESS documentation and support are of high quality.

----

* Substantial improvements in both DocEXPRESS and Teamwork document generation have been made since the time of this experience. Even with these improvements, however, generation of deliverable-quality documentation remains a significant challenge.

Lessons Learned Applying CASE Methods/Tools To Ada Software Development Projects 33

Miscellaneous Lessons Learned

- The impact of introducing multiple changes/technologies was underestimated (the number of changes may have been too ambitious for a relatively small project). These changes included:
  - Migration from a centralized, Vax-based environment to a networked AIX-based environment.
  - Use of several significant new tools, e.g., Ada, Teamwork, Interleaf.
  - Adaptation of the existing software methods and process to the above.
- One of the greatest performance improvements accrued from upgrading the SEE CPU "horsepower":
  - Ada compilations
  - System builds
  - Test runs using the target computer emulator

- Higher quality was realized in the first phase of the contract (FAADS Model I) and higher productivity is expected for subsequent phase (Model II), due to:
  - Most of,the learning curve is over, and the team is acclimated to the adapted software process using the new environment and tools
  - Many of the problems encountered during the first phase have either been solved or workarounds have been devised
  - A Teamwork reuse base has been established for the next phase, providing a head-start in developing the three Model II CSCIs

Session 4: Advanced Concepts

Omit

David J. Campbell, Unisys Corporation

Betty H. C. Cheng, Michigan State University

Janet E. McCandlish, TRW

Software Engineering With Application-Specific Languages

David J. Campbell
Unisys Corporation
Valley Forge Engineering Center
P.O. Box 517, Paoli, PA 19301-0517

Linda Barker
Deborah Mitchell
Unisys Corporation
Space Systems Division
Mail Stop U04D
600 Gemini Ave, Houston, TX 77058

Robert H. Pollack
Unisys Corporation
Valley Forge Engineering Center
P.O. Box 517, Paoli, PA 19301-0517

Abstract

Application-Specific Languages (ASLs) are small, special-purpose languages that are targeted to solve a specific class of problems. Using ASLs on software development projects can provide considerable cost savings, reduce risk, and enhance quality and reliability. ASLs provide a platform for reuse within a project or across many projects and enable less-experienced programmers to tap into the expertise of application-area experts.

ASLs have been used on several software development projects for the Space Shuttle Program. On these projects, the use of ASLs resulted in considerable cost savings over conventional development techniques. Two of these projects are described.

1 Introduction

An application-specific language is a special purpose language that is oriented towards writing programs for a specific class of problems. An ASL presents the programmer with a higher level of abstraction than a general-purpose programming language, and, as a result, the programmer needs to write much less code to implement a software system.

The ASL code written by a programmer is called a *specification*: it describes the requirements for a software system. A *translator* reads a specification, as shown in Figure 1, and automatically generates software and perhaps other related products, such as accompanying

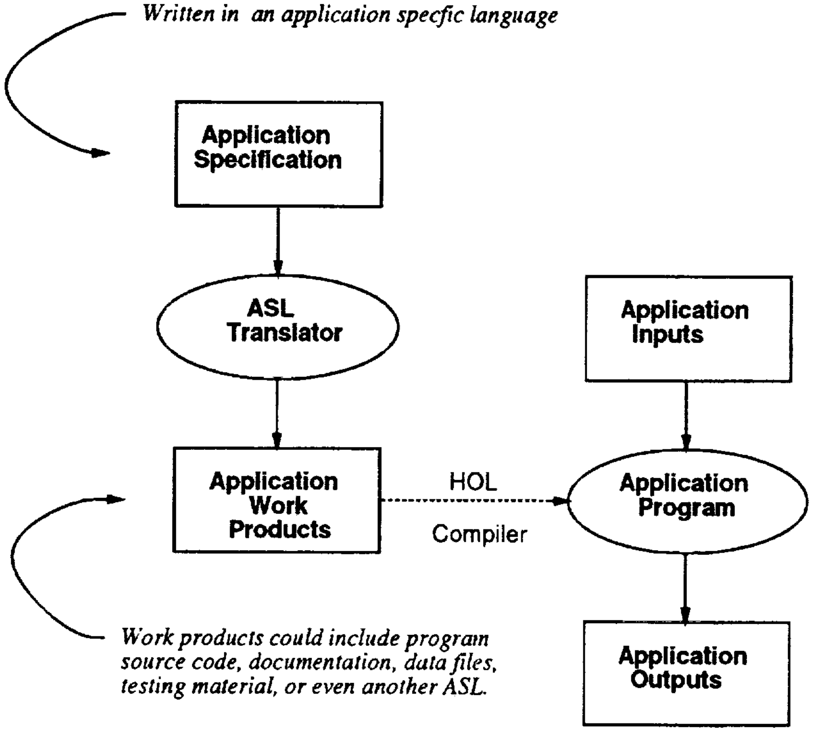

Written in an application specfic language
Application Specification
ASL Translator
Application Inputs
Application Work Products
HOL
Compiler
Application Program
Work products could include program source code, documentation, data files, testing material, or even another ASL.
Application Outputs

Figure 1: The ASL translator generates software and other related products based on a specification written in a high level language.

design documentation, that satisfy the specification. Usually, the generated software is in a high-order language such as C or Ada.

Today, there are many ASL based commercial off-the-shelf products (sometimes called 4GLs), that address such diverse application areas as data base applications, spread sheets, and graphical user interfaces. If a COTS ASL can be found which meets the needs of a software development project, it will often produce seemingly miraculous results. If such a tool cannot be found, however, an ASL approach is usually abandoned.

This is unfortunate because custom ASLs can be created rather inexpensively, and they can provide considerable advantages to projects that are developing software with certain characteristics. ASLs can increase productivity and reliability by shifting more of the tedious work and mechanical details to the computer, freeing programmers to spend more time addressing the decisions that require creative thinking. ASLs also provide a single point of control for a large amount of software. This enables requirements and design decisions to change with minimal impact on cost and schedule.

2 An Overview of ASL-Based Software Engineering

ASL-based software engineering is a software engineering technique for creating software through automatic code generation. It is not suited to all projects, but there is a large class of applications where its use can dramatically reduce cost and schedule. For any given project, many different techniques may be applicable, and the best approach may be a combination of techniques. Since software engineers are relied upon to identify the most cost-effective approach, they should be knowledgeable of this technique.

An ASL approach is indicated for a software system when it has recurring similar requirements, especially if there is a large number of them. For example, the requirements might define a series of screens that a system uses to interact with its user. While each screen is different, they are also similar, e.g., each screen contains editable fields for data entry and data validation must be done on each field. If these similar requirements can be implemented with similar code, and an algorithm to transform the requirements into the code can be found, then an ASL can be used.

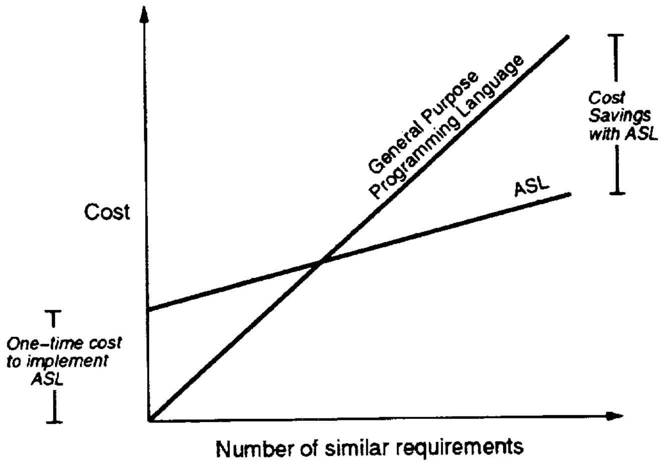


Figure 2: This graph compares the cost of using an ASL versus the cost of using a general-purpose programming language, based on the number of similar requirements. Initially, the ASL is more expensive, because of the one-time only cost to develop the translator. With sufficient repetition in the requirements, however, the cost to develop translator pays for itself.

With ASLs, there is a one-time cost to implement the translator. After the translator is implemented, a specification still must be written to obtain any application code. However, compared with a general-purpose programming language, fewer lines of ASL code are required to implement a corresponding amount of the system's functionality. Moreover, a programmer can typically write more lines of ASL code per day, because, with ASLs, the programmer is transcribing already written requirements into the syntax of the ASL, whereas with a general-purpose programming language, the programmer must write code which describes how to implement the requirements. Consequently, as the amount of repetition in the requirements increases, the cost of implementation with an ASL falls below the cost of implementing software with general-purpose programming language. This relationship is shown in Figure 2

Even if there is not enough repetition to produce a dramatic cost difference, other factors may warrant the use of an ASL. For example, can the ASL be reused on other projects?

Is the algorithm to transform requirements into implementation so complex, that it is best handled by a computer? Are the requirements volatile? Are there risk factors that might cause a possible re-design of the software, e.g., performance issues? If there is significant risk that the requirements or the design may change, then using an ASL will make the software more manageable, because the code is controlled from a single point.

Implementation of an ASL requires a team of engineers with collective expertise both in the application area being addressed and in language implementation. This team must design a generic solution to the problem, which is expressed as a set of reusable code templates and an algorithm to instantiate these templates based on requirements. This design, i.e., the templates and the translation algorithm can be reviewed just like any other form of design.

The language expert designs a language for expressing the information required to instantiate the templates. This language will typically incorporate terminology and notations used by the application experts so that they can easily write or review the specifications. The language enables the *variances* in the similar requirements to be expressed. For example, while each screen consists of a set of unique fields and control buttons, they may also contain a set of standard controls, e.g., controls that return to the previous screen or quit the program. Since the standard controls appear on all screens, they do not need to be specified in the ASL; instead, the translator can automatically supply them.

The language expert also builds the translator. The translator reads an input specification, extracts the information needed by the translation algorithm, and generates the output products by instantiating the templates. The translator may perform semantic checks on the input specification to check that it describes a valid application.

In order to produce other related products from the same specification, such as design documents, test plans, or test data, templates for these products must be designed and logic must be added to the translator to instantiate these templates. The ASL may be enhanced to include additional information that is necessary to instantiate these templates.

Based on our experiences at Valley Forge Engineering Center implementing many different ASLs over the past decades, implementing an ASL, i.e., designing the language and implementing the translator, typically takes from a few weeks to several months, depending on the complexity of the specification language. This cost includes designing the language and implementing the translator only; it does not include the cost of writing any required support software which the generated code may call upon. Since the support software (or software with similar functionality) is usually required whether or not an ASL is used, it is not be figured into the cost of implementing the translator.

There are two reasons why ASLs are relatively inexpensive to implement. First, the underlying technology and theory used to build ASL translators comes from the well-understood software domain of compilers. Many automated tools exist for this domain, e.g., code generators for building lexical analyzers and parsers. Besides automated tools, there are standard architectural designs for translator programs and libraries of commonly used components.

Second, ASLs are much easier to implement than compilers. The generated code is a high-level language, instead of a machine language. The generated code can interface with other software components to implement its functionality. Also, ASLs are much simpler languages than general-purpose programming languages. Since the design of the language is under the control of the implementer, language constructs which are hard to implement can be avoided. It, therefore, is possible to design and implement a translator for a small, special-purpose language, at lower cost and risk than most other types of software.

The Benefits of ASL-Based Software Engineering

ASL-based software engineering provides a number of benefits, including:

- Increased Productivity
- Increased Reliability
- Better Control
- Lower Maintenance Cost
- Increased Reusability

Increased Productivity

First, there is less code to write, because a software description written in an ASL is much shorter than that same software written in a general-purpose programming language. Second, more lines of ASL code can be written per day than lines of a general-purpose programming language, because, when an ASL is used, the programmer writes a description of *what* the application does, instead of writing a description of *how* it does it.

Moreover, ASLs can be use to capture the expertise of an experienced programmer and transfer it to less experienced programmers. For example, an ASL that allows programmer to build screens for X-windows by just describing their appearance, enables the coding of the screens to be done by a programmer that does not know X-windows. The translator contains the knowledge of an X-windows expert on how to transform the descriptions into the appropriate X-windows code.

Increased Reliability

Generated code is more reliable than hand-written code. Since all of the code is based on the same set of templates, once the templates are correct, all of the code will be correct. The computer can be counted on to perform the repetitive task of instantiating the templates accurately.

Better Control

The form and content of the generated code is controlled from a single-point, the translator; consequently, all of the generated code can easily be changed. A single point of control reduces risk by allowing many design decisions to be deferred. For example, if a generated system interfaces with another complex system, e.g., X-Windows, the design of the generated system can be fine-tuned later, after more experience is gained, by simply changing the generator. On the other hand, when there is large amount of hand-written code, it is desirable to completely decide on the design before the code is written, because of the cost of retrofitting a change in all of the code.

Also, if the translator generates multiple products, then the products are kept in synchronization automatically. For example, if a translator generates a program and a structure chart which describes the design of the program, then the design documentation and program always parallel each other.

Lower Maintenance Cost

Perhaps the biggest benefit of using an ASL approach is realized in the maintenance phase of the life cycle. There is less code to maintain. Moreover, the capability evolve the system to accommodate new requirements is built into the system; features can be added or modified by making changes to the specification.

Sometimes, over the lifetime of a program, fundamental changes must be made to its overall design, e.g., porting the program to a different hardware platform, operating system, windowing system, database, or even programming language. ASLs facilitate this, because the specification and translator maintain a clean separation between what a program does and how a program does it. In order to retarget a program, only the translator must change. All of the code invested in the specification is still valid because it is independent of the implementation.

Increased Reusability

ASLs extend the scope of reuse beyond what is possible with conventional development techniques and general-purpose programming languages. When an software component is implemented in a general-purpose programming language, the amount of customization that can be done is limited by the parameterization methods available in the language. When a component is generated, however, more possibilities for customization exist, because the generator can add, modify or omit code.

3 Examples of ASLs

ASL technology has been applied on several software development project at NASA/Johnson Space Center. The work was performed under the Space Transportation System Operations Contract (STSOC) on which Unisys Space Systems Division is a subcontractor to Rockwell Space Operations Company.

In this section, we present the work done on two projects to give examples of two ASLs that address completely different kinds of problems. On one project, done for the Payload Operations branch, a command editor for the Tethered Satellite was implemented using ASLs. On the other project, done for the Shuttle Flight Design and Dynamics branch, an ASL was implemented to serve as a general-purpose tool for analyzing data files used during flight design.

3.1 Tethered Satellite Command Editor

Approximately 500 Tethered Satellite System (TSS) payload commands required editing. The ground control specialist uses menus to select commands for editing. Menu buttons either display a submenu or a screen for editing commands. A sample of a menu and a screen is shown in Figure 3.

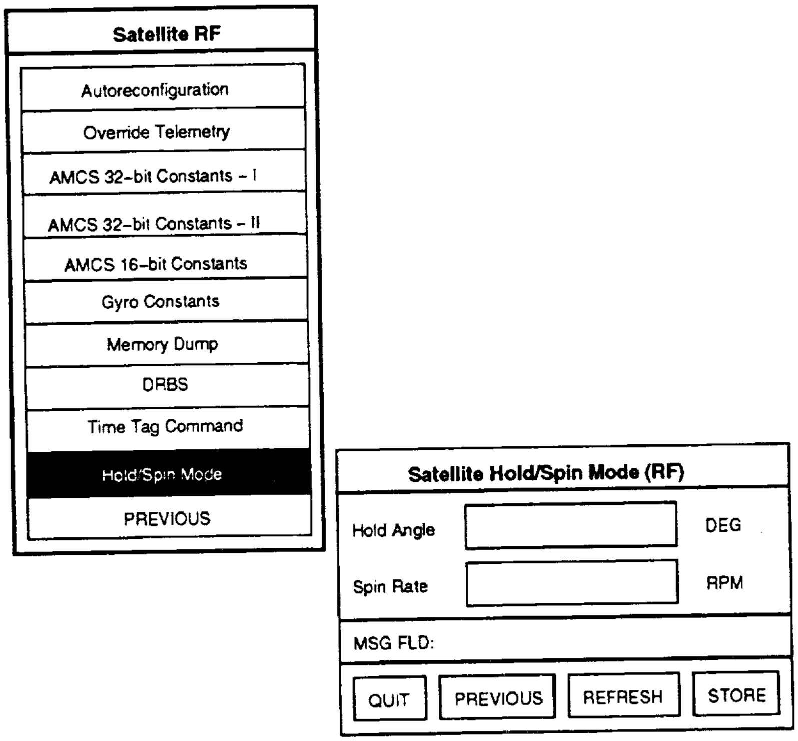


Figure 3: The command editor provides a GUI for selecting and editing commands. A sample menu and and a screen for editing two commands is shown.

Screens have varying requirements for grouping of commands; some screens process one command, while others process 35 or more commands. Each command must be retrieved from a database and stored again after it has been modified. Five different command formats must be processed, each with a unique checksum calculation. Some commands required values to be converted to engineering units, and most commands require values to be displayed both symbolic and in hexadecimal. A single group of commands can be optionally loaded from an external file, rather than the database.

Rather than assigning many programmers to build 140 or so screens—having each programmer code similar sorts of things, but each doing it differently—we invested in the design of special specification language, in which each command and screen layout can be described. A sample of this language is shown in Figure 4. Common capabilities such as the need for certain buttons on each screen, the retrieval of data, and conversion and checksum calculation were built directly into the associated generation process. The specification had

```
Command Format RF_32_bit_degrees is
   Format : RF;
   (7,0)[32] Degrees : Sat_Degrees;
end RF_32_bit_degrees;

Command format RF_16_bit_RPM is
   Format : RF;
   (7,0)[16] RPM : Sat_RPM;
end RF_16_bit_RPM;

Satellite_Hold_Mode_On_RF: P13K1020L RF_32_bit_degrees;
Satellite_Spin_Mode_On:    P13K1022L RF_16_bit_RPM;

Form Hold_Spin_Mode_RF is
   title : "Satellite Hold/Spin Mode (RF)";
   "Hold Angle", Satellite_Hold_Mode_On_RF.Degrees, "DEG";
   "Spin Rate",  Satellite_Spin_Mode_On.RPM,        "RPM";
end Hold_Spin_Mode_RF;

Menu RF_Menu is
   title : "Satellite RF";
   "Autoreconfiguration"        => Auto_Reconfiguration_RF;
   "Override Telemetry"         => RF_Override_Telemetry_Form;
   "AMCS 32-bit Constants - I"  => amcs_constants_32_RF_page1;
   "AMCS 32-bit Constants -II"  => amcs_constants_32_RF_page2;
   "AMCS 16-bit Constants"      => AMCS_Constants_16_RF;
   "Gyro Constants"             => RF_Gyro_Constants;
   "Memory Dump"                => RF_Memory_Dump_Form;
   "DRBS"                       => DRB_Menu;
   "Time Tag Command"           => RF_Time_Tag_Command;
   "Hold/Spin Mode"             => Hold_Spin_Mode_RF;
end RF_Menu;
```

Figure 4: This is the specification for the screens shown in Figure 3. Besides displaying the menu, the code generated for this specification fetches two commands from the database (P13K1020L and P13K1022L), extracts the Degrees and RPM field from each command respectively, and displays their values on the screen for editing by the user. If the user presses the STORE button, the commands in the database will be updated with the last value the user entered.

all the implementation details for each command; the generator integrated all special process requirements with common capabilities.

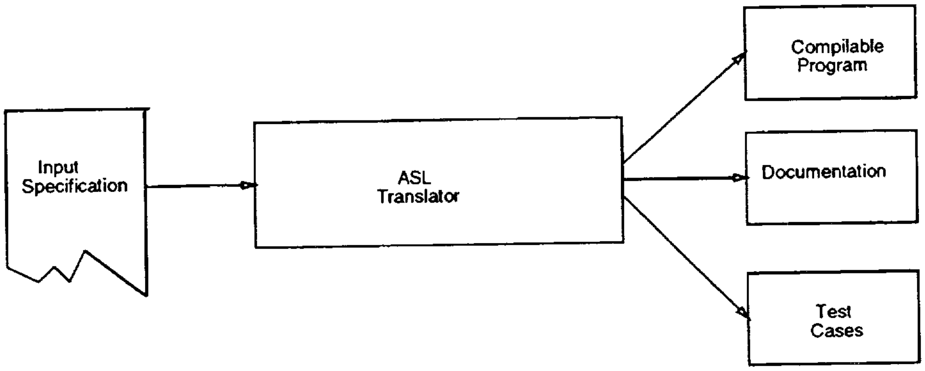


Figure 5: The TSS ASL translator generates a command editor, a user's manual for the command editor, and test program from a single specification.

The translator generates three significant products for this project as shown in Figure 5. The main product consists of several thousand lines of high quality, maintainable C code. In addition, a 200 page user's manual and test program are produced. The user's manual describes how to use the editor and the screens that editor is capable of displaying. The test program validates that each TSS command exists in the database and is defined as specified. Additionally, high and low value entry is simulated for each editable data value.

The ASL approach accommodated introducing new requirements in the unit testing phase with *no* impact to schedule. During this phase, about 40 new screens were requested by the customer to handle science commands. To accommodate this request, no actual C coding was required, only descriptions of the new screens had to be added to the specification. Then a new editor, user's manual, and test program were generated automatically.

The productivity for the command editor application was not tracked in detail. The translator consists of 7K lines of code, 4K lines were hand written for this project and 3K lines were reused or generated; the level of effort to produce the translator was 3 person months, including the design of the templates for the generated code. The TSS Command Editor is 12K lines of code, 7K lines are generated by the translator, and 5k lines are hand written. The hand-written code is used by the generated code and is not changed to accommodate new specifications. The generated test program for the TSS editor is 6K lines of code, and the generated user's manual is 12K lines of `troff` and `pic` commands. Additional productivity gains have been achieved, because the command editor generator has been used for other payloads, e.g., SSBUV and Wake Shield.

3.2 Strip Manipulate and Merge Tool

The Strip Manipulate and Merge (STMM) tool was created by the Common Software task as part of its overall goal to reduce maintenance cost by creating a common set of tools for use by flight designers, since many of the existing tools duplicate functionality.

STMM accepts a specification that describes operations to be performed on standard flight design data files. There are several different types of data files used for flight design. Each data file type has its own physical format; however, all of the data files are logically similar—Each file consists of a collection of records; each record is the same type, consisting of a set of named fields; and each file has a data dictionary which describes the structure of the records, i.e., the names of the fields in the record, the number of bytes allocated to the field, the type of data in the field (e.g. ASCII or binary), and the engineering units represented by the data.

STMM replaces an existing set of forty or so tools that perform similar, but specific, operations on flight design data files, such as converting from one file format to another; creating a file from selected records of another file; or omitting, reordering, renaming, or adding fields to the records of a file. In addition, some tools perform operations on multiple data files, such as concatenating, merging or joining them. Each tool did some specific combination of the above operations on a specific set of data files. With STMM, these forty custom tools are replaced by forty small specifications and the STMM tool itself.

Originally, STMM was to be implemented using a COTS product that manipulates flat files. After analysis, it was found that the COTS product could not adequately replace the existing set of tools. The COTS product did not support the number fields that records in some of the data files had. It did not support operations such as joining or merging files based on a tolerance for the key fields. And finally, it could not convert from one file type to another. The additional support code required to use the COTS solution made the COTS implementation unfeasible, so a custom ASL was implemented.

```
merge "run1.cff"(cff) and "run2.cff"(cff) giving "out.merge"(fcff);

   record selection for "run1.cff"(cff) is
      range : Number in 1.0e6 .. 2.0e6;
   end;

   key is Pressure;
end;
run
```

Figure 6: This sample language specification merges two data files, *run1.cff* and *run2.cff*, producing a the result file *out.merge*. The files are merged on the key field *Pressure*. The only records selected from *run1.cff* are records where the value of the field *Number* is in the range one million to two million.

One of goals of STMM, was to make the language easy to use by flight design engineers, who are not necessarily computer programmers, so that new file manipulation programs could easily be created by them. The language designed for STMM allows the user to express operations on data files using an is English-like syntax, which is easy to read and write. A sample of the STMM language is shown in Figure 6. Also, extensive error checking was built into the translator to make it easier for the user to debug specifications.

The architecture of STMM is slightly different from the other ASLs that we have been discussing. Instead of translating the user specification into an HOL program, which must then be compiled, the translator generates an internal, intermediate language that represents the user's program. A component called an *interpreter* executes this intermediate language.

The interpreter for STMM makes use of a library that defines a class of objects called *filters*. There are several types of filters; each type of filter can be connected to one or more input streams of data and produces an output stream of data. In addition, each type of filter is capable of doing some kind of transformation on its input streams to produce its output stream. For example, there are filters which select records based on parameterizable criteria, strip fields from records, or concatenate, merge, or join multiple streams of data. The STMM translator translates the specification into the appropriate chain of filters. Once the filter chain has been constructed, the translator turns to control over to the filters to executed the operations.

Summary

The way in which software is produced has changed several times since the invention of electronic computers. All of these changes consist of transferring an increasing amount of work from human beings to the machine itself. Application-specific languages are a step in this trend. They enable software engineers to leverage the tools and techniques from a well-understood domain—compilers—against problems of developing new software.

Application-specific languages provide many important benefits to a project during implementation and maintenance phases. They increase productivity, increase reliability, provide control of a large amount of software and related products from a single point, and enhance the ability of a system to adapt changing requirements.

Because of the success of ASLs on these and other STSOC software development projects, ASL training was given to a team of about twenty STSOC software engineers. These engineers will assess new projects and existing maintenance efforts to find areas where ASLs can reduce cost.

Biographical Sketch

David J. Campbell has over seventeen years experience in compiler, operating system, and support tools development. In addition to his work at Unisys, he is a part-time instructor for the Mathematical Sciences Department/Computer Science Division, Villanova University. For the past seven years, the main focus of his work has been on automatic generation of software, chiefly through the use of compiler development technology. His work includes the implementation of many software generators and the creation of tools to build software generators. He has also been involved with many tasks on the STARS program, including porting a Sun Unix version of the Common APSE Interface Set, revision A, to the MACH operating system, and serving as chief programmer on the rapid software modeling task.

Mr. Campbell is currently a Staff Engineer in the Research and Development Division of Valley Forge Laboratories. He holds an B.S. degree in computer science from Wichita State University.

Linda Barker has over seventeen years experience in the computer industry. She is currently Supervisor of Software Engineering for the Mission Control Center, Data Systems Software Section, which is responsible for maintaining several applications used in the ground support operations for space shuttle flights. She is also a charter member of the Houston-SSO Software Engineering Process Group (SEPG).

Deborah A. Mitchell has over fourteen years experience in programming and software support on a variety of hardware systems. For the past six years, she has worked on the Space Transportation System Operation Contract in the Flight Design and Dynamics Department. Her work includes project management of Common Software applications, the development of two ASL applications, the General Purpose Input Processor and the Strip Manipulate and Merge, Generic Report Writer.

Deborah Mitchell is currently a project manager in the Reconfiguration Department of the Unisys, Houston, division. She holds a Bachelor of Science in Electrical Engineering (BSEE) from Prairie View A&M University.

Robert H. Pollack has over twenty years experience in programming and software support, on a variety of hardware and operating systems. For the past nine years, the main focus of his work has been on the automated creation of application software, chiefly through the use of compiler development technology. His work includes the creation of a system to generate Ada message validation code from abstract specifications of the message formats, a system which is used for software development in several Unisys projects. He is also the creator of a major subsystem of an interpreter for the Ada language developed under the STARS program.

Mr. Pollack is currently a Staff Engineer in the Research and Development Division of Valley Forge Laboratories, where he is assigned to the Re-Engineering IR&D project. He holds an M.S.E. (Computer and Information Science) from the University of Pennsylvania.

Software Engineering With Application-Specific Languages

David J. Campbell
Unisys Corporation
PO Box 517
Paoli, PA 19301
Campbell@VFL.Paramax.COM

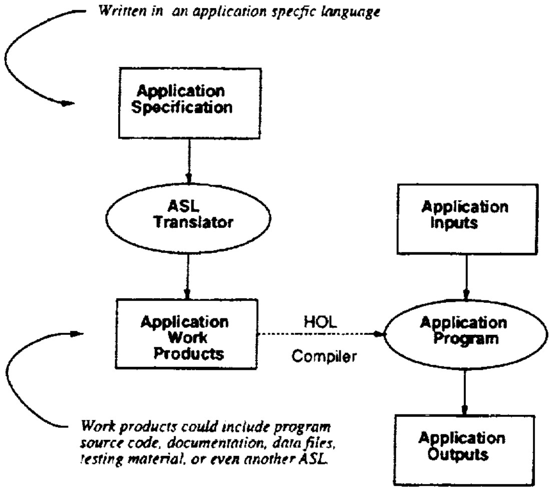
Application-Specific Languages(29 November 1993) Foil 1

Application-Specific Languages (ASLs)

- Special-purpose languages targeted to solve a specific class of problems
- Present programmers with a higher level of abstraction than general-purpose languages, allowing a programmer to write less code
- Used to automatically generate required software or other related work products
- Inexpensive to produce (typically, from a few weeks to a few months)

Application-Specific Languages(29 November 1993) Foil 2

Automatic Software Generation With ASLs


Written in an application specfic language
Application Specification
ASL Translator
Application Inputs
Application Work Products
HOL
Compiler
Application Program
Application Outputs
Work products could include program source code, documentation, data files, testing material, or even another ASL

Application-Specific Languages(29 November 1993) Foil 3

Automatic Software Generation With ASLs (Cont.)

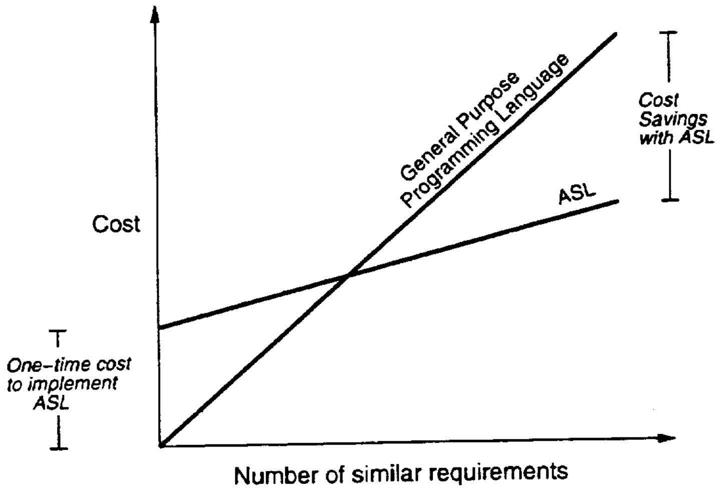
- Specification and translator maintain a clean separation between *what* software does, and *how* it does it
- Generic solution to problem is formulated as a set of reusable code templates
- Translator executes an algorithm that instantiates templates from a specification which describes the requirements for the software

Application-Specific Languages(29 November 1993) Foil 4

Evaluation process

- Determine if a software component is a candidate for ASL implementation
  - Repetitive coding tasks
  - Complex or error-prone coding tasks
  - Requirements subject to change
  - Recurring problem (i.e., ASL is reusable on other projects)
- Perform tradeoff analysis, ASL vs other approaches

Cost Tradeoff



ASL Development Activities

- Language Design
  - Design a language for specifying requirements in terms familiar to the application expert
- Translator Development
  - Develop a translator that checks the input specification for errors and generates code that satisfies the requirements
- Product Generation
  - Write specification for the required work products and generate the actual components

Benefits

- Increased Productivity
  - Less code to develop and maintain
- Increased Reliability
  - All code based on same templates
  - Computer accurately instantiates templates
- Increase Manageability
  - Translator provides a single-place for controlling a large amount of code and related work products
  - Design decisions are encapsulate in the translator
  - Less impact to evolve design or tune implementation

Benefits (Cont.)

  - Related work products are always consistent
  - Less impact to handle anticipated requirements changes
- Increased Reusability
  - Generated components are more tailorable than components implemented in programming languages

Application-Specific Languages(29 November 1993) Foil 9

Examples of ASLs

- Editor Generator (Egen)
- Strip Manipulate and Merge (STMM)

Application-Specific Languages(29 November 1993) Foil 10

Egen (Editor Generator)

- Egen is an ASL that generates a payload command editor from a high-level specification
- Initially developed for the TSS payload, subsequently used on the SSBUV and Wake Shield payloads

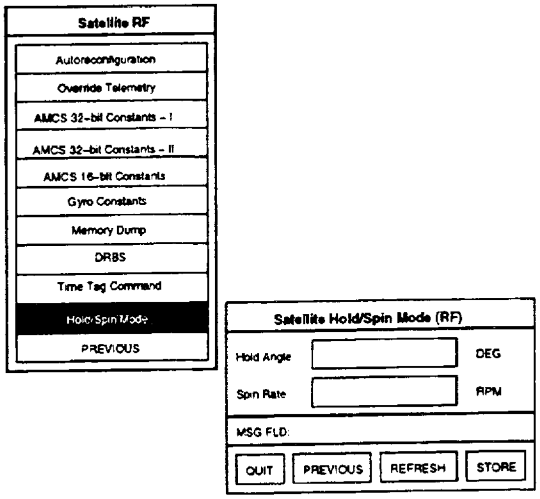
Application-Specific Languages(29 November 1993) Foil 11

Command Editor

- Fetches and stores commands from a data base
- Enables the user to display and change the variable fields of commands
- Converts values to engineering units
- Handles different command formats and computes checksum required by formats
- Provides a GUI for editing commands

Application-Specific Languages(29 November 1993) Foil 12

Example of User Interface



Application-Specific Languages(29 November 1993) Foil 13

The Egen Specification

```
Command Format RF_32_bit_degrees is
   Format : RF;
   (7,0)[32] Degrees : Sat_Degrees;
end RF_32_bit_degrees;

Command format RF_16_bit_RPM is
   Format : RF;
   (7,0)[16] RPM : Sat_RPM;
end RF_16_bit_RPM;

Sat_Hold_Mode_On_RF: P13K1020L RF_32_bit_degrees;
Sat_Spin_Mode_On:    P13K1022L RF_16_bit_RPM;

Form Hold_Spin_Mode_RF is
   title : "Satellite Hold/Spin Mode (RF)";
   "Hold Angle", Sat_Hold_Mode_On_RF.Degrees, "DEG";
   "Spin Rate",  Sat_Spin_Mode_On.RPM,        "RPM";
end Hold_Spin_Mode_RF;
```

Application-Specific Languages(29 November 1993) Foil 14

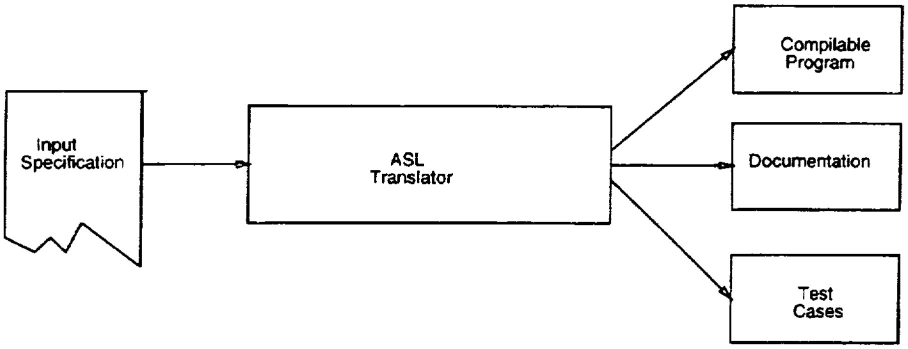
The Egen Specification (Cont.)

```
Menu RF_Menu is
   title : "Satellite RF";
   "Autoreconfiguration"         => Auto_Recon_RF;
   "Override Telemetry"          => RF_Override_Telm;
   "AMCS 32-bit Constants - I"  => amcs_32_RF_page1;
   "AMCS 32-bit Constants -II"  => amcs_32_RF_page2;
   "AMCS 16-bit Constants"      => AMCS_16_RF;
   "Gyro Constants"             => RF_Gyro_Constants;
   "Memory Dump"                => RF_Memory_Dump_Form;
   "DRBS"                       => DRB_Menu;
   "Time Tag Command"           => RF_Time_Tag_Command;
   "Hold/Spin Mode"             => Hold_Spin_Mode_RF;
end RF_Menu;
```

Application-Specific Languages(29 November 1993) Foil 15

Egen Translator

- Egen produces multiple work products



Application-Specific Languages(29 November 1993) Foil 16

STMM (Strip Merge and Manipulate)

- STMM programs describe operations to be performed on flight design data files
  - Create files from selected records of other files
  - Omit, rename, reorder, or add additional fields to records
  - join, concatenate, or merge files
  - convert files from one format to another
- It replaces forty programs that perform specific operations on given files

Application-Specific Languages(29 November 1993) Foil 17

Example of a STMM Specification

```
merge "run1.cff"(cff) and "run2.cff"(cff)
   giving "out.merge"(fcff);

   record selection for "run1.cff"(cff) is
      range : Temperature in 1.0e4 .. 2.0e4;
   end;

   key is Pressure;
end;
run
```

Application-Specific Languages(29 November 1993) Foil 18

Applying Formal Methods and Object-Oriented Analysis to Existing Flight Software

Betty H. C. Cheng[*]
Michigan State University
Department of Computer Science
A714 Wells Hall
East Lansing, MI 48824-1027
chengb@cps.msu.edu

Brent Auernheimer
California State University, Fresno
Department of Computer Science
Fresno, CA 93740-0109
brent_auernheimer@CSUFresno.edu

Abstract

Correctness is paramount for safety-critical software control systems. Critical software failures in medical radiation treatment, communications, and defense are familiar to the public. The significant quantity of software malfunctions regularly reported to the software engineering community, the laws concerning liability, and a recent NRC Aeronautics and Space Engineering Board report additionally motivate the use of error-reducing and defect detection software development techniques.

The benefits of formal methods in requirements-driven software development ("forward engineering") is well documented. One advantage of rigorously engineering software is that formal notations are precise, verifiable, and facilitate automated processing. This paper describes the application of formal methods to reverse engineering, where formal specifications are developed for a portion of the shuttle on-orbit digital autopilot (DAP). Three objectives of the project were to: demonstrate the use of formal methods on a shuttle application, facilitate the incorporation and validation of new requirements for the system, and verify the safety-critical properties to be exhibited by the software.

1 Introduction

Correctness is paramount for safety-critical software control systems. Critical software failures in medical radiation treatment [1], communications [2], and defense [3] are familiar to the public. The significant quantity of software malfunctions regularly reported to the software engineering community [4], the laws concerning liability [5], and a recent NRC Aeronautics and Space Engineering Board report [6] additionally motivate the use of error-reducing and defect detection software development techniques.

The benefits of formal methods in requirements-driven software development ("forward engineering") is well documented [7, 8, 9, 10]. One advantage to using rigorous approaches to software engineering is that formal notations are precise, verifiable, and facilitate automated processing [11, 12, 13].

We claim that maintenance of critical existing ("legacy") code also benefits from formal methods. For example, formal specifications can be reverse engineered from existing code. The resulting formal specifications can then be used as the basis for change requests and the foundation for subsequent verification and validation. Considering re-implementation's high cost and, even worse, the failure of critical software, reverse engineering of code into formal specifications provides an alternative or a supplement to traditional approaches for maintaining safety-critical systems.

This paper describes a project that applies formal methods to a portion of the shuttle on-orbit digital autopilot (DAP). Three objectives of the project were to: demonstrate the use of formal methods on a shuttle application, facilitate the incorporation and validation of new requirements for the system, and verify the safety-critical properties to be exhibited by the software.

In addition to developing formal specifications of a critical module, a graphical depiction of the subsystem was constructed using the *Object Modeling Technique* (OMT) [14] to provide an object-oriented view of the system as it relates to the functional and dynamic views. Lessons learned from this project are described, including discussions of the benefits of constructing specifications and the ability to generate proofs from the formal specifications.

The remainder of the paper is organized as follows. Section 2 gives a brief introduction to formal methods and object-oriented development techniques. Section 3 gives an overview of the entire project, including a discussion of the object-oriented analysis and the development of the OMT diagrams. A summary of lessons learned from this project are discussed in Section 4. Finally, concluding remarks and future investigations are given in Section 5.

2 Background Material

This section briefly defines and motivates the use of formal methods. Also, the benefits of object-oriented analysis and design are presented.

[*]This author is also supported in part by NSF grant CCR-9209873.

2.1 Formal Methods

Formal methods in software development provide many benefits in the forward engineering aspect of software development [7, 8, 9, 15]. For any specification, there can be any number of implementations that satisfy the specification [16].

Due to the criticality and the volume of much of the software being developed by many agencies involved in flight systems, there are several projects incorporating formal methods into the software development process [17]. In addition, there have been recent investigations into reverse engineering that focus on the use of rigorous mathematical methods for extracting formal specifications from existing code [18, 19, 20].

A *formal method* consists of a *formal specification language* and *formally defined inference rules* [15]. The specification language is used to describe the intended system behavior and the inference rules provide a sound method for reasoning about the specifications. Using formal specifications for software design serves several general purposes. First, it forces the designer to be thorough in the development and the documentation of a system design. Second, the developer is able to obtain precise answers to questions posed about the properties of the system, and therefore be able to rigorously test (by developing theorems) the design for the satisfaction of its requirements. Unfortunately, since the requirements are traditionally expressed informally, there remains a (albeit decreased) potential for errors to remain undetected. Third, the developer is able to reason about the correctness of a system or a safety-critical component of the system with respect to its specification. The latter category of reasoning can be divided into two approaches: *program verification* and *program synthesis*. Program verification is the process of checking the semantics of a program text against its specification. A program whose semantics satisfies its specification is said to be correct. Program synthesis refers to formal techniques for systematically developing a program from a specification such that the correctness of the resulting program (with respect to its specification) is inherent in the development process itself [21, 22, 23, 13].

Formal methods are typically more difficult to apply than informal approaches and require a great deal more discipline. Furthermore, the state of the current technology is such that verification and the use of formal methods is largely done manually, thus requiring a tremendous effort to perform tedious, but necessary tasks. In general, the introduction of formality in software development is a difficult but valuable step in the construction of reliable and maintainable computer systems. The difficulty is largely due to the quantity of detail required by formalization as well as the tedious process by which the formalisms must be manipulated. However, the detection and correction of design flaws, ability to use automated tools for manipulation, elimination of ambiguity, precise documentation for maintenance, and improved reusability are a few examples of the overwhelming value, and often necessary benefits, that formal methods brings to the software development process.

2.2 Object-Oriented Techniques

There are a wide variety of approaches to requirements analysis, many of them in the broad category known as *object-oriented requirements analysis* (OOA) [14]. An *object* is a data abstraction, and it is the goal of OOA to construct an abstract, object-based model of the problem domain. The OOA focus on objects is in contrast to the more traditional approach to analysis that focuses on procedures [24]. That is, instead of modeling the problem domain as a system of operations that process data objects, OOA modeling centers on a description of data objects and their interactions.

Most OOA techniques begin by a careful assessment of the natural language problem description. A simple first step in developing an OOA model is to extract the *nouns* from the problem description. Many of these nouns will share common properties and may be more easily described as instances of *types*. For example, *Galileo*, *Voyager*, and *Magellan* are all spacecrafts, and *Venus*, *Mars*, and *Mercury* are all planets. In this context, spacecraft and planet can be considered as types, where the type of an object is called its *class*. Some classes, referred to as *subclasses*, may be specializations of other classes. For example, an interplanetary spacecraft is a specialization of the type spacecraft. As such, OOA organizes types into a class hierarchy based on a *isa* (as in "an X *is a* Y") relationship.

It may be natural to think of an object as being composed of other objects. For example, an interplanetary spacecraft may consist of numerous jets, guidance and navigation control system, and a probe to study a planet's atmosphere. This dependence introduces an additional dimension of relations into the class hierarchy, that is, a *part of* relation. The *parts of* an object are often called its *attributes*.

The nouns of the problem description can be used to identify candidate objects (and therefore, classes), and accordingly, the verbs in the problem description can provide information on interactions between objects. Some verbs may describe a service for a particular class of objects, such as *fire* in the phrase "fire the jets". Other verbs may describe a possible state of an object, such as *coast* in the phrase "the spacecraft begins to coast." Therefore, verbs help to define the services of a class of objects, usually referred to as the *operations* or *methods* of a class, and the computational processes of the system as a whole (the dynamic behavior).

In the early stages of software development, includig object-oriented approaches, diagrams are frequently used to describe requirements and guide development. For example, data flow diagrams (DFD) [25] have been widely used to visualize functional behavior of processes. Entity-relationship (E-R) diagrams [26] have been used to pictorially describe a wide variety of concepts, foremost among them is the relational data base organization.

In general, a single diagramming notation is not sufficient to capture the complex information needed to build software systems [27]. The *Object Modeling Technique* (OMT) [14] uses DFDs, hybrid E-R diagrams, and statecharts to model software requirements using object-oriented concepts. Collectively, these diagrams address properties that should be modeled, including flow of control, flow of data, patterns of dependency, time sequence, and name-space relationships. The OMT approach is appealing in its multiple views of software requirements and is fairly comprehensive in its (albeit informal) treatment of development issues. Furthermore, OMT is commonly used in industry and in academic settings.

3 Project Overview

A portion of the shuttle software was chosen for a formal methods demonstration project involving NASA's Jet Propulsion Laboratory, Johnson Space Center, and Langley Research Center [28]. This multi-NASA site project was supported as a *Research and Technology Objectives and Plans* (RTOP). A related project of a smaller scale was performed by the authors in conjunction with the larger demonstration project. The Phase_Plane module, the control system for automatic attitude control of the shuttle, was the subsystem selected for the smaller project. The criteria that led to the selection of Phase_Plane included finding a module with difficult to understand requirements and potential for critical change requests. Although the Phase_Plane module has worked correctly in thousands of hours of use in simulation and flight, its specific properties remains obscure (at least to the requirements analyst and software developers) [29].

Three tasks were performed in the development of the formal specifications of the module's high-level requirements. First, an understanding of the original requirements was needed. This involved consulting the *Functional Subsystem Software Requirements* (FSSR) document [30] (also known as Level C requirements, consisting largely of "wiring diagrams"), *Guidance and Control Systems Training Manual* [31], source code, informal design notes [32], and discussions with shuttle software personnel. An "as-built" formal specification capturing the functionality depicted by the FSSR "wiring diagrams" was then developed.

Second, when attempting to derive a more abstract requirements-level formal specification, it was difficult to eliminate the implementation bias present in the as-built layer. A level of OMT diagrams was developed to depict the information from the first level of specifications. These diagrams facilitated the abstraction process and lead to the next higher level of specifications. This iterative process consisting of developing a level of formal specifications, followed by constructing the corresponding OMT diagrams lead to the identification of the high level, critical requirements of the Phase_Plane module. Example specifications and OMT diagrams are described below.

The third task involved outlining proofs between the levels of specifications developed. That is, each specification must be shown to correctly implement the more abstract specification above it. These proofs provide traceability from the implementation details as described by the "wiring diagrams" to the high level requirements.

3.1 Phase Plane

The *Reaction Control System* (RCS) Digital Autopilot system (DAP) achieves and maintains attitude through an error correction method, involving the control of jet firings. Figure 1 gives a high-level view of the DAP, where the *State Estimator* gives the current attitude, while taking into consideration spacecraft dynamics. This information is then supplied to the Phase_Plane component that calculates the attitude and rate errors with respect to desired values specified by the crew.

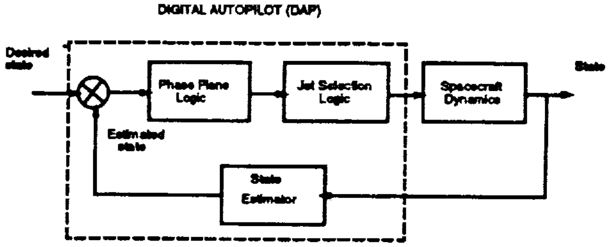


Figure 1: High-level view of DAP, including the Phase_Plane module [32]

A phase plane may be visualized as a graph plotting spacecraft rate errors against attitude errors for one rotational axis, with a "box" drawn around the center. There is a separate phase plane for each of the vehicle rotation axis (roll, pitch, and yaw). The "box" (with parabolic sides), whose limits are defined by the crew with attitude and rate deadbands, is used to determine when, if, and in what direction rates must be generated to null the errors [32]. If the shuttle is within the specified deadband limits, the rate and attitude errors are represented by a point plotted inside the box. If the point travels outside the box, then jets fire to return the point inside the box, thereby reducing the errors and achieving the maneuver request or maintaining the attitude hold as requested by the crew. Figure 2 gives a simplified graphical representation of the phase plane [30]. The shaded regions depict the *coast regions* where the Orbiter does not need any corrective action. The remaining regions are known as *hysteresis regions*, where external factors such as positive (negative) acceleration drift, propellant usage, inertia, time lags between firing commands, and sensor noise require the calculation of corrective action to ensure that the Orbiter remains within the deadband limits.

In an attitude hold situation, the error plot cycles around the zero error point with jets turning off and on again each time the limits of the "box" are exceeded. This activity is known as "limit cycling" or "deadbanding". The phase plane generates positive or negative rate commands on an axis by axis basis, where the jet select component determines which jet(s) to fire (the topic of the RTOP project [28]). The dashed lines outline the deadbanding path in Figure 2.

The requirements for the Phase_Plane module are described in terms of a "wiring" diagram (see Figure 3 [30]), indicating the input and output values, and several tables describing the calculation for the boundaries of the phase plane and its different regions.

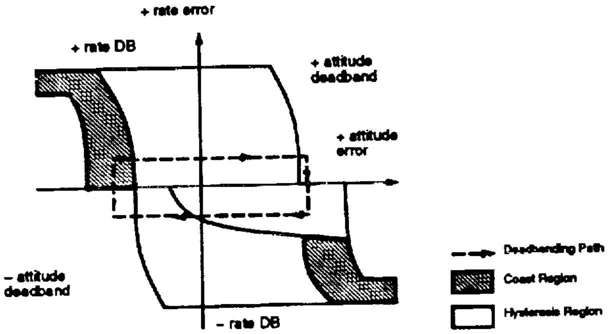


Figure 2: Graphical depiction of the phase plane, with coast and hysteresis regions [30]

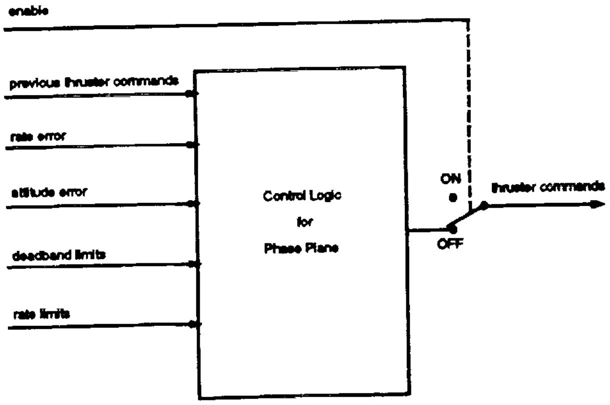


Figure 3: Simplified wiring diagram for the Phase_Plane module [30]

3.2 Formal Specifications

One aspect of formal methods for critical software development is the use of a particular rigorous notation to precisely define the function of the system and requirements that the system software *must* meet. These formal specifications are syntax- and type-checked using compiler-like parsers. This project used the *PVS* (Prototype Verification Systems) formal specification tools [33, 34] under development by SRI International. *PVS* is written in Common Lisp but runs on interpreters of other Lisp dialects. A *PVS* user, however, interacts with a customized Emacs [35] interface and needs no knowledge of Lisp.

Our goal was to specify Phase_Plane's functionality and execution constraints at several levels of abstraction. Specification of a system through increasingly more detailed levels of abstraction is a well-established strategy used by specifiers [15, 21]. Although these levels may appear almost disjoint, the proof of correct refinement of a level of specification by the level below assures the specifier the model is correct in addition to providing requirements traceability.

A general rule is that abstract, upper-level specifications should establish system inputs, outputs, and basic functionality of the system. Critical correctness requirements that the system must satisfy are stated at this level and become the criteria by which the specification is judged to be correct. Therefore, upper-level specifications tend to be black-box models of the system.

Mid-level specifications introduce both data type and functional detail that may constrain the eventual implementation of the system. These levels are the core of the specification since design decisions and and execution environment issues can be introduced. Change requests for modules will most likely be addressed in these levels.

A low-level ("as-built") specification is a straightforward representation of a particular implementation. It is from this detailed specification that source code can be automatically generated, or verification conditions for programmer-produced code derived.

The nature of Phase_Plane demanded a bottom-up approach instead of the top-down strategy described above. High-level English descriptions of this portion of the shuttle DAP were readily available, as was source code that had executed without error in hundreds of hours of use. This project explored the use of formal specifications to derive requirements that are more detailed and precise than an English paragraph and less obscure than tightly optimized source code.

A low-level formal specification was developed from the existing source code, the Crew Training Manual [31], and the low level "wiring diagrams" of data flow and formula tables. This specification mirrored the functionality of the existing system, but did not offer an abstract view of the module's functional requirements.

A high-level black-box specification was then developed corresponding to the level zero DFD (Figure 4). This formal specification did not include implementation details. At this level it was straightforward to state abstract properties that any software implementing Phase_Plane must have.

Finally, a mid-level formal specification was outlined to capture critical aspects of functionality and requirements at a level useful to shuttle "requirements analysts" when reviewing proposed modifications to the module. Due to time constraints, this level is still under development.

The challenge at the mid-level is to omit extraneous implementation details, yet be precise enough to capture necessary properties concerning minimization of fuel usage, thruster firings, and movement about the desired attitude. Included in this challenge iş the linkage of the three specification levels by proofs that trace abstract, critical properties from the top-level specification through the mid-level, and to the low "code-level" specification.

It should be noted that since the *PVS* environment is interactive, it is possible for a user to make a "claim" and attempt a proof of the claim immediately. This feature can be particularly useful when attempting to deduce requirements from a code-level specification. This tactic can also be used to "test" a specification interactively. A current NASA RTOP has documented other advantages of formal methods in general and *PVS* in particular [28].

3.3 Construction of OMT Diagrams

This section describes the OMT diagrams that have been generated thus far for the Phase_Plane module. Since we started the reverse engineering process with the source code and implementation specific wiring diagram of the Phase_Plane module, we created two levels of data flow diagrams depicting the flow of information into, from, and within the Phase_Plane. These diagrams assisted in the abstraction process to obtain an architectural view of the phase plane as it related to the overall DAP system, thus leading to the construction of the object models. The object and the functional models offered one level of abstraction, thus leading to the development of the next layer of formal specifications (mid-level specifications describing data structure and operations on the data structures). Finally, using the functional and object diagrams in conjunction with the description of the deadbanding states, we created the dynamic model for the Phase_Plane module. The dynamic model depicts the states between jet firings as the Orbiter deadbands. A high level of specifications was generated based on the dynamic model.

The remainder of this section describes the OMT diagrams constructed during the reverse engineering and formal specification construction process.

3.3.1 Functional Models

Data flow diagrams (DFD) facilitate a high level understanding of systems, both in terms of forward and reverse enginering. Static analysis of program code provides information that accurately describes flow of data in a system. In general, process bubbles denote procedures or functions of a given system. Arrows represent data flowing from one process to another. And rectangles represent external entities.

The simplest functional model (DFD) is a *context diagram* or Level 0 diagram and is shown in Figure 4, where the entire phase plane module is reduced to a process bubble, with the external input and output labeled. This diagram provides the context for the process in question. Note that the Level 0 DFD closely resembles the structure of the "wiring" diagram for Phase_Plane given in Figure 3.

The child diagram for Figure 4 gives the next level DFD, which shows the different processes making up the Phase_Plane module and is shown in Figure 5. In this figure, the input variables are used to calculate boundaries for the phase plane. The boundaries and the attitude and rate limits are supplied to the process that calculates the thrust commands (jet firings).

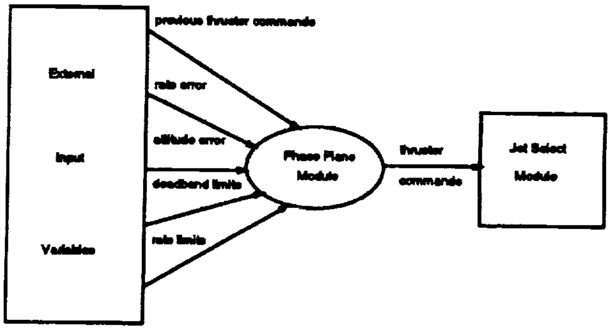


Figure 4: High Level (0) DFD for Phase_Plane Module

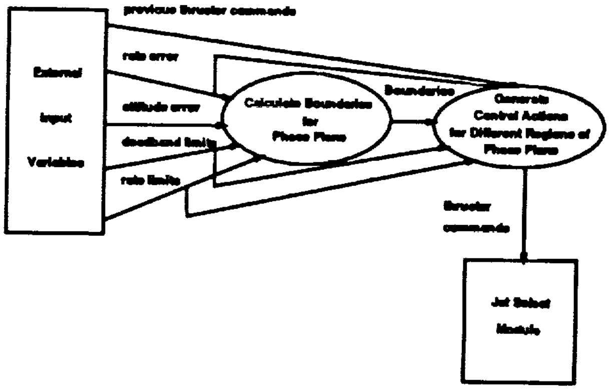


Figure 5: Level 1 DFD for Phase_Plane Module

3.3.2 Object Models

Studying the "as-built" layer of specifications, the different DFDs, and the requirements document for Phase_Plane led to the development of an object model for the Phase_Plane. As mentioned previously, an object is a self-contained module that includes both the data and procedures that act on that data. An object can be considered to be an abstract data type (ADT). A class is a collection of objects that have common use [36].

The object diagram for the Phase_Plane is shown in Figure 6. This diagram is a class entity with attributes *rate error*, *attitude error*, and *rotation axis*. The operation for this class is *calculate thrust commands* based on the rate and attitude errors. Also included in the object diagram are Phase Plane class instances (rounded rectangles) for each of the rotational axes (roll, pitch, and yaw). Each of the class instances will calculate different thrust commands for each of the specific rotational axes. Notice that there are two subclasses for the Phase Plane class, Coast Region and Hysteresis Region. In the coast region, the values of the attitude and rate errors are within acceptable bounds, thus there is no need to calculate new thrust commands. In the hysteresis region, however, the "Calculate new thrust commands" operation is inherited from the Phase Plane class.

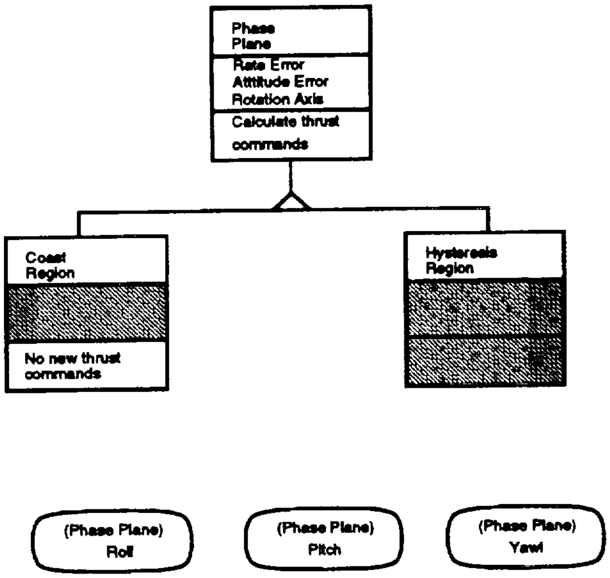


Figure 6: Object Model for Phase Plane Module

Next, we performed more abstraction steps in order to obtain a high-level object model for the DAP, consisting of the *State Estimator*, *Phase Plane*, and the *Jet Select Module*, corresponding to the diagram given in Figure 1. Figure 7 gives the object model for the DAP, where each class consists of three parts corresponding to the name of the class, list of attributes, and list of operations. The diamond symbol denotes aggregation, where the class above the diamond is said to consist of the three classes below the diamond. If either attributes or operations are not known (or do not exist) for a given class, then the corresponding area is shaded.

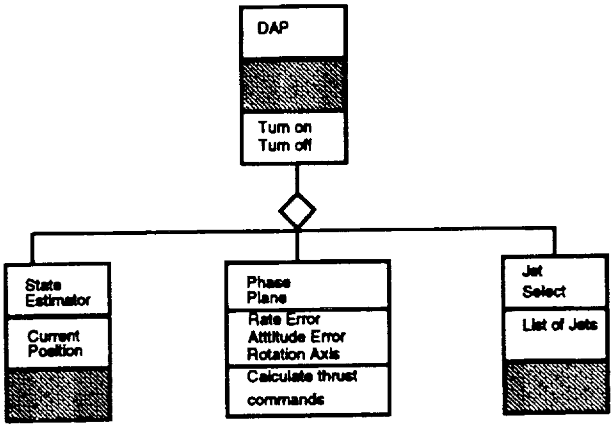


Figure 7: High Level Object diagram for DAP

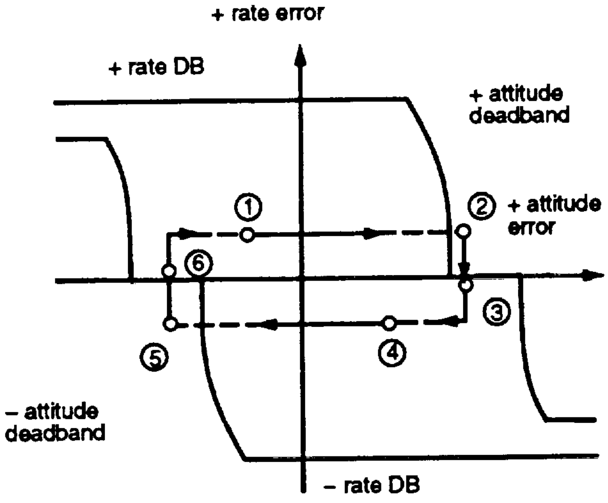


Figure 8: Graphical depiction of the phase plane, with deadbanding cycles [31]

3.3.3 Dynamic Models

This section gives the dynamic models for the phase plane, which describes the states in which the DAP can be with respect to the Phase_Plane component. Also, included are the transitions that take the DAP from one state to another. A pictorial diagram of the envelope depicting the position of the Orbiter is given in Figure 8. The "○" plots the current vehicle attitude and rate errors with respect to the phase plane. As long as the current position is within the limits imposed by the deadbands (the heavy lines), the deadband constraints are satisfied and no jets will be commanded to fire. Once the Orbiter exceeds the bounds of the "box", jets will be commanded to fire in an effort to cancel the errors, thereby reducing the errors and achieving the requested maneuver or maintaining the attitude hold, whichever was requested by the crew. Once the Orbiter returns to the deadband area, the jets will stop firing.

Figure 9 gives an explanation of the different states in which the Orbiter can be while it is deadbanding [31]. Figure 10 gives a statechart depiction of the states through which the Orbiter transitions while it is deadbanding. The state transitions are in the form of jets terminate (begin) firing and the Orbiter drifting in (out) of the deadband region.

Note that Figure 8 depicts the clockwise traversal of the states in which the Orbiter cycles through the deadband limits. It is also possible for the Orbiter to traverse the cycle in a counterclockwise fashion, in which case, the arrows in Figure 10 would be reversed.

Finally, a very high-level view of the states in which the Orbiter can be is given in Figure 11. Included

1. No jets fire. Since the rate error is positive, the attitude error will grow in a positive direction.
2. Jets fire to nullify the positive rotational rate.
3. Jets stop firing when the deadband line is crossed, but a little negative rate errors is inevitable.
4. No jets fire. With a negative rate error, the attitude error will also drift negatively.
5. Jets fire to nullify negative rate error.
6. Jets stop firing, but residual positive rate error causes attitude error to go positive again and the cycle repeats.

Figure 9: Explanation of deadbanding states [31]

in the diagram are the actions or conditions that cause the Orbiter to transition from one state to the next. The rectangle containing "Phase Plane" and the labeled arrows pointing to the states indicate that the state transitions describe the `Phase_Plane` module.

4 Lessons Learned

The results from this reverse engineering project have provided several lessons for the overall project as well as for future reverse engineering projects. First, in order to obtain high-level requirements for existing software, it is not feasible to obtain the specifications (formal or informal) in one step. Instead, several layers of specifications must be developed, starting with the "as-built" specification. The "as-built" specification closely mirrors the programming structure of the existing software in order to provide traceability through the different levels of specifications. After creating the levels of specifications, theorems need to be constructed to demonstrate that critical properties are preserved from one level of specification to the next.

Second, formal specification languages and their corresponding reasoning systems provide a mechanism for bringing together disparate sources of project information into one integrated framework. In particular, the project information may be in a variety of formats, from different sources, and subjected to varying levels of formal review. For this particular project, information was obtained from the *Functional Subsystem Software Requirements* (FSSR) document [30] (also known as Level C requirements, consisting largely of "wiring diagrams"), *Guidance and Control Systems Training Manual* [31], source code, informal design notes [32], and discussions with shuttle software personnel. Accordingly, formal specifications were constructed based on all of the information in order to describe the phase plane operation. The *PVS* proof system provided a mechanism for checking the completeness and consistency of the specifications, while also supporting the proof construction of the relevant theorems.

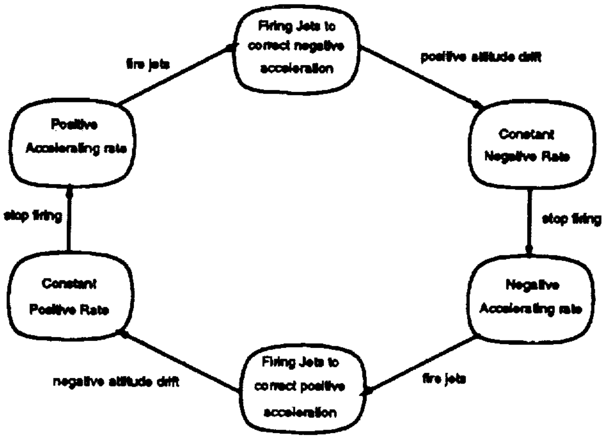

Firing Jets to correct negative acceleration
positive attitude drift
Constant Negative Rate
stop firing
Negative Accelerating rate
fire jets
Firing Jets to correct positive acceleration
negative attitude drift
Constant Positive Rate
stop firing
Positive Accelerating rate
fire jets

Figure 10: States representing the clockwise deadbanding of the Orbiter

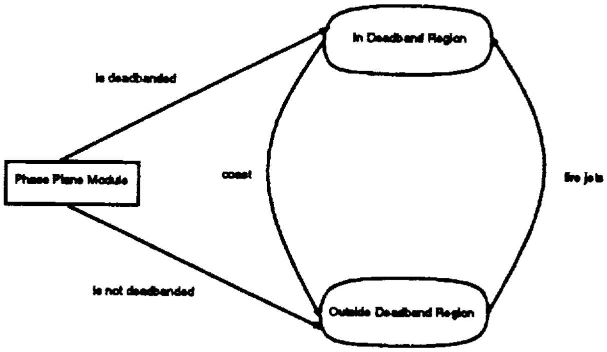


Figure 11: High-level states for Orbiter with respect to the `Phase_Plane` module

Third, the benefits of object-oriented analysis and design can be exploited for reverse-engineering as well as forward engineering projects. Specifically, object-oriented analysis and design assists in the understanding and the simplification of the complexity of a large system. Furthermore, having an object-oriented perspective facilitates future modifications by providing the developer with a high-level, abstract view of system components, thus avoiding the difficulties associated with attempting to understand all of the details of a large, complex system at once.

Finally, an iterative process consisting of the construction of a level of formal specifications, followed by a set of corresponding diagrams is needed to develop several layers of specifications for an existing system. The diagrams introduce abstractions that can be used to guide the construction of the next level of specifications. Furthermore, the complementary diagrams available in the OMT approach enable the specifier to consider different perspectives of the system with notations best suited for the respective perspective. The major advantage to this diagramming approach is that one notation does not consist of many different symbols in an attempt to capture very different aspects of a system, which would make it too complex to use effectively.

5 Conclusions and Future Investigations

Using formal specifications and object-oriented analysis to describe the software that implements the Phase_Plane module of the DAP has demonstrated that this rigorous technology can be used for existing, industrial applications. Constructing the different levels of specifications, with increasing abstraction, supplemented by the OMT diagrams provided a means for integrating information regarding the Phase_Plane module from disparate sources. Having access to this information will facilitate the verification that the original (critical) requirements or properties are not violated by any future changes to the software. In addition to facilitating verification tasks, the formal specifications can be used as the basis for any automated processing of the requirements, including checks for consistency and completeness. Interaction with the requirements analyst and other members of the original development team for the project strongly support the conclusion that the specification construction process, in addition to the actual specifications are useful to the overall software development and maintenance processes of existing (safety-critical) systems.

Future investigations will continue to refine the mid-level and high-level specifications and develop more theorems to relate the different levels of specifications. We are also investigating the formalization of the OMT diagramming notation, which will provide a means for using automated techniques for extracting formal specifications from the OMT diagrams in order to facilitate the specification process. Furthermore, extracting the specifications directly from the diagrams will allow us to reason about the completeness and consistency of the diagrammed system, thus greatly facilitating the requirements analysis, design, and maintenance phases of software development.

6 Acknowledgements

Several people have provided valuable information and assistance during the course of the project. Specifically, we would like to thank David Hamilton, John Kelly, Rick Covington, and John Rushby.

References

[1] Nancy G. Leveson and Clark S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, pages 18–41, July 1993.

[2] Bev Littlewood and Lorenzo Strigini. The risks of software. *Scientific American*, pages 62–75, November 1992.

[3] Eric Schmitt. Army is blaming patriot's computer for failure to stop dharan scud. *New York Times*, May 1991.

[4] P. G. Neumann and contributors. Risks to the public. In *Software Engineering Notes*. ACM Special Interest Group on Software Engineering, 1993. Regular column published on a monthly basis.

[5] Victoria Slid Flor. Ruling's Dicta Causes Uproar. *The National Law Journal*, July 1991.

[6] Aeronautics and Space Engineering Board National Research Council. *An Assessment of Space Shuttle Flight Software Development Practices*. National Academy Press, 1993.

[7] Susan L. Gerhart. Applications of formal methods: Developing virtuoso software. *IEEE Software*, 7(5):7–10, September 1990.

[8] Nancy G. Leveson. Formal Methods in Software Engineering. *IEEE Transactions on Software Engineering*, 16(9):929–930, September 1990.

[9] Richard A. Kemmerer. Integrating Formal Methods into the Development Process. *IEEE Software*, pages 37–50, September 1990.

[10] Susan Gerhart, Dan Craigen, and Ted Ralston. An international study of industrial applications of formal methods. Technical report, NIST,NRL, and Atomic Energy Control, 1992.

[11] Betty H.C. Cheng. Synthesis of Procedural Abstractions from Formal Specifications. In *Proc. of COMPSAC'91*, pages 149–154, September 1991.

[12] Jun jang Jeng and Betty H.C. Cheng. Using Automated Reasoning to Determine Software Reuse. *International Journal of Software Engineering and Knowledge Engineering*, 2(4):523–546, December 1992.

[13] Betty H.C. Cheng. Applying formal methods in automated software development. *accepted to appear in Journal of Computer and Software Engineering*, 1993.

[14] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[15] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.

[16] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[17] John Rushby. Formal methods and the limits of dependability. In *Proceedings of Foundations of Theoretical Computer Science (FTCS 23)*, Toulousse, France, June 1993. Position paper for Panel.

[18] Betty H.C. Cheng and Gerald C. Gannod. Constructing formal specifications from program code. In *Proc. of Third International Conference on Tools in Artificial Intelligence*, pages 125–128, November 1991.

[19] Gerald C. Gannod and Betty H.C. Cheng. A two-phase approach to reverse engineering using formal methods. In *Lecture Notes in Computer Science, Proc. of Formal Methods in Programming and Their Applications Conference.* Springer-Verlag, June 1993.

[20] M. Ward, F.W. Calliss, and M. Munro. The maintainer's assistant. In *Proceedings Conference on Software Maintenance*, pages 307–315, Miami, Florida, October 1989. IEEE.

[21] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., second edition, 1990.

[22] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[23] D. R. Smith. KIDS: A Semi-automatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[24] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw Hill, third edition, 1992.

[25] Alan M. Davis. *Software Requirements, Analysis and Specification.* Prentice-Hall, Inc., 1990.

[26] P. Chen. The entity relationship model: Toward a unifying view of data. *ACM Transactions on Database Systems 1*, pages 9–36, March 1977.

[27] F.P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, April 1987.

[28] Jet Propulsion Laboratory and Johnson Space Center and Langley Research Center. Formal Methods Demonstration Project for Space Applications: Phase I Case Study: STS Orbit DAP Jet Select. Research and Technology Objectives and Plans (RTOP), December 1993.

[29] David Hamilton. Discussion of phase plane requirements. Private Communication, August 1993. Hamilton is a software/knowledge engineer in the Advanced Technology Department of the Federal Sector Division for IBM-Houston working in conjunction with Johnson Space Center on the shuttle project.

[30] Space Shuttle Orbiter Operational Level C Functional Subsystem Software Requirements: Guidance Navigation and Control – Part C Flight Control Orbit DAP. Technical Report OI-21 edition, Rockwell International, Space Systems Division, February 1991.

[31] Sara Beck. G & C Systems Training Manual: Guidance and Flight Control – Insertion, Onorbit and Deorbit. Technical Report I/O/D G&C2102, Mission Operations Directorate, Training Division, Flight Training Branch, October 1985.

[32] D. Johnson and R. Davison. Phase Plane Logic Design. JSC Memorandum concerning details of the design of the Phase Plane Logic., May 1986.

[33] N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker. Reference manual, March 1993.

[34] N. Shankar, S. Owre, and J.M. Rushby. The PVS specification language. Technical report, March 1993.

[35] Richard Stallman. *GNU Emacs Manual*, fifth edition, 1986.

[36] Ann L. Winblad, Samuel D. Edwards, and David R. King. *Object-Oriented Software.* Addison-Wesley, Publishing Company Inc., 1990.

Applying Formal Methods and Object-Oriented Analysis to Existing Flight Software

Betty H.C. Cheng
Department of Computer Science
Michigan State University
East Lansing, Michigan 48824-1027
ph: (517) 355-8344; fax: (517) 336-1061
email: chengb@cps.msu.edu

Brent Auernheimer
Department of Computer Science
California State University
Fresno, California 93740-0109
ph: (209) 278-2573; fax: (209) 278-4197
email: brent_auernheimer@csufresno.edu

18th Annual Software Engineering Workshop, Dec. 1-2, 1993

Background for Project

- Integrate formal methods to portion of shuttle software
- Construct an object-oriented view of system
- Demonstrate the numerous utilities of formal methods in software development
- Facilitate current and future maintenance
  "Due to careful review of changes, it takes an average of 2 years
  for a new requirement to get implemented, tested,
  and into the field."
- Facilitate verification of safety-critical properties
- Address one major issue encountered in industry:
  reverse engineering of existing (legacy) system.

Formal Methods

- What is a formal method?
  - Formal languages with well-defined syntax
  - Well-defined semantics
  - Proof systems
- Why use Formal Methods?
  - Improve quality of software systems
  - Reveal ambiguity, incompleteness, and inconsistency in a system
- Important Characteristics:
  - Abstraction
  - Proof obligations
  - Tool support
  - Systematic Process

Software Engineering Workshop (12/93)-1

Object-Oriented Software

- Represent real-world problem domain and maps it into software solution domain
- OO Design interconnects data objects and processing operations
- Modularizes information and processing, not just processing
- Three Main concepts:
  - Abstraction
  - Information hiding
  - Modularity

Software Engineering Workshop (12/93)-2

Object Modeling Technique

- Three diagramming notations give complementary perspectives of system
  - *Object Model* presents the architectural view (traditional object-oriented diagramming notation)
  - *Functional Model* presents a functional view (data flow diagrams)
  - *Dynamic Model* presents the behavioral view (state diagrams)
- More amenable to formalization than other OO diagramming notations
- Widely used in industry and universities, including IBM at JSC.

"What would help me do RA for Orbit DAP"

It is highly unlikely that we'll find a product that will understand shuttle requirements. Some degree of customization will need to be performed in whatever tools we choose to support our formal methods activities.

- From the beginning, shuttle requirements authors were given the freedom to express requirements in whatever form they preferred.
- Consequently, the shuttle requirements are a combination of many formats, styles, conventions, and perspectives.
- It has historically been very difficult to insert new technologies into the shuttle program.
- Any tool that takes steps to the existing shuttle requirements or automatically convert the existing requirements into a format it can understand will be much more likely to succeed.

Project Selection Criteria

- Current RTOP is demonstration project:
  - *Jet Select* module for space shuttle
  - Determine which jets should be fired to achieve desired position(s)
  - Select module that is accommodating *Change Requests*
- Faculty Fellowship project complements RTOP project
  - *Phase Plane* module: control system for monitoring angular rotation
  - Determines amount of corrective action needed

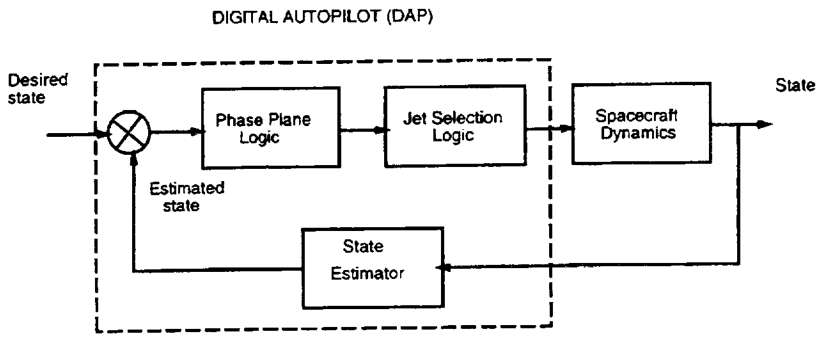
Software Engineering Workshop (12/93)-7

Phase Plane

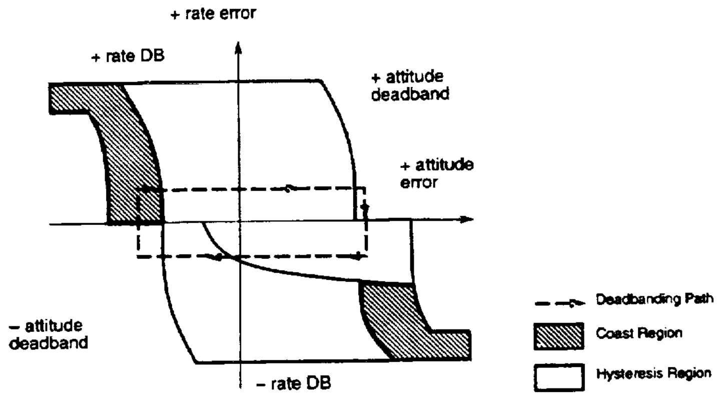
- JSC expressed keen interest in Phase Plane module
  - JSC had difficulty fully understanding module
  - Difficulty in testing module
  - Will need to make changes in future
  - Results feed directly to Jet Select module
- Phase Plane applicable to other spacecraft
- Main component of control system
  - Uses thrusters to control angular state of spacecraft
  - Monitor state errors
  - Determine when and how corrective control should be applied

Software Engineering Workshop (12/93)-8

Pictorial View of DAP Control Loop



Graphical Representation of Phase Plane

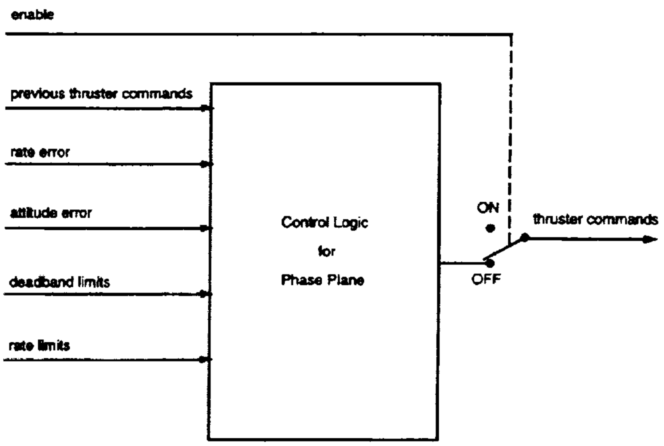


Preliminary Tasks

- Learn new specification language (PVS), including support tools.
- Become familiar with Jet Select and Phase Plane domain
  - *Functional Subsystem Software Requirements* (wiring diagrams)
  - *Crew Systems Training Manual*
  - Informal requirements discussions from JSC, IBM, Draper Labs (software designers), including site visit to IBM at JSC.
  - Informal design notes
- Become familiar with commonly used object-oriented diagramming technique and support tools.

Software Engineering Workshop (12/93)-11

Wiring Diagram for `Phase Plane`



Software Engineering Workshop (12/93)-12

Project Overview

- Apply reverse engineering techniques
- Develop levels of specifications
- Each level is more abstract than previous
- Objective: obtain a high-level specification of requirements
- Identify and prove critical properties that link the levels.
- Develop an OMT hierarchical "roadmap" of module
- Establish linkage between specifications and OMT diagrams.

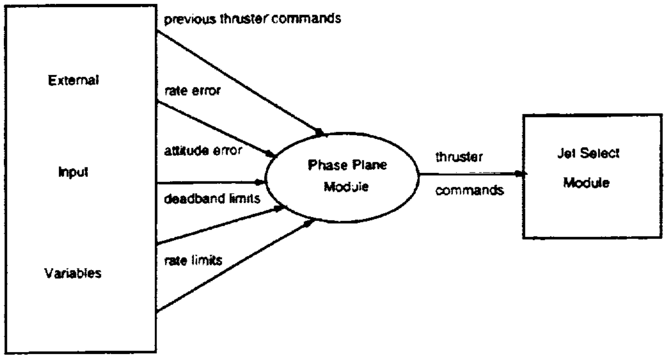
Software Engineering Workshop (12/93)-13

Iterative Process

- Construct low-level specifications correspond to wiring diagrams
- Use code for clarification
- Construct OMT diagrams for wiring diagrams
- Identify properties required for system.
- Construct high-level specifications for properties of Phase Plane
- Construct high-level OMT diagrams that apply to Phase Plane
- Integrate specifications with OMT diagrams.

Software Engineering Workshop (12/93)-14

Functional Model: Level 0 (context) DFD



External

Input

Variables

previous thruster commands

rate error

attitude error

deadband limits

rate limits

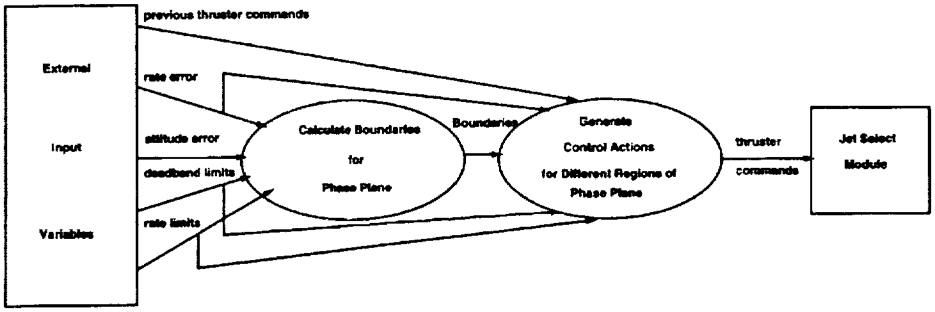
Phase Plane Module

thruster commands

Jet Select Module

Software Engineering Workshop (12/93)-15

Functional Model: Level 1 DFD



External

Input

Variables

previous thruster commands

rate error

attitude error

deadband limits

rate limits

Calculate Boundaries for Phase Plane

Boundaries

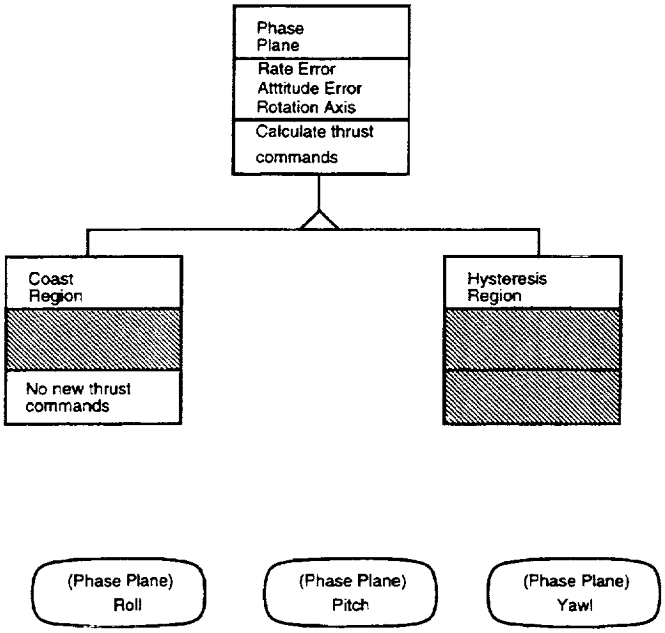
Generate Control Actions for Different Regions of Phase Plane

thruster commands

Jet Select Module

Software Engineering Workshop (12/93)-16

Object Model for Phase Plane



Phase Plane

Rate Error
Atttitude Error
Rotation Axis

Calculate thrust commands

Coast Region

No new thrust commands

Hysteresis Region

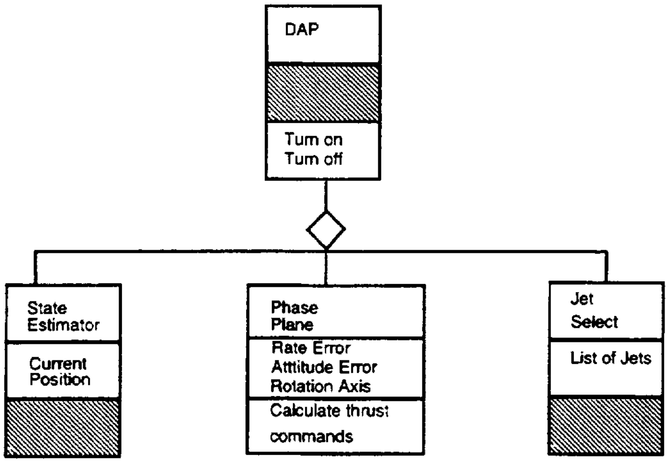
(Phase Plane) Roll

(Phase Plane) Pitch

(Phase Plane) Yawl

Software Engineering Workshop (12/93)-17

Object Model for DAP



DAP

Tum on
Tum off

State Estimator

Current Position

Phase Plane

Rate Error
Atttitude Error
Rotation Axis

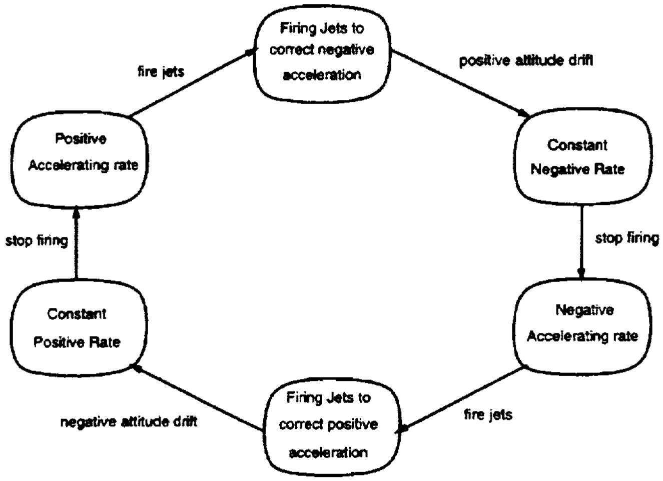
Calculate thrust commands

Jet Select

List of Jets

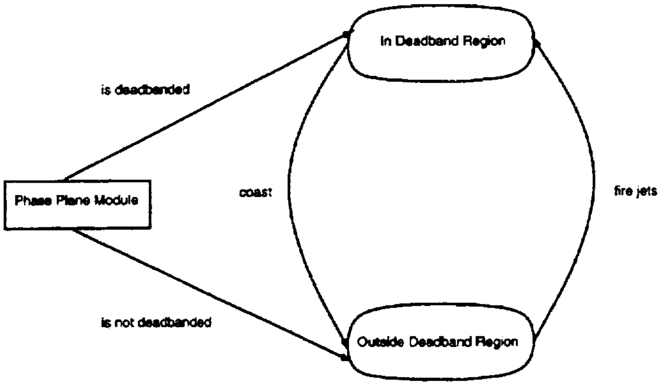
Software Engineering Workshop (12/93)-18

Dynamic Model for Deadbanding


Firing Jets to correct negative acceleration
positive attitude drift
Constant Negative Rate
stop firing
Negative Accelerating rate
fire jets
Firing Jets to correct positive acceleration
negative attitude drift
Constant Positive Rate
stop firing
Positive Accelerating rate
fire jets

Software Engineering Workshop (12/93)-19

More Abstract Dynamic Model for Deadbanding



Software Engineering Workshop (12/93)-20

Lesson I

- More than one step from high-level requirements to existing code.
- Must create several layers of specifications
- "As-built" layer closely mirrors code (traceability)
- Need to construct theorems relating layers of specifications

Software Engineering Workshop (12/93)-21

Lesson II

- Formal methods provide mechanism for integrating disparate sources of project information.
- Project information may be:
  - in a variety of formats,
  - subjected to varying levels of formal reviews
  - located physically apart
- Examples include:
  - Functional Subsystem Software Requirements ("wiring diagrams")
  - Crew Training Manual
  - Design notes
  - Discussions with shuttle software personnel.
- Use formal specifications to integrate information from different sources.

Software Engineering Workshop (12/93)-22

Lesson III

- Object-oriented analysis and design can be exploited for reverse engineering tasks.
- OO introduces abstraction to simplify complexity of system
- OO perspective can facilitate future maintenance tasks

Software Engineering Workshop (12/93)-23

Lesson IV

Reverse engineering process is iterative

- Construct level of formal specifications
- Create a set of diagrams (introduces abstraction)
- Repeat.

Software Engineering Workshop (12/93)-24

Summary

- Incorporate formal methods into existing system
  - Assist maintainers in understanding module
  - Facilitate future changes
  - Facilitate verification of critical properties
- Develop reverse engineering process using FM and OO
- Develop OMT models usable by RTOP project
- Identify obstacles (and solutions) in abstraction (reverse engineering) process usable by RTOP project
- Demonstrate utility of FM and OO on real project.

Software Engineering Workshop (12/93)-25

Current and Potential Future Tasks

- Develop mid-level specifications
- Construct multi-level correctness proofs
- Demonstrate how FM can be used to gain confidence in the correctness of software after modification using critical correctness criteria and proofs.
- Integrate more closely the formal specifications with OMT diagrams.

Software Engineering Workshop (12/93)-26

Acknowledgements

- The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the National Aeronautics and Space Administration. Additionally, the authors' work on this project was supported by NASA/ASEE Summer Faculty fellowships.

- Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

INTEGRATING END-TO-END THREADS OF CONTROL INTO OBJECT-ORIENTED ANALYSIS AND DESIGN

Janet E. McCandlish
TRW
213 Wynn Drive
Huntsville, Ala 35805

Dr. James R. MacDonald
ALPHATECH, Inc.
50 Mall Road
Boston, Mass 01803

Dr. Sara J. Graves
University of Alabama
in Huntsville
Alumni House, Room 102
Huntsville, Ala 35899

ABSTRACT

Current object-oriented analysis and design methodologies fall short in their use of mechanisms for identifying threads of control for the system being developed. The scenarios which typically describe a system are more global than looking at the individual objects and representing their behavior. Unlike conventional methodologies that use data flow and process-dependency diagrams, object-oriented methodologies do not provide a model for representing these global threads end-to-end.

Tracing through threads of control is key to ensuring that a system is complete and timing constraints are addressed. The existence of multiple threads of control in a system necessitates a partitioning of the system into processes. This paper describes the application and representation of end-to-end threads of control to the object-oriented analysis and design process using object-oriented constructs. The issue of representation is viewed as a grouping problem, that is, how to group classes/objects at a higher level of abstraction so that the system may be viewed as a whole with both classes/objects and their associated dynamic behavior. Existing object-oriented development methodology techniques are extended by adding design-level constructs termed logical composite classes and process composite classes. Logical composite classes are design-level classes which group classes/objects both logically and by thread of control information. Process composite classes further refine the logical composite class groupings by using process partitioning criteria to produce optimum concurrent execution results. The goal of these design-level constructs is to ultimately provide the basis for a mechanism that can support the creation of process composite classes in an automated way. Using an automated mechanism makes it easier to partition a system into concurrently executing elements that can be run in parallel on multiple processors.

INTRODUCTION

The philosophy upon which object-oriented analysis and design is based does not lend itself well to the representation of how a system operates as a whole. The object-oriented premise emphasizes the extraction of objects to be modeled from the problem domain in contrast to traditional methods which rely on the functionality of the system. A review of some of the more current object-oriented analysis and design methodologies highlights this shortcoming by these methodologies' inability to effectively represent the end-to-end processing of a system. A global representation is key to understanding how the system operates. As described in [Fichman], conventional methodologies use tools such as data flow and process-dependency diagrams for representing global threads end-to-end, but object-oriented methodologies have nothing comparable. Because conventional methods stress functionality over object partitioning, showing the operation of the overall system via functions is consistent with the methodology approach in general. In contrast, object-orientation concentrates on objects as stand-alone reusable components instead of how those components tie together. Object-oriented methodologies partition objects and their relationships into several models which represent different views of the objects and their interactions. These views are generally presented in the form of static architectures and dynamic behavior. There are typically multiple instances of each view, with each instance representing a fragment of the system. The observer must deal not only with these multiple views of the system, but also with fragments of the system at a time. Much effort is required to obtain a synergistic understanding of the system being modeled as a whole.

One of the key mechanisms for ensuring system completeness is to trace through threads of control. A thread of control is a path through a sequence of operations representing a particular scenario in the system being modeled. Threads of control integrate the overall flow of data, control, events, and timing up to the system level. They provide a means by which the system may be analyzed and understood as a whole. Thread of control information is desirable in two ways: first, it ensures that all of the pertinent objects exist to support the system as a whole; and secondly, if timing is critical in the system, tracing through the threads of control may identify essential timing constraints and potential bottlenecks. The presence of multiple threads of control is an indicator that the system will need to be partitioned into processes, that is, separate executable entities. Identification of these critical areas early on will drive decisions concerning process allocation, and how data will be transferred, accessed, and shared. Because of their significance, successfully representing threads of control for a system being modeled greatly enhances understanding the operation of the system as a whole.

Viewing a system in terms of the processes which make it up adds additional complications. On the one hand, large-scale real-time distributed systems reconcile competing demands for resources by partitioning the system into multiple processes. On the other hand, object-oriented technology strives to partition a system by objects where all data and operations associated with an object are encapsulated within the object. The partitionings for processes and objects appear to be orthogonal in this context when threads of control are considered. Hence, the partitioning goals associated with object-oriented and distributed systems are conflicting.

This paper introduces a means of representing threads of control and their associated classes/objects to better illustrate how the system operates. Towards this end, an analysis of five predominant object-oriented analysis and design methodologies was performed.

The methodologies reviewed include Coad and Yourdon [Coad91a] [Coad91b], Shlaer and Mellor [Shlaer88] [Shlaer91], Booch [Booch], Firesmith [Firesmith], and Rumbaugh [Rumbaugh]. While some of the methodologies reviewed describe both analysis and design (Coad and Yourdon, Firesmith and Rumbaugh), Shlaer and Mellor focus more on analysis and Booch on design. The distinction between object-oriented analysis and design is not precise. There are inconsistencies in the research about what comprises each, and the lines between analysis and design in object-orientation are blurred [Berard], [Korson]. It is not the goal of this paper to distinguish between object-oriented analysis and design. Instead, the intent is to focus on the constructs necessary to support end-to-end processing during object-oriented analysis and design as opposed to object-oriented programming.

A review of methodologies indicates that both static (class/object architecture) and dynamic (control and data flow) representations of systems exist; however, threads of control are only minimally represented and are fragmented. This paper extends the static and dynamic concepts by introducing a representation which overlays dynamic flow (via thread of control information) onto a static structure. In order to combine dynamic and static representations to show end-to-end processing, class/objects are grouped so that they may be represented at a higher level of abstraction. Determining how the class/objects were to be grouped resulted in a partitioning problem. To simplify the partitioning problem, the proposed grouping approach is performed in two phases. The first phase involves a logical grouping of class/objects. The logical groupings are further refined with thread of control (state, control, and data flow) information, providing a coarse-grained partitioning referred to as *logical composite classes*. The second phase further extends the partitioning using process partitioning criteria based on other thread of control information involving communication and timing constraints to develop *process composite classes*. The introduction of these design-level constructs provides the basis for a mechanism to automate different instances of composite process classes for timing and concurrency comparisons.

ANALYSIS OF CURRENT OBJECT-ORIENTED ANALYSIS AND DESIGN METHODOLOGIES

Each of the five methodologies reviewed provided some means of representing both the static architecture and dynamic behavior of a system. The following is a brief description of the techniques each methodology employs for representing static and dynamic views of a system and an overall assessment of these techniques.

Static Architecture

The static architecture refers to a non-temporal representation of the system. A static representation of the system is generally reflected by some variation of entity-relationship diagram. Entities, in this context, are either classes or objects. The distinction between classes and objects is that a *class* serves as a template for defining the characteristics of an object. An *object* is a software abstraction that models a concept, abstraction, or thing which represents the application domain (analysis) or the solution space (design). To further distinguish the two, an object is an instance of a class. Further, a *concrete class* is a class for which object instances may be created, as opposed to an *abstract class* for which objects may not exist.

The static models and diagrams associated with the methodologies reviewed are summarized in Figure 1. The diagrams for each methodology which depict classes/objects and their relationships are those listed first in the Class/Object Representation row of Figure 1. These diagrams, when used in a general context, will be termed class/object diagrams since they generally contain more information than what is usually associated with an entity-relationship diagram as is described below.

	Coad and Yourdon	Shlaer and Mellor	Booch	Firesmith	Rumbaugh
Class/ Object Model(s)	• Class-&-Object Layer • Structure Layer • Attribute Layer • Service Layer	• Information Model	• Class Structure • Object Structure	• Class Model • Object Model	• Object Model
Class/ Object Represent-ation(s)	• Class-&-Object Diagram • Gen-Spec Structure • Whole-Part Structure	• Information Structure Diagram • Inheritance Diagram	• Class Diagram • Object Diagram • Class Template	• General Semantic Net • Interaction Diagram • Classification Diagram • Composition Diagram • Class Specification	• Object Diagram • Generalization Notation • Aggregation Notation
The above representations all include classes/objects, relationships, and attributes. All include operation specifications with the exception of Shlaer and Mellor.					

Figure 1. Static Models and Diagrams

An object, or the class template for the object, is usually defined in terms of its attributes and operations. *Attributes* are fields which describe data values within a class/object, and *operations* are functions performed by a class/object. Two methodologies, Coad and Yourdon and Firesmith, represent attributes and operation specifications on their Class-&-Object Diagram and Object Diagram respectively. Shlaer and Mellor include only attributes on their Information Structure Diagram. Both Booch and Firesmith use a separate means for representing attributes and operation specifications. Booch describes a Class Template, and Firesmith a Class Specification.

Relationships in a class/object diagram refer to associations between two or more classes/objects indicating some type of structural or semantic link. In addition to simple association, two special types of relationships exist in most object-oriented methodologies: *is-a* and *has-a* relationships.

Is-a relationships introduce the concepts of generalization, specialization, and inheritance. A *generalization* is a higher level of abstraction of a class. For example, the class *animal* is a generalization of the classes *cat* and *dog*. Because it is a generalization, animal is a *superclass* of cat and dog. Animal might be described as furry and four-legged. While both the cat and dog are furry and four-legged, the cat meows and the dog barks. Because they are *specializations* of animal, cat and dog are *subclasses* of animal; they *inherit* the characteristics of being furry and four-legged, but they extend the animal class by adding special characteristics such as meowing or barking. Specifically, a subclass inherits the attributes and operations of its superclass, and extends it further with additional attributes or operations. Several of the methodologies contained special diagrams to represent the *is-a* relationship: Coad and Yourdon/Gen-Spec Structure, Shlaer and Mellor/Inheritance Diagram, Firesmith/Classification Diagram, and Rumbaugh/Generalization Notation.

Has-a relationships depict an *aggregation* of class/objects. A class/object which contains at least one other class/object is referred to in this paper as a *composite class/object*. For example, the composite class/object *car* is made up of doors, wheels, an engine, etc. Conversely, a class/object which does not contain other classes/objects is termed an *atomic class/object*. Several methodologies represented aggregation associations with special notation: Coad and Yourdon/Whole-Part Structure, Firesmith/Composition Diagrams, and Rumbaugh/Aggregation Notation.

Dynamic Behavior

Dynamic behavior is behavior attributable to timing and the flow of information in the system being modeled. The information typically represented in dynamic models includes state, control and data flow, and timing information. The *states*, or modes, of a class/object reflect the attribute values of a class/object at a given point in time. *State information* contains the states of a specific class/object, and the operations or events that effect transitions between the class/object's states. State transition diagrams (STDs) are the most common representation of state information. STDs are generated for each class/object which has interesting behavior. All of the STDs in the methodologies reviewed contained states, events, transitions and operations with the exception of Coad and Yourdon's Object State Diagram which contained only states and transitions. *Control flow information* describes the control and sequencing of a message within or between classes/objects. It is most often represented in a control flow diagram (CFD). A *message* may be a request for service, event, or passing of data. *Data flow information* describes the flow of data among classes/objects via their operations. A data flow diagram (DFD) is commonly used to show data flow among the class/objects. *Timing information* contains the duration of operations within and between class/objects and is usually associated with control flow information.

Control flow information is only minimally represented on the Coad and Yourdon Class-&-Object diagram via arrows between the class/objects which represent message connections. Booch shows control flow only in the context of a Timing Diagram which displays objects and the invocations of their operations along a time axis. Rumbaugh's primary mechanism for control flow is his State Diagram; although control may also be shown on a DFD but is considered redundant. Both Shlaer and Mellor and Firesmith combine control and data flow information onto one diagram, the Action DFD and Object-Oriented CFD respectively. Neither Coad and Yourdon or Booch describe representations for data flow. For timing information, only Booch (as previously mentioned) and Firesmith provide a timing diagram. Coad and Yourdon allow that a time requirement may be annotated with the specification of a particular class/object. Shlaer and Mellor describe time only in the context of threads of control which is addressed later in this paper.

The dynamic models and diagrams associated with the methodologies reviewed are summarized in Figure 2. The first bullet in each cell lists the model used to address each type of dynamic information, and the second bullet lists the diagram (or diagrams) the methodologies use to represent information.

	Coad and Yourdon	Shlaer and Mellor	Booch	Firesmith	Rumbaugh
State Information	• Services Layer • Object State Diagram	• State Model • STD	• Class Structure • STD	• State Model • STD	• Dynamic Model • State Diagram
Control Flow Information	• Services Layer • Message Connections on Class-&-Object Diagram	• Process* Model • Action DFD	• Object Structure • Timing Diagram	• Control Model • Object-Oriented CFD	• Dynamic/Functional Models • State Diagram/ DFD
Data Flow Information	• N/A	• Process* Model • Action DFD	• N/A	• Control Model • Object-Oriented CFD	• Functional Model • DFD
Timing Information	• Services Layer • Timing Textual Annotation in Class-&-Object Specification	• N/A	• Object Structure • Timing Diagram	• Timing Model • Timing Diagram	• N/A

STD: State Transition Diagram, DFD: Data Flow Diagram, CFD: Control Flow Diagram

*Process is a *transform* in this context.

Figure 2. Dynamic Models and Diagrams

Dynamic behavior is the basis of information upon which threads of control are built; however, dynamic behavior models only fragments of the system. State information is associated with a particular class/object. Control and data flows are usually represented between a particular group of classes/objects. Timing diagrams depict time durations of operations associated with segments of the system. Threads of control track the information provided in the dynamic behavior models along a particular path which is representative of a system scenario. In that threads of control integrate the puzzle pieces which make up the system, their representation is fundamental to understanding how a system operates as a whole.

THREADS OF CONTROL

A *thread of control* is a path which traces a sequence of operations among or within objects or classes. This path represents a scenario which may be used during analysis, design, or testing to trace through the model. Threads of control are valuable for analyzing the model for completeness to ensure that all aspects of the system being modeled are represented. Additionally, for real-time systems, they are essential in identifying real-time processing requirements for timing constraints and bottlenecks. Threads of control represent the integration, along a particular path, of the state, control flow, data flow, and timing data contained in the dynamic behavior models. State information is needed because the thread of control may vary depending on the state of the class/object. Timing, data flow and control flow data provide the sequencing information, data required along a particular sequence, and associated duration.

Coad and Yourdon presented thread of control information in a cursory fashion via message connections on their Class-&-Object diagram; although descriptive information about the threads of control as related to a particular class/object may be contained within that class/objects specification. In all of the methodologies reviewed, only Shlaer and Mellor had a clear representation of the relation between the states of a class/object and the threads of control associated with the states using a thread of control chart; however, the tie back to the associated class/object was not apparent and none of the data associated with the flow was represented. The Timing Diagram was the only mechanism available in Booch's methodology which reflected thread of control related information. While it tied operations, sequencing, and times to objects, it was deficient in representing data and state information. Firesmith provided three different diagrams containing various thread of control information: an object-oriented control flow diagram for each major thread of control, a thread-level interaction diagram to show the interactions of classes/objects for a given scenario, and a timing diagram for each thread of control. Rumbaugh used event trace diagrams to show the sequencing of events in a system; however the information provided in this diagram depicted only the event sequencing and the class/objects impacted by the event. Thread of control representations associated with the methodologies reviewed are summarized in Figure 3.

The review of methodologies for thread of control information indicated that none of the methodologies covered all of the information associated with threads of control. Firesmith's method appeared to provide the best and most comprehensive thread of control information of the methodologies reviewed, but the information is spread over several diagrams and is therefore difficult to assimilate.

	Coad and Yourdon	Shlaer and Mellor	Booch	Firesmith	Rumbaugh
Thread of Control (TOC)	**Message Connections on Class-&-Object Diagram. TOC in Class-&-Object Specification in bullet list format or in Service Chart.**	**Thread of Control Chart shows events and states occurring in a thread and associated times.**	**Timing Diagram shows objects and operations sequence and duration.**	**TOC Object-Oriented CFD, Thread-level Interaction Diagrams, TOC Timing Diagram**	**Event traces - shows event sequencing and the associated class/objects**
State Information	**no**	**yes**	**no**	**no**	**no**
Control Flow Information	**yes**	**yes**	**yes**	**yes**	**yes**
Data Flow Information	**no**	**no**	**no**	**yes**	**yes**
Timing Information	**yes**	**yes**	**yes**	**yes**	**no**
Associated Class/Object	**yes**	**no**	**yes**	**yes**	**yes**

Figure 3. Thread of Control Representations

Because of the significant role that threads of control play in understanding the overall operation of a system, an effective means of representing them in object-oriented analysis and design is needed. Current methodologies tend to fragment this information showing only segments of the system at a time, using multiple models for different views of these segments. The approach to integrating threads of control in object-oriented analysis and design described here begins by abstracting classes/objects at a higher level. The rationale for this higher level of abstraction is two-fold: first, end-to-end processing is easier to show, as well as understand, at a higher level of abstraction; second, thread of control information is attached at the higher level of abstraction which lessens the amount and complexity of the information to be handled. Abstracting classes/objects at a higher level implies that classes/objects are aggregated into larger groups based on some criteria, such as being logically related to each other. These logically related groups make up a logical view of the system.

LOGICAL COMPOSITE CLASSES

A *logical view* represents the groupings of classes/objects which are logically related into higher levels of composition. Partitioning into groups is usually based on engineering judgment, and minimizing the associations, aggregations and generalizations between groups. Traditionally, the rationale for grouping classes/objects is for partitioning large projects, and to provide a means of understanding the overall system and its interfaces. A logical view which overlays a class/object diagram might look something like what is shown in Figure 4.

Logical View

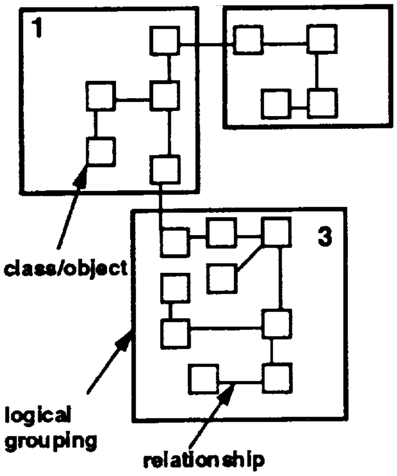


Figure 4. Groupings for a Logical View

Although no formal method for grouping classes/objects was presented in the methodologies reviewed, each methodology touched on the concept in some fashion. Yet, the terminology and approaches associated with the groupings of classes/objects varies among the different methodologies. Coad and Yourdon use the term Subject. Subjects are initially created by identifying the uppermost class, the parent, in each is-a or has-a structure and calling it a subject. Subjects are further refined by minimizing the relationships and message connections between the subjects. Shlaer and Mellor use a top-down approach to grouping. They begin by identifying the different domains which make up a system and partition large domains into subsystems. It is for each of these subsystems that the class/object diagrams are constructed. Booch combines classes/objects into modules which are actually physical representations in that they are intended to represent software modules. Modules may be further logically grouped into subsystems. Firesmith groups classes/objects into subassemblies which ultimately make up an assembly. He describes several approaches to identifying subassemblies depending on the situation involved. Some of the approaches involve bottom-up development where class/objects are identified and then grouped based on the criteria dictated by the approach, such as coupling and cohesion criteria. Other approaches, such as recursion, begin with top-down development by identifying parent subassemblies, and recursively defining other subassemblies as needed. Rumbaugh introduces modules which are logical groupings of the class/objects and associated relations defined in his object model. Modules are the lowest level subsystems. The terminology, models and representations associated with each of the methodologies for a logical view are shown in Figure 5.

	Coad and Yourdon	Shlaer and Mellor	Booch	Firesmith	Rumbaugh
Terminology	Subjects	• Domain • Subsystem	• Subsystem • Module	• Assembly • Subassembly	• Subsystem • Module
Model	Subject Layer	** • Domain Chart • Subsystem Relationship Model • Subsystem Communi-cation Model • Subsystem Access Model	Module Architecture	Assembly Model	N/A
Represent-ation	Class-&-Objects Diagram		• Subsystem Diagram • Module Diagram	• Context Diagram • Assembly Diagram	Object Diagram (modules only)

** No distinction between model and representation

Figure 5. Logical View Representations

While these methodologies all describe logical groupings, they do not use this construct in conjunction with thread of control information to represent end-to-end processing. It is in this context that the logical composite class construct is introduced. A *logical composite class* is a grouping of classes/objects which are logically related and further refined/extended by integrating thread of control information. The rationale for logical composite classes is that they provide a mechanism for representing end-to-end threads of control through class/object groupings combining both static architecture and dynamic behavior. They are also a precursor to process composite classes which further refine groupings using process partitioning criteria. Process composite classes are detailed in a later section. These constructs should be viewed as design-level classes which can be integrated into a design language. Instances of this class are the actual groupings and their associated data.

The methodology used to generate logical composite classes is a bottom-up approach which begins with the initial groupings formed from the logical view. Next, the pertinent state, data and control flow information required for threads of control is aggregated for each logical grouping. As previously described in the dynamic behavior models, this information is already available in fragmented form at the class/object level. To aggregate the information means to recompose the information at the class/object level to the level of abstraction of the logical groupings in a summarized form. This aggregated information is assessed at the boundaries of the logical groupings by focusing on the information required between the *boundary* classes/objects. The boundary class/objects are those class/objects in the logical view that play an interface role between the groups defined in the logical view. The aggregated information is attached to the associated logical grouping. Groupings are then refined to minimize connections among groups. Figure 6 shows the logical composite classes which evolved from the logical view groupings of Figure 4.

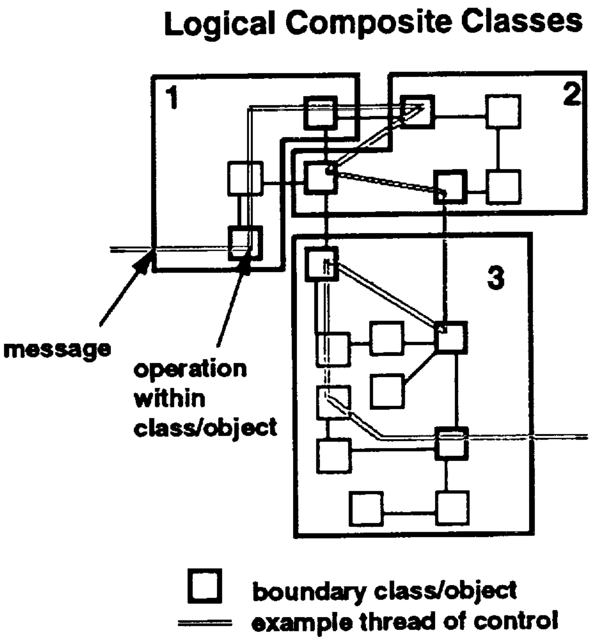

Logical Composite Classes
1
2
3
message
operation within class/object
□ boundary class/object
═ example thread of control

Figure 6. Logical Composite Class Representation

The philosophy of using composite classes as aggregations of class/objects is probably most closely associated with how Coad and Yourdon identify and refine subjects, since subjects evolve partially out of has-a relationships. Refining the groupings as development continues is consistent with Firesmith's recursive approach to development. However, the logical composite class extends these concepts further by introducing a design-level construct which contains a grouping of classes/objects at a higher level of abstraction and attaches aggregated data representing thread of control information to those constructs.

While all of the methodologies described logical groupings, none of them addressed the refinement of these groupings for processes as is required in real-time and distributed systems. In this paper, the concept of grouping is extended even further using the process composite class construct as a mechanism for refining groupings along process lines.

PROCESS COMPOSITE CLASS

A *process view* represents the mapping of class/objects to *processes*. In this context, processes are entities implemented in software that may execute concurrently and compete for resources. The introduction of multiple threads of control necessitates partitioning systems into processes. The logical composite classes shown earlier might contain multiple threads of control as shown in Figure 7.

Logical Composite Classes

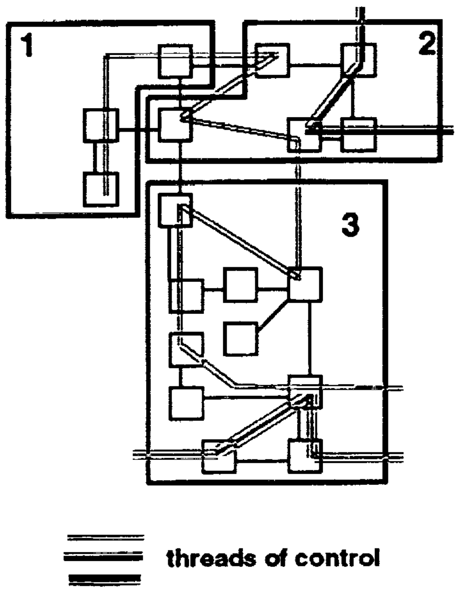


Figure 7. Logical Composite Classes with Multiple Threads of Control

Of the methodologies reviewed, the only methodology that provided a model for processes was the Process Architecture model presented in Booch. This model described templates for processes and processors. These templates contained information concerning the characteristics of the computer, processes associated with each processor, priority for each process and the scheduling approach. However, the model lacked any transition or correlation to Booch's previously described class structure, object structure or module architecture models. Additionally, no criteria for how processes should be allocated or identified was provided.

The need for process partitioning has long been recognized in the real-time development community. The merging of this technology with object-orientation is still in its infancy. The key criteria for process partitioning have to do with communication and timing. In terms of communication, the ideal is to minimize communication between processes by grouping classes/objects which interface extensively within a process, thereby reducing the interaction between groups. The interface between groups is referred to as *coupling*, and within a group, *cohesion*. An excellent discussion on the coupling and cohesion of objects and modules is presented in [Berard].

Timing criteria affect process partitioning in a number of ways. For example, those classes/objects whose operations support services which must be performed within a specified time should be grouped in an independent process. Classes/objects whose operations support services which perform on different cycles, sporadically, or at a low level of priority should be separated into different processes. As previously mentioned, threads of control may be used to trace through critical paths in a system to determine total execution criteria. While a determination may be made to add processes due to timing constraints, the tradeoff between adding these processes versus the overhead to run them must be weighed. Additionally, the more processes that are added, the more complex the system becomes. A representative listing of partitioning criteria for processes is provided in [Neilsen].

The construct introduced in this paper to represent the partitioning of systems into processes, is the process composite class. A *process composite class* is a grouping of classes/objects originating from the logical composite class groupings and further refined based on process partitioning criteria. The logical composite classes already represent an initial partitioning based on the existence of interactions between groups. The methodology for developing process composite classes begins by extending these logical composite classes with timing information. The timing information associated with each logical composite class is assessed. Class/objects or class/object groupings which have distinguishing timing criteria such as being time critical or the other extreme, low priority, are extracted from within the logical composite classes. Weights may then be assigned to interfaces between modified groupings as a function of the number of data/control flows among the groupings. These weights determine the need for further repartitioning based on changed interactions between groups resulting from the previous repartitioning based on timing. Weights reflect the magnitude of communications between the groups. Repartitioning is performed as needed to achieve total execution time criteria. Figure 8 highlights how these sequences of repartitionings might look. Beginning with the grouping of the logical composite classes from Figure 7, Figure 8 shows subsequent groupings into process composite classes based on various process partitioning criteria. Keeping track of these numerous classes/objects, the interrelationships among them, the threads of control through them, and the partitioning criteria needed to determine the potential groupings into composite structures, quickly becomes a complex problem which is well suited for a database environment.

The formulation of groupings into process composite classes involves taking the thread of control information attached to the logical composite class, and applying process partitioning criteria with system constraints to result in process composite classes. Classes and their associated attributes, operations and state data are contained in a database. The relationships that tie operations to particular state values or changes in attribute values are also maintained. In the context of a logical composite class, thread of control information is extracted from the appropriate classes. That is, the class/objects whose operations are invoked along that thread of control, and the attributes and data impacted or used in conjunction with those operations, are linked to the thread of control. Additional information associated with the particular thread of control such as operation precedence, identification of time critical operations (priorities and deadlines), priority and timing constraints, and communication interface requirements is also included.

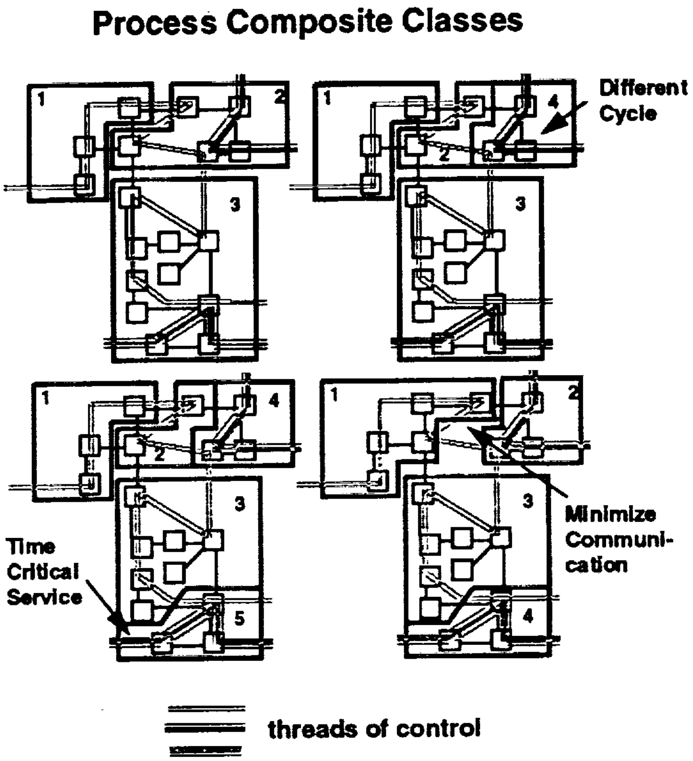


Figure 8. Repartitionings of Process Composite Classes

After threads of control are enumerated, interrelationships may be identified and assessed. For example, different threads of control may use different operations within a class. Interrelationships may be involved if one thread of control alters attribute values by invoking a particular operation in a class where these attribute values are also used by another operation invoked by a separate thread of control. The intra-dependencies of attributes affected by operations within a given class is maintained in the database. These intra-dependencies must be considered among the various threads of control. The interdependencies along various threads of control between logical composite class groupings must also be considered. These dependencies and their magnitude provide much of the data needed to make process partitioning decisions.

The process composite class definition can be augmented by algorithms which provide optimal solutions to allocations. Given the proper criteria, these algorithms can provide solutions using various methods such as graph-theoretic allocation or a heuristic branch and bound allocation that minimizes or maximizes performance objectives

[Horowitz], [Reeves]. Typical constraints minimize the cost of running the total system by partitioning the process composite classes efficiently. The partitioning resulting from the process partitioning criteria, combined with system constraints such as communication bandwidth, processor speed or concurrency limitations, provide the information needed to define the performance objectives. Figure 9 depicts the overall formulation of groupings into process composite classes.

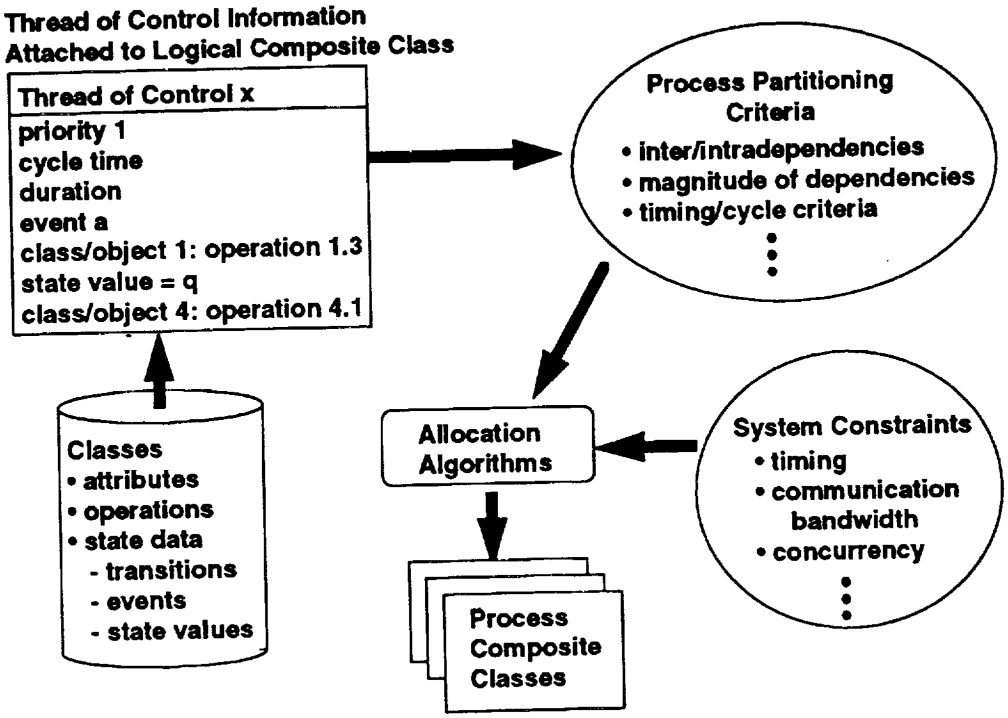

Thread of Control Information
Attached to Logical Composite Class
Thread of Control x
priority 1
cycle time
duration
event a
class/object 1: operation 1.3
state value = q
class/object 4: operation 4.1
Process Partitioning Criteria
• inter/intradependencies
• magnitude of dependencies
• timing/cycle criteria
Classes
• attributes
• operations
• state data
- transitions
- events
- state values
Allocation Algorithms
System Constraints
• timing
• communication bandwidth
• concurrency
Process Composite Classes

Figure 9. Formulating Process Composite Classes

The results of this work are being used to develop a streamlined methodology for use with distributed, real-time applications. Basic class/object static architectures and dynamic behaviors will be drawn from the strengths of the methodologies reviewed and consolidated. The logical and process composite class structures will provide a layer above these other constructs and will be integrated in a design language. Integrating this concept into a design language provides a means of representing the structure graphically, building a database, and generating consistency checks. Additionally, it provides a basis for an automated mechanism so that regrouping for logical and composite class structures, and the application of algorithms to these structures, may be easily accomplished for efficiency comparison purposes.

FUTURE WORK

Several issues have arisen as a result of this research which require further investigation. These issues focus on specific cases where object-oriented and distributed system partitionings are in conflict. One case concerns the fact that distributed systems sometimes require that parts of the same object be in multiple locations. For example, different operations may be required on the same object depending on where it is located in the system. This requirement is contrary to all of the attributes and operations associated with an object being encapsulated within the object. Another case is one in which the various operations contained in an object may have different timing constraints. For example, one operation may be along a time critical thread of control while another may not. The first inclination would be to group the object into a process in accordance with the highest priority operation. The down side of this, however, is that all of the other information related with that object, such as the secondary operations, and threads of control and objects associated with those secondary operations, are then grouped into the same time critical process. These cases and others like them require further exploration in order to integrate solutions into the process partitioning approach.

SUMMARY

The results of this research indicate that current object-oriented analysis and design methodologies' representations do not provide a clear understanding of the end-to-end processing which defines system operation. This research has introduced logical and process composite classes that act as structures for representing groupings of class/objects. These structures reflect classes/objects and the threads of control through those classes/objects. Further study is needed to extend these structures into a design language, and refine the partitioning conflicts which arise between objects and processes.

REFERENCES

[Berard] Berard, Edward V., *Essays on Object-Oriented Software Engineering, Volume I*, Prentice Hall, Englewood Cliffs, NJ, 1993.

[Booch] Booch, Grady, *Object-Oriented Design with Applications*, Benjamin/Cummings Publishing, Redwood City, CA, 1991.

[Coad91a] Coad, Peter and Yourdon, Edward, *Object-Oriented Analysis, Second Edition*, Yourdon Press, Englewood Cliffs, NJ, 1991.

[Coad91b] Coad, Peter and Yourdon, Edward, *Object-Oriented Design*, Yourdon Press, Englewood Cliffs, NJ, 1991.

[Fichman] Fichman, Robert G., and Kemerer, Chris, F., "Object-Oriented Analysis and Design Methodologies Comparison and Critique," *Computer*, Vol. 25, No. 10, October 1992, pp. 22-39.

[Firesmith] Firesmith, Donald G., *Object-Oriented Requirements Analysis and Logical Design - A Software Engineering Approach*, John Wiley & Sons, New York, NY, 1993.

[Horowitz] Horowitz, E., and Sahni, S., *Fundamentals of Computer Algorithms*, Computer Science Press, Inc., Rockville, Maryland, 1978.

[Korson] Korson, T., and McGregor, J. D., "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, Vol. 33, No. 9, September 1990, pp. 40-60.

[Nielsen] Nielsen, Kjell, *Object-Oriented Design with Ada*, Bantam Books, New York, New York, 1992.

[Reeves] Reeves, Colin R., *Modern Heuristic Techniques for Combinatorial Problems*, John Wiley & Sons, Inc., New York, NY, 1993.

[Rumbaugh] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Shlaer88] Shlaer, Sally, and Mellor, Stephen J., *Object-Oriented Systems Analysis - Modeling the World in Data*, Yourdon Press, Englewood Cliffs, NJ, 1988.

[Shlaer91] Shlaer, Sally, and Mellor, Stephen, J., *Object Lifecycles - Modeling the World in States*, Yourdon Press, Englewood Cliffs, NJ, 1992.

Integrating End-to-End Threads of Control into Object-Oriented Analysis and Design

Eighteenth Annual Software Engineering Workshop
NASA/Goddard Space Flight Center
December 1-2, 1993

Janet E. McCandlish
TRW System Development Division
Huntsville Operations

Dr. James R. MacDonald
ALPHATECH, Inc.

Dr. Sara J. Graves
University of Alabama
in Huntsville

Problems

- Current object-oriented analysis and design methodologies fall short in their representation of end-to-end processing
  - system is represented with multiple views
  - only pieces of the system are represented
  - people have difficulty in seeing how system operates
- Goals associated with object-oriented and distributed systems are conflicting
  - large-scale real-time distributed systems reconcile competing demands for resources by partitioning the system into multiple processes
  - object-oriented technology strives to partition a system by objects where all data and operations associated with an object are encapsulated within the object
  - the partitionings for processes and objects appear to be orthogonal in this context when threads of control are considered

Solution/Approach Overview

- Represent threads of control and their associated class/objects to better illustrate how the system operates
  - Five current object-oriented analysis and design methodologies assessed: Coad and Yourdon, Shlaer and Mellor, Booch, Firesmith, and Rumbaugh
  - Introduce a representation which overlays dynamic flow (threads of control) onto a static structure
- Group class/objects at higher level of abstraction for process partitioning
  - Combining dynamic and static representation to show end-to-end processes requires some grouping of classes/objects at higher levels
  - To simplify partitioning problem, grouping is two-phased: (1) logical groupings, further refined with thread of control information (provides a coarse-grained partitioning) (2) process groupings, extend logical groupings with process partitioning criteria

Background

- Static Architecture
  - non-temporal representation of the system
  - typically depicted as an enhanced entity-relationship diagram
- Dynamic Behavior
  - behavior attributable to timing and flow of information
  - may include state, control flow, data flow and timing information
- Thread of Control
  - path which traces a sequence of operations among or within objects or classes
  - represents a scenario which may be used during analysis, design, or testing to trace through the model for completeness and real-time processing requirements
- Static and dynamic representations exist, but thread of control representation is limited

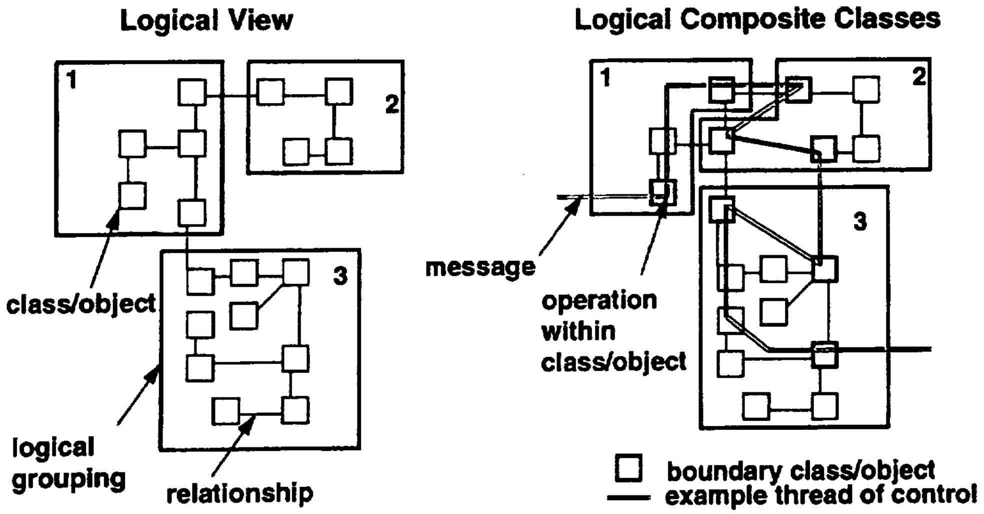
Thread of Control Representations

	Coad and Yourdon	Shlaer and Mellor	Booch	Firesmith	Rumbaugh
Thread of Control (TOC)	Message Connections on Class-&-Object Diagram. TOC in Class-&-Object Specification in bullet list format or in Service Chart.	Thread of Control Chart shows events and states occurring in a thread and associated times.	Timing Diagram shows objects and operations sequence and duration.	TOC Object-Oriented CFD, Thread-level Interaction Diagrams, TOC Timing Diagram	Event traces - shows event sequencing and the associated class/objects
State Information	no	yes	no	no	no
Control Flow Information	yes	yes	yes	yes	yes
Data Flow Information	no	no	no	yes	yes
Timing Information	yes	yes	yes	yes	no
Associated Class/Object	yes	no	yes	yes	yes

Logical View

- The *Logical View* represents groupings of classes/objects which are logically related. Partitioning into groups is based on:
  - engineering judgement
  - minimizing the associations, aggregations and generalizations between groups
- Rationale for Logical Groupings
  - Partitioning for large projects
  - Means of understanding overall system and interfaces
- Terminology
  - Subjects - *Coad and Yourdon*
  - Domains/Subsystems - *Shlaer and Mellor*
  - Subsystems/Modules - *Booch, Rumbaugh*
  - Assemblies/Subassemblies - *Firesmith*

Logical Composite Class Representation



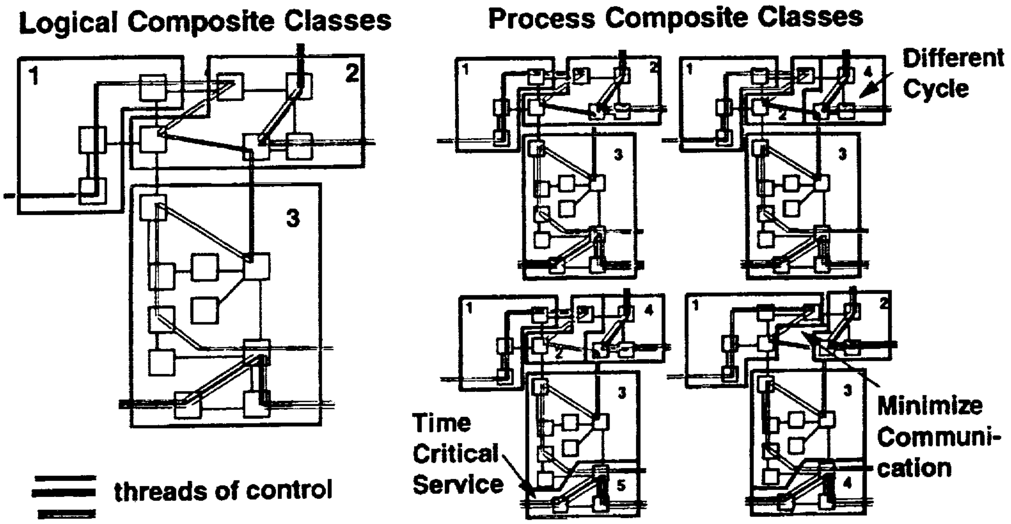
Process View

- The *Process View* represents the mapping of class/objects to *processes* (entities implemented in software that may execute concurrently and compete for resources).
- The introduction of multiple threads of control is a major reason for partitioning the systems into processes.
- The only methodology reviewed that provides a model for processes is Booch's Process Architecture.
  - Describes templates for processes and processors
  - Provides no transition from his other models (class structure, object structure or module architecture) to the process architecture

Process Composite Class

- A *process composite class* is a grouping of classes/objects originating from the logical composite class groupings and further refined based on process partitioning criteria
- Process Partitioning Criteria
  - Communication
    - » minimize communication between processes
  - Timing
    - » class/objects whose operations support services which must be performed within a specified time
    - » class/objects whose operations support services which perform on different cycles, sporadically, or at a low level
    - » adjust process groupings as needed to meet total execution time criteria

Process Composite Class Representation



Logical Composite Classes

Process Composite Classes

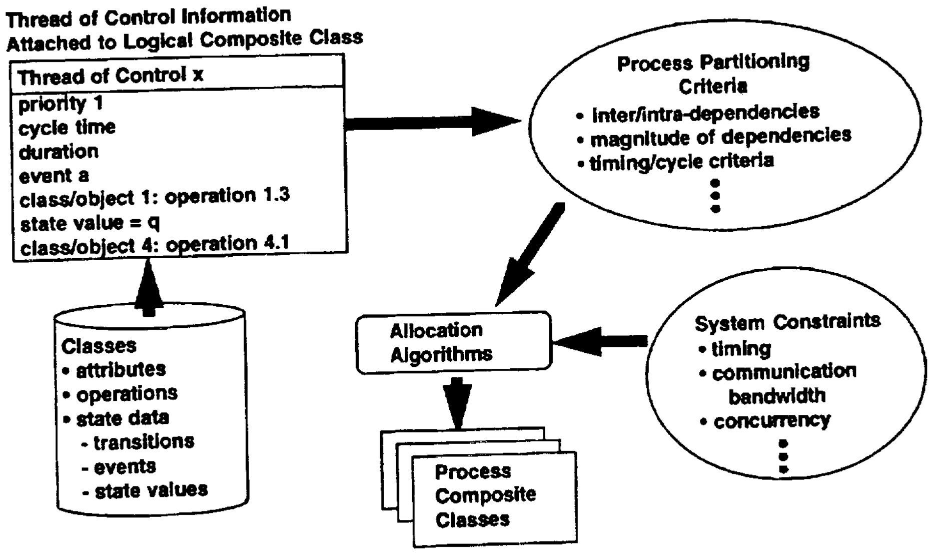
Different Cycle

Time Critical Service

Minimize Communication

threads of control

Formulating Process Composite Classes



Thread of Control Information
Attached to Logical Composite Class

Thread of Control x
priority 1
cycle time
duration
event a
class/object 1: operation 1.3
state value = q
class/object 4: operation 4.1

Classes
- attributes
- operations
- state data
  - transitions
  - events
  - state values

Process Partitioning Criteria
- inter/intra-dependencies
- magnitude of dependencies
- timing/cycle criteria

⋮

Allocation Algorithms

System Constraints
- timing
- communication bandwidth
- concurrency

⋮

Process Composite Classes

11

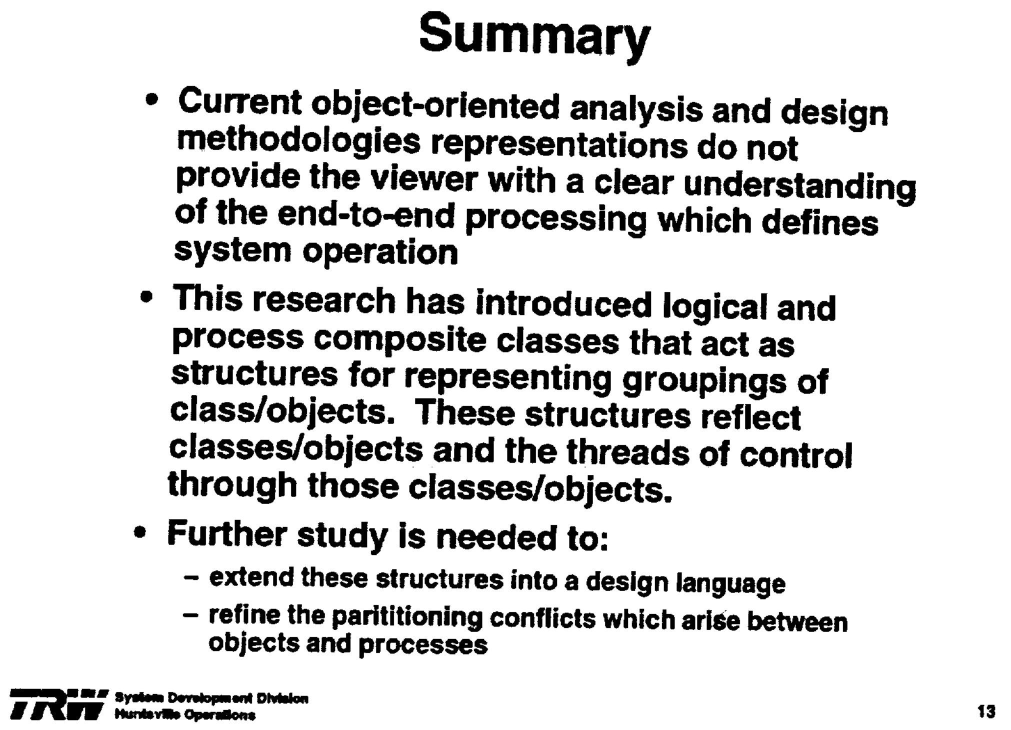
TRW System Development Division
Huntsville Operations

Ongoing Work

- Represent the logical and process composite class structure in a design language
  - provides a means of representing the structure graphically, building a database, and generating consistency checks
  - basis for automated mechanism so that regrouping for logical and composite class structures may be easily accomplished for efficiency comparison purposes
- Address specific cases where object-oriented and distributed system partitionings are in conflict and integrate solutions into process partitioning approach. Examples include:
  - distributed systems sometimes require that parts of the same object be in multiple locations
  - one operation in an object may be along a time critical thread of control while another may not; requires that entire object and all associated threads and objects be grouped into time-critical process

12

TRW System Development Division
Huntsville Operations



Summary

- Current object-oriented analysis and design methodologies representations do not provide the viewer with a clear understanding of the end-to-end processing which defines system operation
- This research has introduced logical and process composite classes that act as structures for representing groupings of class/objects. These structures reflect classes/objects and the threads of control through those classes/objects.
- Further study is needed to:
  - extend these structures into a design language
  - refine the parititioning conflicts which arise between objects and processes

TRW System Development Division
Huntsville Operations

13

Session 5: Process

omit

Susan Main Hall, SofTech, Inc.

Jeffrey M. Voas, Reliable Software Technologies Corp.

Regina Palmer, Martin Marietta Astronautics

Fusing Modeling Techniques to Support Domain Analysis for Reuse Opportunities Identification

Susan Main Hall
Eileen McGuire
SofTech, Inc.
1600 N. Beauregard St.
Alexandria, Virginia 22311
(703)824-4561 FAX: (703)931-6530
email: shall@softech.com

Functional modeling techniques or object-oriented graphical representations, which are more useful to someone trying to understand the general design or high level requirements of a system?

For a recent domain analysis effort, the answer was a fusion of popular modeling techniques of both types. By using both functional and object-oriented techniques, the analysts involved were able to lean on their experience in function oriented software development, while taking advantage of the descriptive power available in object oriented models. In addition, a base of familiar modeling methods permitted the group of mostly new domain analysts to learn the details of the domain analysis process while producing a quality product.

This paper describes the background of this project and then provides a high level definition of domain analysis. The majority of this paper focuses on the modeling method developed and utilized during this analysis effort.

Project Background

The analysis work described in this paper was performed in support of the Software Development Center - Washington, Army Reuse Center (ARC). Using functional descriptions and design documentation of four Army software systems under development and the Department of Defense Technical Reference Model, the application support layer services, such as database services, network communications, and the human machine interface, were studied. In addition, technical references were used to support the development of the description for the User-Machine Interface (UMI). The primary goal of the effort was to develop a complete, understandable model of a generic application support layer system. When completed, this model was utilized to identify potential reuse opportunities between the existing software and future system development efforts. The majority of the work performed by the ARC and its supporting staff focuses on increasing software reuse in the government sector.

Domain Analysis

Domain Analysis is the process of identifying the commonalities in a class of similar systems [Priento-Diaz 90]. Domain analysis could be considered as requirements analysis performed on more than one system. The activities performed during domain analysis include collecting, organizing, analyzing and concisely capturing information from systems which perform similar tasks. System specifications, requirements documents, functional descriptions, design documents, and even users manuals can provide the information needed for domain analysis. The key to successful domain analysis is to have complete descriptions for at least three systems in the software family being studied. At least three systems are needed in domain analysis in order to obtain a non-system-specific view of the domain.

There are two ways to view a family of systems or domain: vertically or horizontally. A vertical domain encompasses systems which perform the same system application. For example, in Figure 1, Embedded Weapons Systems, Management Information

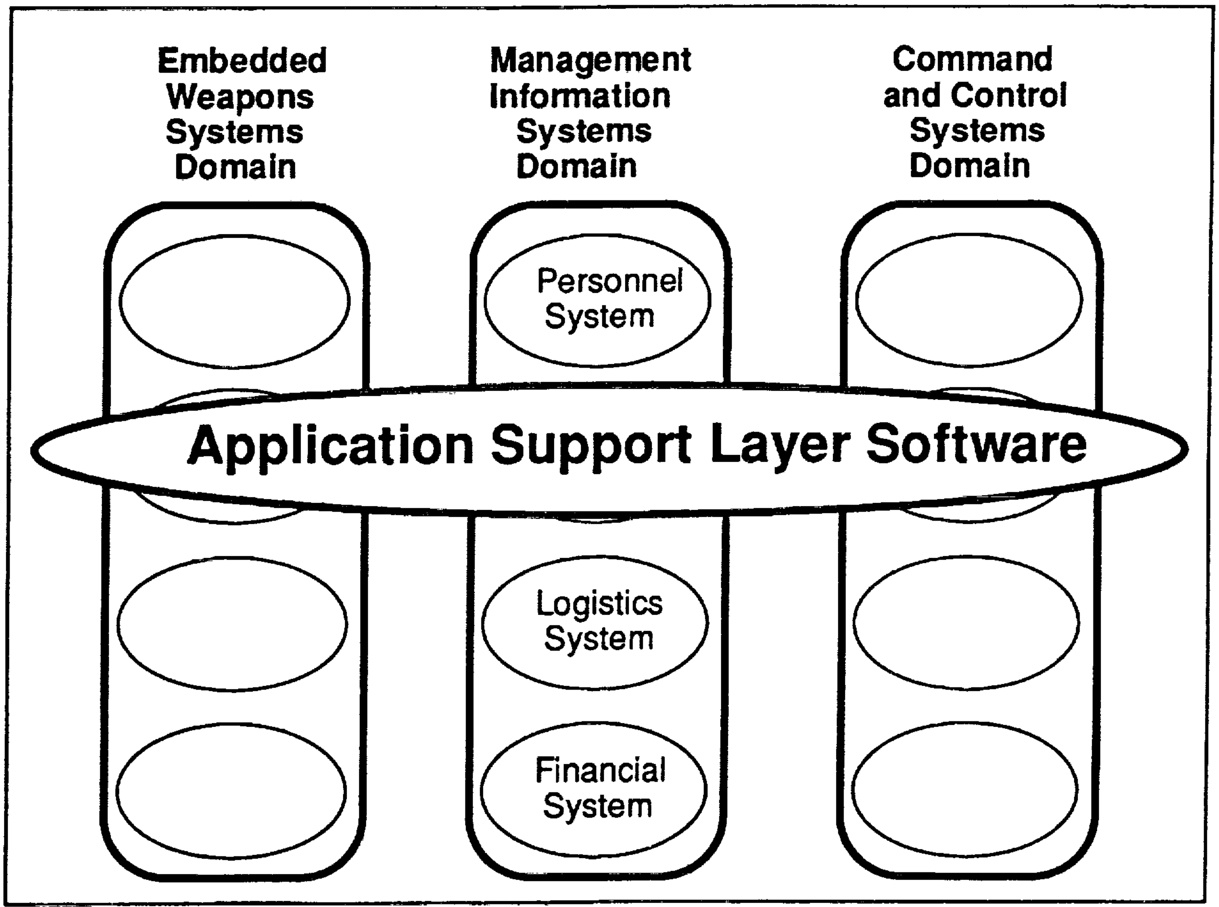

Embedded Weapons Systems Domain
Management Information Systems Domain
Command and Control Systems Domain
Personnel System
Application Support Layer Software
Logistics System
Financial System

Figure 1 *The Application Support Layer Software is a horizontal sub-domain of many software domains.*

Systems and Command and Control Systems represent three high-level vertical domains. A horizontal domain is an area of activity or knowledge that spans the vertical application oriented domains. The Application Support Layer Software domain is used as the example domain in this paper to describe the modeling technique/procedure that has been performed by this analysis team.

Unlike other domain analysis methods, the analysis procedure this SofTech team employs incorporates the concept of domain-oriented high demand categories and a knowledge base of domain and system information to support the identification of areas predisposed to reuse. The identification of reuse opportunities is a priority in our domain analysis work. The analysis procedure utilized is also unique because of the effective method of combining several modeling techniques that was developed (see the section below for a detailed description of this fused modeling technique). Note, although a thorough study of the application support layer services was performed, the pictorial description captured was limited to include only a high level illustration of the domain.

Combining Multiple Modeling Techniques

Domain modeling is the process of capturing, in graphical form, the conclusions resulting from analysis of a functional family of systems. Specifically, the operations, data and data attributes need to be recorded in a clear, concise format. In general, since object-oriented system descriptions make reuse opportunities easier to locate [Weesale 92], an object-oriented model was targeted to be the final product of the analysis effort. Unfortunately, one of the greatest challenges our team faced was striving to bridge the gap between systems that are still being functionally designed and the advantages that object oriented technologies offer. Therefore, we also came to the conclusion that a fusion of modeling methods was necessary due to the relative immaturity of the available techniques [Weesale 92].

When comparing and contrasting the understandability of modeling techniques, we have found that one modeling technique could not do the entire job well. Since the development of an understandable generic Application Support Layer (ASL) model is critical to its future use we decided to combine several very different modeling techniques. Our new modeling process includes utilizing functionally oriented models, moving into a functional hierarchical grouping model, and then transitioning into a set of object oriented models. Specifically, we used data flow diagrams [DeMarco 78], state transition diagrams, flow charts, hierarchical diagrams, and object models [Coad/Yourdon 91, Rumbaugh 91].

Studying the ASL software began by creating sketches of and reviewing pre-existing data flow diagrams, state transition diagrams, and flow charts from documentation available on the completed Army systems. These functionally oriented diagrams were beneficial to our understanding the systems because of the analysis team's experience in developing functionally oriented software. In addition, capture of the functionality of an ASL in these types of diagrams was performed quickly, since the example systems being studied had been developed using functionally-oriented methods. Both the analysts' experience and the system development techniques supported the easy understanding of the processes performed by typical ASL software.

High-level data flow diagrams provided the basis for the majority of the analysis work on the processes of the ASL software. For example, Figure 2 is a data flow diagram (DFD) of the primary functions performed by the ASL software, according to the functional descriptions of one of the systems studied. Also, shown in the same DFD are the general data flows between the functions. This type of diagram provided an understanding of the basic activities performed by an actual ASL code module.

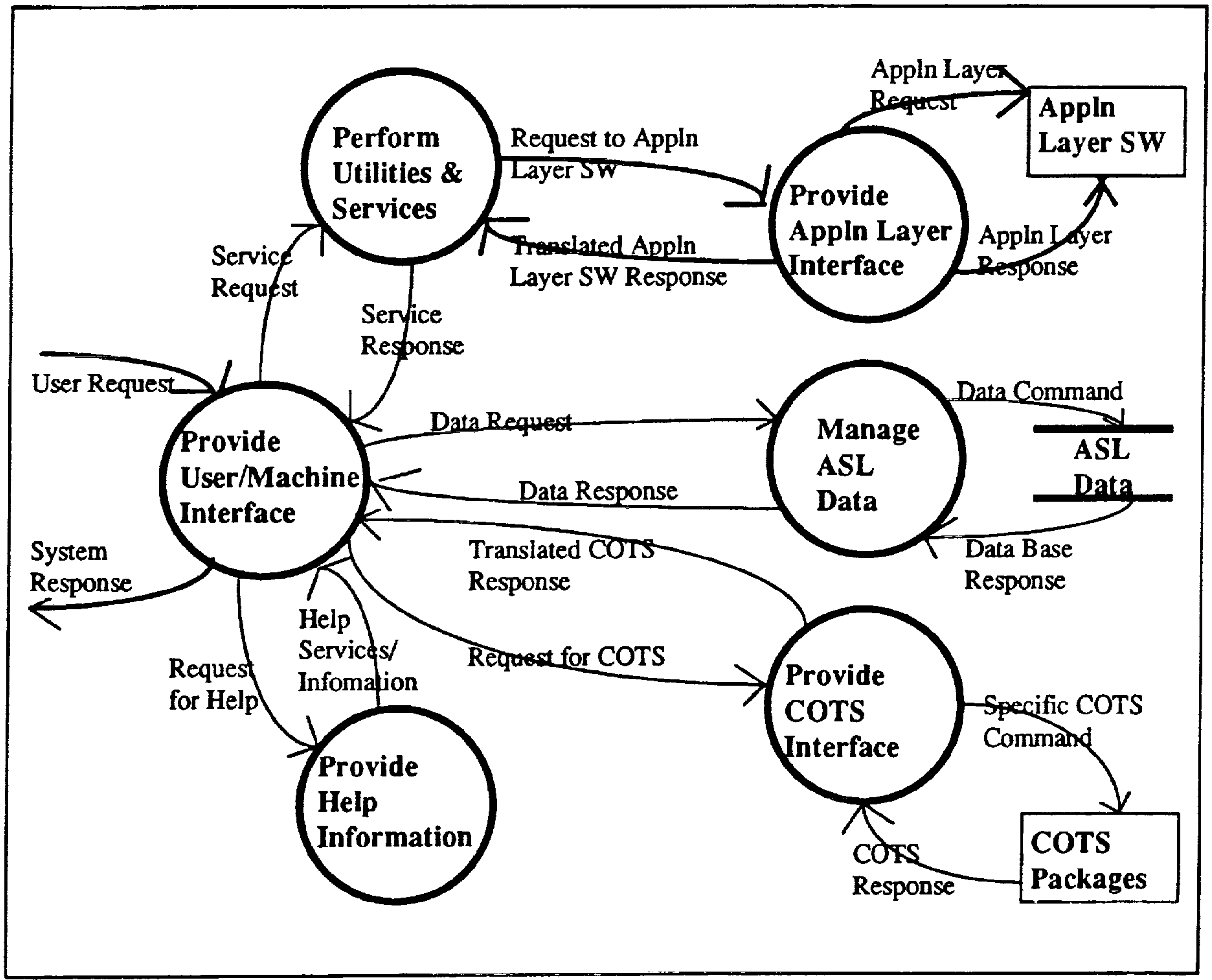


Figure 2 High-level data flow diagram of the ASL functionality of one of the systems studied.

Further breakdown of the processes helped to define the specific functions performed, the role of these functions, and the existence of hardware and system dependencies in the ASL software. For instance, the process in Figure 2 called Perform ASL Utilities and Services includes sub-functions such as: Manage Errors, Perform Execution Management, Manage Report Requests, Perform Platform Services, and Handle Interprocess Communication. Note, the last two sub-functions in the previous list are examples of hardware and software dependent activities. Though complete DFDs were not created for each of the processes described by each of the ASL systems studied, select functions were analyzed in greater detail to clarify the data and specific operations involved.

As with many analysis efforts, the most familiar functions proved to be the most difficult to accurately model. State transition diagrams and flow charts were used occasionally to focus the analysis team on actual processing activities and data manipulation details, instead of letting the team rely on sweeping assumptions. In some cases, functions were reviewed at a level of detail much finer than would be captured in the final object-oriented model in order to avoid missing important functionality.

Figure 3 shows an example flow chart of part of the analysis team's discussion on how the ASL software provides the interface between the system user and the

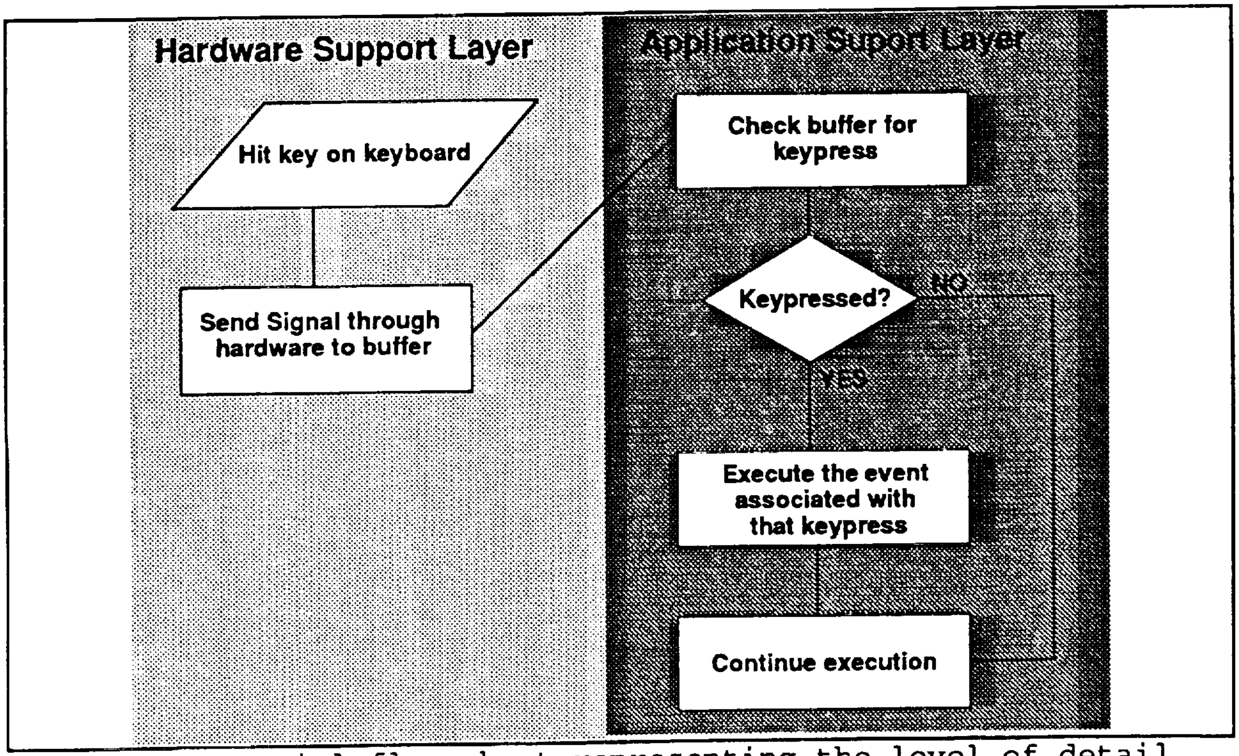

Hardware Support Layer
Application Suport Layer
Hit key on keyboard
Send Signal through hardware to buffer
Check buffer for keypress
Keypressed?
NO
YES
Execute the event associated with that keypress
Continue execution

Figure 3 A partial flow chart representing the level of detail discussed for some of the components of the Application Support Layer domain.

machine. Though details on keyboard use do not define software, they did provide some insight as to the specific software objects involved such as text, lines and shapes. The group reviewed the physical activities (i.e. pushing a function key) to pinpoint the associated software (i.e. the ASL commands that perform the specific data manipulation). By exploring the operations performed by the ASL software, the data objects in this hidden layer of software were identified which assisted in providing a more complete picture for the final object-oriented models.

Moving from a functional model to a object model can result in losing important information. Therefore, in an effort to minimize the impact, a third technique called a functional hierarchical grouping model was applied. This home-grown technique is the fusion between a functional model and an object model. The technique consists of putting the identified functions of the system in a hierarchical model and then grouping the lowest level functions together based on the objects being manipulated. For example, in Figure 4 below, all functions involving the human interaction with the computer system were grouped together to form the basis of an object oriented user machine model.

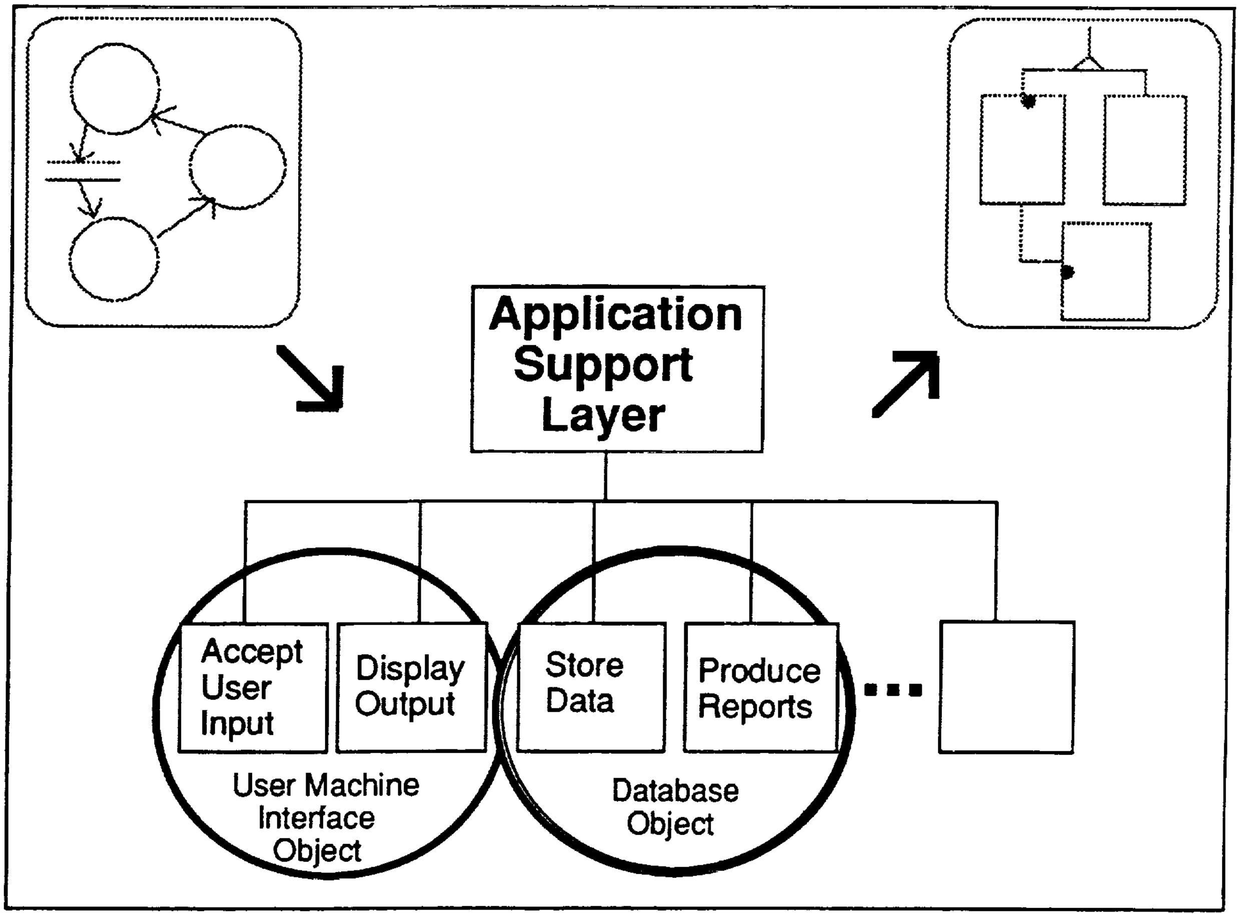


FIGURE 4 *Starting with functional modeling techniques and moving toward object oriented techniques, a fusion of methods occurred.*

6

The first objective in producing the hierarchical model was to list all of the functions potentially performed by an ASL. The word *potential* is used because the interfaces to the ASL also, needed to be defined. Therefore, in this case, too much high-level information is actually helpful. As the top-level processes were broken down into less complex sub-processes, the specific functionality of an ASL became apparent. Activities performed by the application layer or the hardware support layer were removed from the hierarchical model. For instance, one of the analysis team's first hierarchical models included all of the components shown in Figure 5 below, but through a series of iterations several of the components were determined not to be a required part by the typical ASL. Some of the components were hardware support layer activities, like the network functions and some of the components were found to be embedded in other components. The Help functions are an example of this; that is, most of the time, software modules contain their own help files, since Help is so application dependent. In addition, some components were raised in importance based on further analysis. For instance, the Kernel Support sub-function Platform Abstraction in the hierarchical model shown in Figure 5 became a primary area of focus in the final object-oriented model.

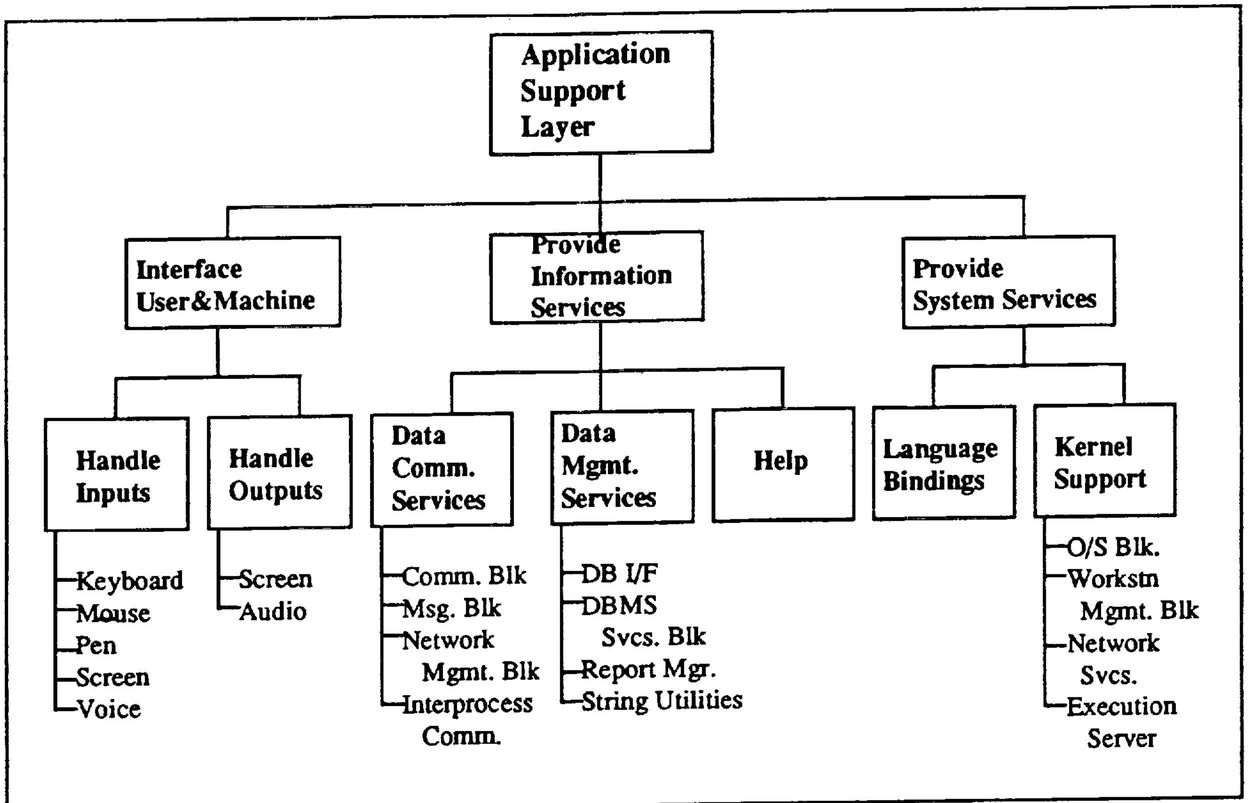


Figure 5 A sample of a draft hierarchical model for the Application Support Layer domain.

One facet of understanding a system that the functional and hierarchical diagrams did not illustrate very well was the commonalities across the different ASL subsystems. This aspect of the system was depicted more accurately by using object-oriented models. Object-oriented models pull all occurrences of the same data-type together, grouping all attributes and operations. Details focus on the data instead of on the functions. This permits code to be written with emphasis on the data being generic or abstract. This data abstraction increases the reusability of the software components - requirements architectures, design models, and code.

Three high-level object-oriented models created during this quick domain analysis were the focus of reuse opportunities identification. These models were the Data Base Management Systems model, the Platform Services model, and the User-Machine Interface model. A simplified version of the Platform Services Object model appears in Figure 6. Note, all data attribute and operation information has been removed in this version of the figure to improve the readability of the model.

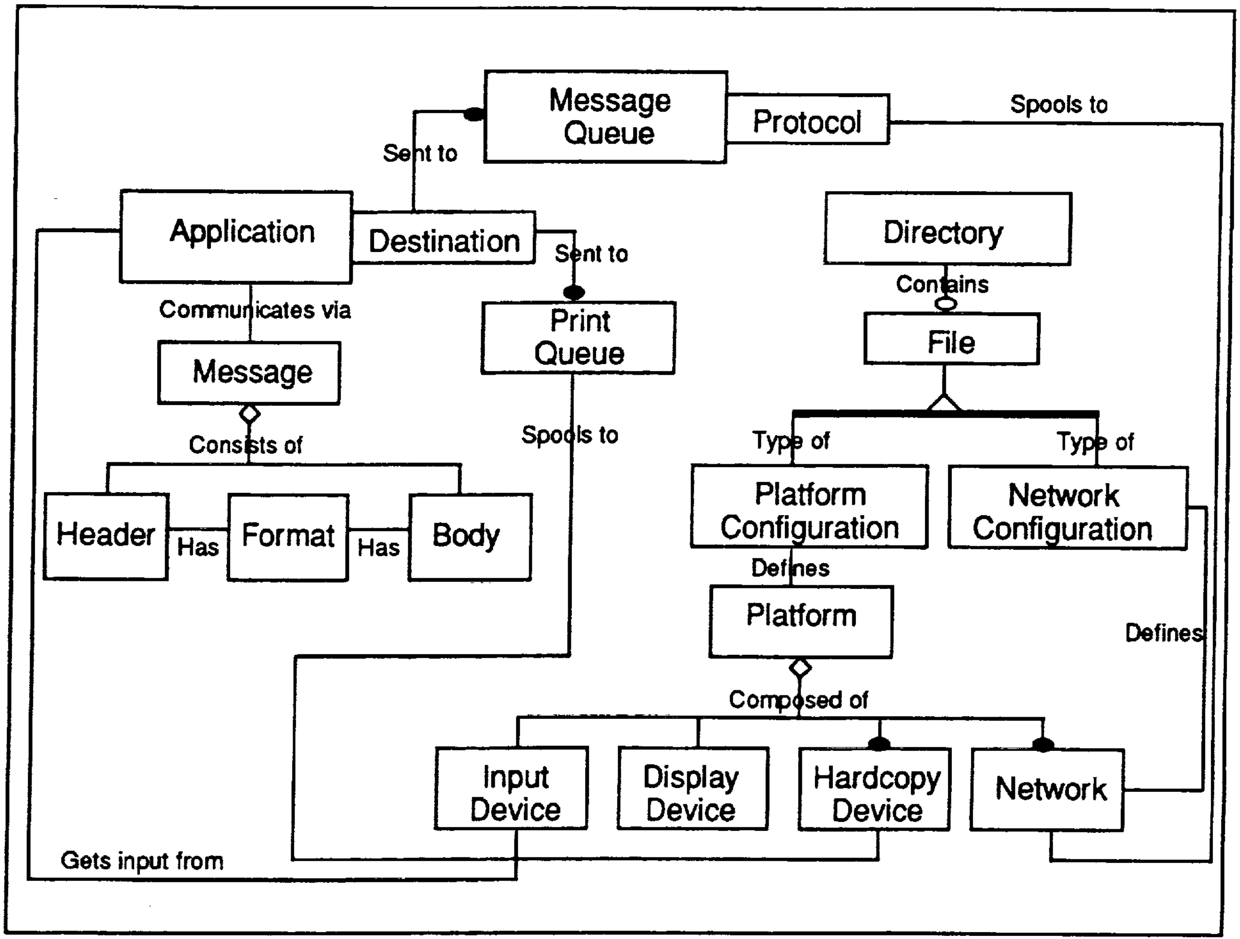


Figure 6 This is a simplified version of the Platform Services Object Model which was used to identify the software's basic functionality.

Unlike traditional domain analysis efforts, the primary objective in developing these domain models was not to explicitly define all the details of each of the primitive functions in the domain. Instead, this effort tried to provide an overview of the data relationships and basic interactions. By determining the general data manipulations of a typical ASL, the categories of components which are critical to the functionality of this domain were pinpointed. For example, as shown in Figure 6, specifics of the network were not needed, but understanding the relationship of the network configuration with the rest of the platform configuration proved very useful. The interaction of the ASL and the hardware support layer provided the distinction between potentially reusable software components and those hardware dependent components which require code, for instance, to be system unique.

Full lists of the data attributes and operations were developed for each object in each model. This permitted each object to be treated as a black box; that is, no further breakdown into sub-objects was necessary to expose software functionality. One case of this occurred with the object Platform. One of the operations associated with the object Platform is Enable/Disable Security. This single operation highlighted the importance current software development efforts place on security functions. Security functions are embedded throughout many software products, at multiple software layers. Though the Platform Services Object model does not provide further details of security functions, the high demand that software developers have placed on this category of software was not lost by this analysis team. Security functions were considered as prime reusable component candidates.

Besides providing a visual representation of the domain to assist in reuse opportunity identification, this process of integrating multiple modeling techniques offers an additional benefit. Though the faceted domain analysis approach described by Prieto-Diaz could have been performed to identify reuse opportunities, no product would have been available for future reusable component development. Typically, domain analysis is considered to be divided into two types:

o	Consumer-oriented	associated with reuse opportunities identification, and
o	Producer-oriented	associated with the creation of reusable components [Moore-Bailin].

However, the object-oriented models and their supporting documentation produced by the procedure described in this paper can be used as a basis for reusable requirements models. All of the high-level information on the domain is available in these models and many of the domain component details can gleaned from the analysis process documentation. Therefore, the final object-oriented models produced by this process not only meet current needs, but also some of those for future reuse planning.

Model and Reuse Opportunities Identification

The purpose of performing reuse opportunity identification is to facilitate reuse within one or among several system development efforts. During reuse opportunity identification, systems are evaluated and selected as candidates for reusing software components in their development life-cycle (client systems) and/or for developing and providing reusable software components to support the software development life-cycle of other systems (donor systems). Each potential client and donor system's schedule, language and functionality are studied. This information, along with data on the organization's policies, reuse knowledge and experiences, reuse training, and any other information that might facilitate or limit reuse is researched.

A system's schedule together with the high demand categories (HDCs) of components included in a system are the most crucial pieces of information needed when trying to coordinate reuse between compatible systems. HDCs are classifications of software components that are defined as being a necessity or requirement of all the systems that are in a particular domain. The HDCs are chosen by domain engineers using the generic architectures and domain models resulting from domain analysis. HDCs may be either functional or object-oriented in nature. For this reason, the study of the system's functionality, as well as, the data or object-oriented aspects of the each system involved in the reuse opportunities analysis is important.

This need of both functional and object-oriented views is where fusing modeling techniques proved to be very beneficial. For example, the HDCs that evolved from the domain analysis effort on the application support layer included process network messages, manage data dictionary, user machine interface, and database management system.

Once the analysis team had established the high demand categories from the domain models, we had a basis from which to identify reuse opportunities. We then took the potential client and donor system's schedules and identified which systems would be the clients and which systems would be the donors. This schedule coordination is critical to performing successful reuse opportunity analysis. The goal in this type of analysis is to begin identifying the client-donor relationships as early as possible in the client system's software development life-cycle (i.e. before requirements analysis, if possible). This permits the software reuse to be planned into the client systems' development schedule and thus, the largest cost benefits can be realized.

For systems which have similar software development schedules, if a client-donor relationship is established early enough the systems can perform requirement analysis or design development as a team. Then, one system could be chosen to write the reusable code and donate it to the other. Or the systems could split the code development effort and swap the highly reusable pieces before system integration testing.

After finding compatible systems according to schedule restrictions, the analysis team took the products produced from the conceptual phase and/or the requirements from the client system (depending on where the system was in the life cycle) and matched them to the requirements and design of the donor system. The HDCs and the generic models also guided this matching process by helping the analysts determine what was reusable and what was application specific. Since the generic domain models produced represent what is common (or reusable) among all systems in the ASL domain the analysts using the models were able to quickly identify potential opportunities for opportunistic and systematic reuse.

Summary

The initial use of this fusion of modeling techniques resulted in the development of a complete, understandable high-level object-oriented ASL domain model. Since that time, the technique has been applied successfully to the analysis efforts of other vertical domains including the personnel and budget domains. In most of these efforts, this fused modeling technique was employed to permit a very fast high-level domain analysis for the purpose of reuse opportunities identification. Since traditional domain analysis can take several person years per domain, this quick process (measured in terms of person months, not years) proved to be substantially cost effective.

However, our experience indicates that using multiple types of modeling techniques closely linked together should enhance traditional domain and system analysis efforts in general. Multiple views of a software modules functionality permits easier identification of reuse opportunities, quickly locates inconsistencies in system design, and encourages the development of more complete, reliable software products.

Ms. Susan Main Hall is a Systems Consultant, Management, for SofTech, Incorporated. She directs a technical group which supports the Army Reuse Center through domain analysis, reuse requirements analysis, reuse opportunities identification, library donor component selection, and quality assurance of reusable software components. Additionally, Ms. Hall has over eight years experience in supporting DoD Ada technical development efforts. She has participated in independent verification and validation, modeling, and development. Ms Hall holds a Bachelors of Science degree in Computer Science and a Masters of Science degree in Computer Science with Software Engineering concentration from George Mason University.

Ms. Eileen M. McGuire is an Associate Software Engineer for SofTech, Incorporated. She preforms domain analysis, reuse requirements analysis, reuse opportunities identification, and library donor component selection. Ms. McGuire holds a Bachelors of Science degree in Management Science (Computer Based Decision Support Systems Option) from Virginia Polytechnic Institute and State University.

References

Blaha, Michael, "Models of Models," September 1991

Caldiera, G. and V. R. Basili, "Identifying and Qualifying Reusable Software Components," IEEE Computer, Vol. 24, No. 2, Feb 1991, pp. 61-70

Coad, and Yourdon, **Object-Oriented Analysis**
Englewook Cliffs, NJ: Yourdon Press/Prentice Hall, 1991

Coleman, Derek, Fiona Hayes and Stephen Bear, "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design," IEEE Transactions on Software Engineering, Vol. 18, No. 1, January 1992

Domain Analysis Guidelines, Draft, SofTech, Inc., May 1992

DeMarco, T., **Structured Analysis and System Specification.** Englewook Cliffs, NJ: Yourdon Press/Prentice Hall, 1978

Fiscal Year 1994 Reuse Opportunities Report, Final, SofTech, Inc., July, 30, 1993

Gomaa, H., L. Kerschberg, C. Bosch, V. Sugumaran and I. Tavakoli, "A Prototype Software Engineering Environment for Domain Modeling and Reuse," 1991

Iscoe, Neil, "Reuse - A Knowledge Based Approach," NASA Software Engineering

Workshop Proceedings, December 1992

Jacobson, Ivar and Frederik Lindstrom, "Re-engineering of Old Systems to an Object-Oriented Architecture," OOPSLA'91

Lubars, Mitchell D., "Domain Analysis and Domain Engineering in IDeA," IEEE 1991

McGarry, Frank, "Lessons Learned", NASA Software Engineering Workshop Proceedings, December 1992

Moore, John M. and Sidney C. Bailin, "Domain Analysis: Framework for Reuse Technical Report", Computer Technology Associates, Rockville, MD, 193

Patel, Sukesh, William Chu, Rich Baxter, Brian Sayrs and Steve Sherman, "A Top-Down Software Reuse Support Environment," 1992

Prieto-Diaz, Ruben, "Domain Analysis: An Introduction," Software Engineering Notes, Vol. 15, No. 2, April 1990

Prieto-Diaz, Ruben, "Domain Analysis for Reusability," Proceedings of COMPSAC '87, pp. 23-29

Rumbaugh, James, Michael Blaha, Wiliam Premerlani, Frederick Eddy and William Lorensen, **Object-Oriented Modeling and Design**, Prentice-Hall, Inc., 1991

Shumate, Ken, "BATCES Solution #1" an Object-Oriented Design from Functional Requirements Analysis," ACM Ada Letters, Nov/Dec 1993, Vol. XIII, Number 6, pp. 133-161

Tracz, Will, "Domain Analysis Working Group Report - First International Workshop on Software Reusability," Software Engineering Notes, Vol. 17 No. 3, July 1992

Wessale, Bill, "Large Project Experience with Object Oriented Methods and Reuse," NASA Software Engineering Workshop Proceedings, December 1992

Fusing Modeling Techniques to Support Domain Analysis for Reuse Opportunities Identification

by
Susan Main Hall
Eileen McGuire

SOFTECH, INC.

Project Background

(Army Reuse Center, Software Development Center-Washington)

Needed quick methodology to:

- **Perform Domain Analysis**
  - Application Support Layer Services Domain
  - Four Army systems currently under development
  - Systems and analysts were functionally oriented

- **Focus on Identification of Reuse Opportunities**
  - Object-oriented models make this process easier

SOFTECH, INC.

Problem Statement

Which provide a clearer understanding of high-level system requirements:

Functional Models

or

Object-Oriented Graphical Representations

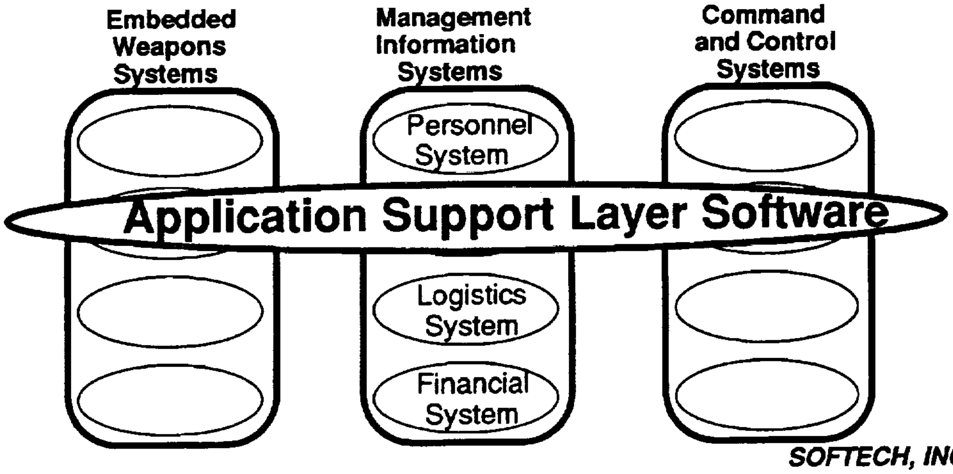
? ? ? ? ? ? ? ? ? ?

SOLUTION: Use BOTH!

SOFTECH, INC.

Domain Analysis

Process of identifying commonalities in a class of similar systems


Embedded Weapons Systems
Management Information Systems
Command and Control Systems
Personnel System
Application Support Layer Software
Logistics System
Financial System

SOFTECH, INC.

Fusing Modeling Techniques

Since one modeling technique could not capture the domain analysis completely, we:

- Started with FUNCTIONAL models
- Moved to a functional HIERARCHICAL GROUPING model
- Transitioned into a set of OBJECT-ORIENTED models

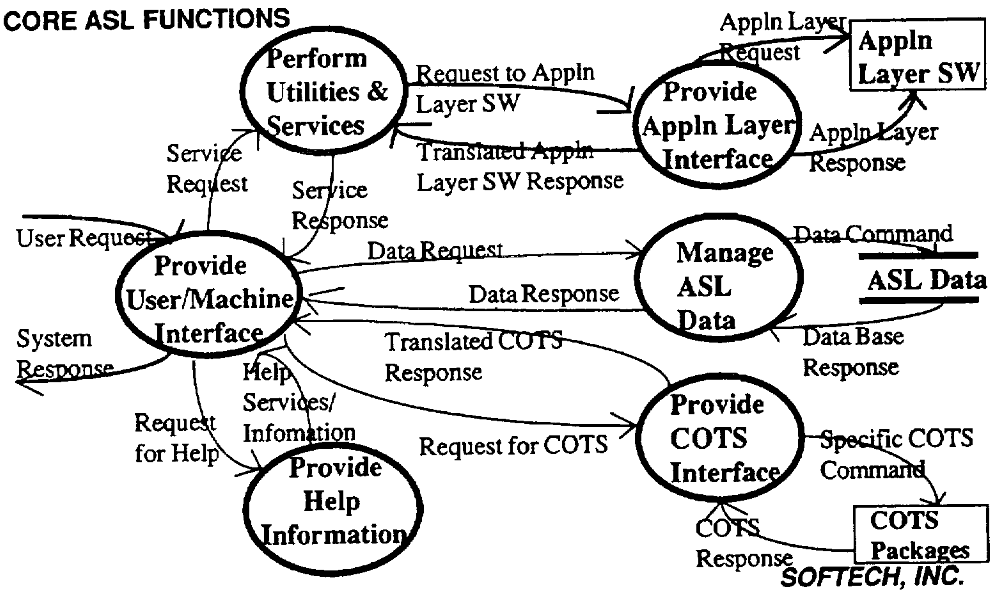
SOFTECH, INC.

Functional Models

- Began by reviewing existing and creating new:
  - data flow diagrams
  - state transition diagrams
  - flow charts
- Captured basic activities performed by the actual Application Support Layer code module being studied
- Overlaid each system on top of one another to highlight commonalities and differences

SOFTECH, INC.

Functional Models (continued)



SOFTECH, INC.

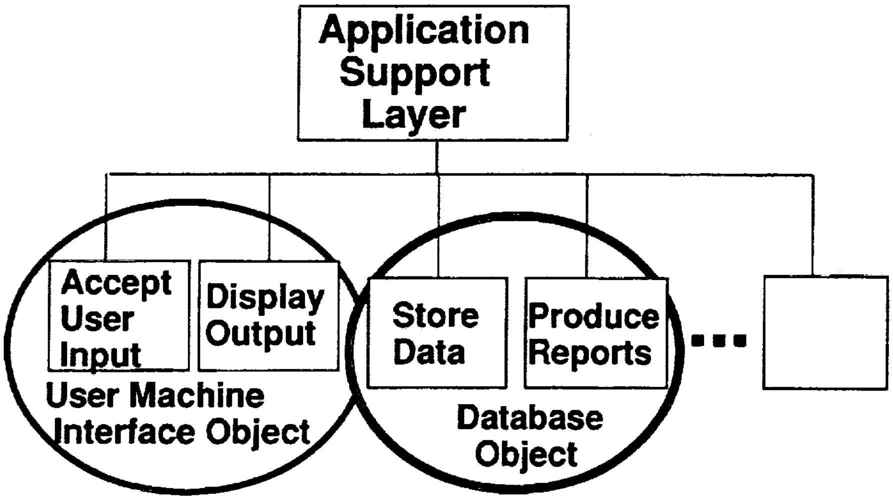
Heirarchical Models

- Moving from a functional to an object model can cause important information to be lost.
- So a home-grown technique was applied.
- This heirarchical technique consists of:
  - –listing the identified functions in a heirarchical tree
  - –grouping the lowest level functions together based on the objects manipulated
  - –dividing functions into those IN the domain and those INTERFACING WITH the domain.

SOFTECH, INC.

Heirarchical Models (continued)

SAMPLE OF HEIRARCHICAL
GROUPING TECHNIQUE



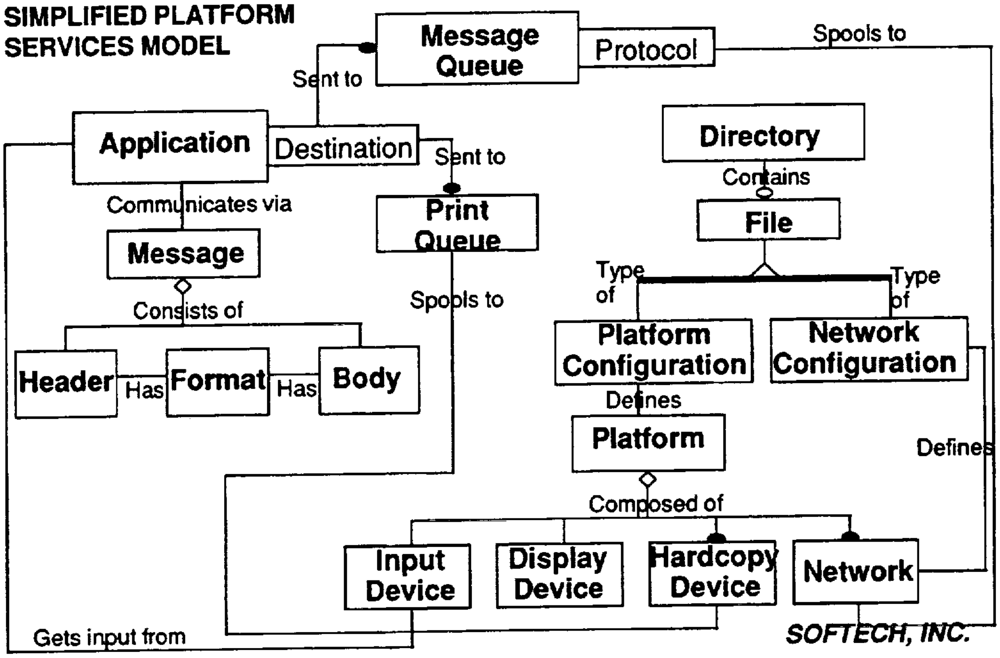
SOFTECH, INC.

Object-Oriented Models

- Used to illustrate commonalities across Application Support Layer sub-systems
- Grouped occurrences of same data-type together
- Captured data attributes
- Assigned functions to data
- Provided level of data abstraction to increase reusability

SOFTECH, INC.

Object Oriented Models (continued)



Reuse Opportunities Identification

Purpose: to facilitate reuse within one or among several softare development efforts.

- During reuse opportunities identification, systems are evaluated and selected as candidates to:
  - reuse software components in their software development life-cycle (clients)

  AND/OR
  - provide reusable software components to support the software development life-cycle of other systems

SOFTECH, INC.

Models Assisted ROI

- Application Specific/System Unique components were stripped away, using the functional and heirarchical models.
- High Demand Categories were established, using the domain models
  - Functional
  - Data-Oriented.
- Reusable software components were identified, factoring in development schedules
  - Requirements Architectures,
  - Design Models,
  - Code Modules.

SOFTECH, INC.

Summary

- Multiple views illustrate a domain more clearly than a single modeling approach
- This fusion of modeling techniques approach:
  - identified more substantial reuse opportunity candidates,
  - completed more quickly than traditional domain analysis, and
  - provided a basis for future developement of a reusable domain model.

SOFTE

1-61
12696
p. 23

An Empirical Comparison of a Dynamic Software Testability Metric to Static Cyclomatic Complexity*

Jeffrey M. Voas
RST Corp.
11150 Sunset Hills Road
Suite 250
Reston, VA 22090 USA
(703) 742-8873
jmvoas@isse.gmu.edu

Keith W. Miller
Dept. of Computer Science
Sangamon State University
Springfield, IL USA
(217) 786-6770

Jeffery E. Payne
RST Corp.
11150 Sunset Hills Road
Suite 250
Reston, VA 22090 USA
(703) 742-8873

Abstract

This paper compares the dynamic testability prediction technique termed "sensitivity analysis" to the static testability technique termed cyclomatic complexity. The application that we chose in this empirical study is a CASE generated version of a B-737 autoland system. For the B-737 system we analyzed, we isolated those functions that we predict are more prone to hide errors during system/reliability testing. We also analyzed the code with several other well-known static metrics. This paper compares and contrasts the results of sensitivity analysis to the results of the static metrics.

I. Introduction

The adage that non-exhaustive software testing cannot reveal the absence of errors and only their existence is as true today as it was when Dijkstra wrote it [4, 1]. Unfortunately, between the time then and now, we have begun to build orders-of-magnitude more complex systems while our testing technologies are no more advanced. Thus the same problems that we had in past years when testing a 1000 line program are compounded when we apply those techniques to a 10M line program today.

We must admit that we are building software systems that are destined to be inadequately tested. Since we know this *a priori*, it suggests that we should look for techniques that aid the testing process where the process is known to be weak. In this paper, we discuss one such technique: a method that quantifies the dynamic testability of a system that is undergoing system/reliability testing. We will then compare the results of this technique to other metrics that are in wide-spread use today.

Software testing is performed for generally two reasons: (1) detect faults so that they can be fixed, and (2) reliability estimation. The goal of the dynamic testability measurement technique presented in this paper is to strengthen the software testing process as it applies to reliability estimation. Dynamic testability analysis is less concerned with fault detection, even though it is plausible that a function that is more likely to hide faults during system testing may also be more likely to hide faults during unit testing. Instead, dynamic testability analysis is concerned with a *lack* of fault detection.

II. Static Software Metrics

The study of software metrics has grown out of a need to be able to express quantitative properties about programs. The first software metric was simply a count of the number of lines. This was acceptable as a way of measuring program size, but was not applicable to other software characteristics.

Software complexity is another metric that tries to relate how difficult a program is to understand. In general, the more difficult, the more likely that errors will be introduced, and hence the more testing that will be required. Thus it is common for developers to relate a software complexity measurement to the allocation of testing resources. It is our experience, however, that software complexity is still too coarse-grained of a metric to relate to the testing of *critical software systems*, those that must fail less than once in 10^9 executions (or some other large number). Thus even though software complexity can be useful as a first-stab at how much testing to perform and where, it is too coarse for assessing reliability in the ultra-reliable region of the input space.

In this paper, we have considered 6 software metrics that are available in the PC-METRIC 4.0 toolkit: (1) Software Science Length (N) (2) Estimated Software Science Length ($N^\wedge$), (3) Software Science Volume (V), (4) Software Science Effort (E), (5) Cyclomatic Complexity (VG1), and (6) Extended Cyclomatic Complexity (VG2). We will briefly mention what these metrics are; in general, any software engineering text will go into more depth on these metrics for the inquisitive reader.

Halstead [2] observed that all programs are comprised of operators and operands. He defined N_1 to be the number of total operators and N_2 to be the number of total operands. He defined length of a program, N, to be:

$$N = N_1 + N_2.$$

Halstead also has a predicted length metric, $N^\wedge$, that is given by:

$$N^\wedge = n_1 \cdot log_2(n_1) + n_2 \cdot log_2(n_2),$$

where n_1 is the number of unique operators and n_2 is the number of unique operands. Halstead has another metric that he terms volume, V, that is given by:

$$V = N \cdot log_2(n_1 + n_2).$$

Halstead's Effort metric, E, is given by:

$$E = V/L,$$

however most researchers use [5]:

$$E = V/(2/n_1 \cdot n_2/N_2)$$

McCabe's cyclomatic complexity metric is less based on program size (as are Halstead's measures) and more on information/control flow:

$$V(g) = e - n + 2$$

where n is the number of nodes in the graph and e is the number of edges, or lines connecting each node. It is the cyclomatic complexity metric that we are more interested in for this paper, and most importantly how cyclomatic complexity compares to the dynamic testability measure presented in Section 3.

III. Testability Analysis

Software testability analysis measures the benefit provided by a software testing scheme to a particular program. There are different ways to define the "benefit" of tests and testing schemes, and each different definition requires a different perspective on what testability analysis produces. For instance, software testability has sometimes been referred to as the ease with which inputs can be selected to satisfy structural testing criteria (e.g., *statement coverage*) with a given program. With this perspective, if it were extremely difficult to find inputs that satisfied a structural coverage criteria for a given source program, then that program is said to have "low testability" with respect to that coverage criteria. Another view of software testability defines it as a prediction of the probability that existing faults will be detected during testing given some input selection criteria C. Here, software testability is not regarded as an assessment of the difficulty to select inputs that cover software structure, but more generally as a way of predicting whether a program would reveal existing faults during testing according to C.

In either definition, software testability analysis is a function of a (*program, input selection criteria*) pair. Different input selection criteria choose test cases differently: inputs may be selected in a random black-box manner, their selection may be dependent upon the structure of the program, or their selection may be based upon other data or they may be based on the intuition of the tester. Testability analysis is more than an assertion about a program, but rather is an assertion about the ability of an input selection criteria (in combination with the program) to satisfy a particular testing goal. The same syntactic program may have different testabilities when presented with different input selection criteria.

In order for software to be assessed as having a "greater" testability by the semantic-based definition, it must be likely that failure occurs if a fault were to exist. To understand this likelihood, it is necessary to understand the sequence of events that lead to software failure. (By software failure, we mean an incorrect output that was caused by a flaw in the program, not an incorrect output caused by a problem with the environment or input on which the program is executing.) Software failure only occurs when the following three conditions occur in the following sequence:

1. A input must cause a fault to be *executed.*
2. Once the fault is executed, the succeeding data state must contain a *data state error.*

3. After a data state error is created, the data state error must *propagate* to an output state.

The semantic-based definition of testability predicts the probability that tests will uncover faults if a fault exists. The software has high testability for a set of tests if the tests are likely to detect any faults that exist; the software has low testability for those tests if the tests are unlikely to detect any faults that exist. Since it is a probability, testability is bounded in a closed interval [0,1]. In order to make a prediction about the probability that existing faults will be detected during testing, a testability analysis technique should be able to quantify (meaning predict) whether a fault will be executed, whether it will *infect* the succeeding data state creating a data state error, and whether the data state error will propagate its incorrectness into an output variable. When all of the data state errors that are created during an execution do not propagate, the existence of the fault that trigged the data state errors remains hidden, resulting in a lower software testability.

Software sensitivity analysis is a code-based technique based on the semantic definition of testability; it injects instrumentation that contains program mutation, data state mutation, and repeated executions to predict a minimum non-zero fault size [7, 13]. The minimum non-zero fault size is the smallest probability of failure likely to be induced by a programming error based upon the results of the injected instrumentation. Sensitivity analysis is not a testing technique, and thus it does not use an oracle, and can be completely automated (provided that the user initially tells the technique where in the code to apply the analysis).

Software sensitivity analysis is based on approximating the three conditions that must occur before a program can fail: (1) execution of a software fault, (2) creation of an incorrect data state, and (3) propagation of this incorrect data state to a discernible output. This three part model of software failure [9, 10] has been explored by others, but not in the manner in which sensitivity analysis explores it. In this paper we examine how to apply sensitivity analysis to the task of finding a realistic minimum probability of failure prediction when random testing has discovered no errors.

In the rest of this section we give a brief outline of the three processes of sensitivity analysis. To simplify explanations, we will describe each process separately, but in a production analysis system, execution of the processes would overlap. As with the analysis of random testing, the accuracy of the sensitivity analysis depends in part on a good description of the input distribution that will drive the software when operational (and when tested).

Before a fault can cause a program to failure, the fault must be executed. In this methodology, we concentrate on faults that can be isolated to a single *location* in a program. This is done because of the combinatorial explosion that would occur if we considered distributed faults. A location is defined as a single high level language statement. Our experiments to date have defined a location as a piece of source code that can change the data state (including input and output files and the program counter). Thus an assignment statement, **if**, and *while* statement define a location. The probability of execution for each location is determined by repeated executions of the code with inputs selected at random from the input distribution. An automated testability system, *PiSCES* [11], controls the instrumentation and bookkeeping.

If a location contains a fault, and if the location is executed, the data state after the fault may or may not be changed adversely by the fault. If the fault does change the data state into an incorrect data state, we say the data state is *infected*. To estimate the probability of infection, the second process of sensitivity analysis performs a series of syntactic mutations on each location. After each mutation, the program is re-executed with random inputs; each time the location is executed, the data state is immediately compared with the data state of the original (unmutated) program at that same point in the execution. If the internal state differs, infection has taken place. And this is recorded by *PiSCES* and reported back to the user.

The third process of the analysis estimates propagation. Again the location is monitored during random tests. After the location is executed, the resulting data state is forcefully changed by assigning a random value to one data item using a predetermined internal state distribution. After the internal data state is changed, the program continues executing until an output results. The output that results from the changed data state is compared to the output that would have resulted without the change. If the outputs differ, propagation has occurred and *PiSCES* reports that back to the user as a probability estimate.

For a test case to reveal a fault, execution, infection, and propagation must occur; without these three events occurring, the execution will not result in failure. And for a specific fault, the product of the probability of these events occurring is the actual probability of failure for that fault. Each sensitivity analysis process produces a probability estimate based on the number of trials divided by the number of events (either execution, infection, or propagation). The product of these estimates yields an estimate of the probability of failure that would result when this location contains a fault. Since we are approximating the model of how faults result in failures, we also take this multiplication approach when we predict the minimum fault size and multiply the minimum infection estimate, minimum propagation estimate, and execution estimate for a given location. This produces the testability of that location. We then take the location with the lowest non-zero testability to be the testability of the overall program.

IV. PiSCES

Several proof-of-concept sensitivity analysis prototypes were built in the early 1990s. *PiSCES* is the commercial software testability tool that evolved from these prototypes. *PiSCES* is written in C++ and operates on programs written in C. The recommended platform for *PiSCES* is a Sparc-2 with 16 mbytes of memory, 32 mbytes of swap space, and 20 mbytes of hard disk space. For larger C applications, the amount of memory that *PiSCES* needs increases, and thus we currently are limited to running around 3,000-4,000 lines of source code at a time through *PiSCES*. For larger systems, we perform analysis on a part of the code, and when that is done, we perform analysis on another part until all of the code has received dynamic testability analysis. This "modular approach" is how we get results for systems larger than 4,000 SLOC.

PiSCES produces testability predictions by creating an "instrumented" copy of your program and then compiling and executing the instrumented copy. Although it is hard to determine precisely, given the default settings that *PiSCES* uses, the instrumented version of your program is approximately 10 times as large as the original source code. The instrumented copy is then executed with inputs that are either supplied in a file or *PiSCES* uses random distributions from which it generates inputs.

V. Dynamic Testability Results

We were supplied with a C version of a B-737 autopilot/autoland that had been generated by a CASE tool; the CASE tool has been under development by NASA-Langley and one of their vendors for several years. We were told that as far as NASA knew, this version of the autopilot/autoland had never failed; it appears to be a correct version of the specification. The version consisted of 58 functions; parameters to the system included information such as direction of wind, wind speed, and speed of gusts. The version we used is not embedded in any commercial aircraft. Instead, the version is based on the specification of the system that is embedded on aircraft, and hence this code should contain most (if not all) of the functionality of the production aircraft system.

We should mention that the B737 source code was approximately 3000 SLOCs, and it represents the largest program to receive sensitivity analysis in its entirety to date. Our results here are based on 2000 randomly generated input cases that are correlated to the following types of landing conditions:

1. no winds at all,
2. moderate winds, and
3. extremely strong winds with high gusts.

(We think it was important to exercise three major classes of scenarios that the system would encounter in operation.) We should mention that we found similar results [12] when we used a different test suite with 1000 randomly generated inputs. The total amount of clock time that it took for *PiSCES* analysis to run and produce the results was 55 hours on a Sparc-2 (there were no other major jobs running on that platform during this time).

According to the Squeeze-Play model for testing sufficiency for the B-737 version, sensitivity analysis recommends 11,982,927 system level tests [8]. This is based on the conservative testability prediction of $< 2.5E - 07$ for the entire program; ***a conservative testability translates into a liberal estimate of the amount of needed testing.*** We use a conservative testability to ensure that we are not fooled into believing that we have done enough testing when we really have not. A testability written as an inequality indicates that *PiSCES* encountered at least one location that did not execute or propagate during analysis. One possible quantification of this situation is to assign a testability of 0.0, but that creates problems for further analysis. Instead, *PiSCES* makes a reasonable estimate on testability and signals the singularity with the inequality. The process for doing this as well as the mathematics are described in the *PiSCES Version 1.0 User's Manual*. (A 0.0 testability produces an infinite amount of testing needed which is useless to testers.) Since testing on approximately 12 million inputs is impractical, there are other alternatives for increasing the testability; if these alternatives are applied successfully, they will decrease the number of tests required, but we will not explain here how that is done.

We now show the results for the 58 individual functions of the B737.c system in Figures 1 and 2. As you can see, there are 15 functions out of the 58 that have a testability of greater than 0.01. These are functions that the developer/tester need not worry over; they appear to have little fault hiding ability. This information also tells the developer which functions (the other 43) are more worrisome (in terms of hiding faults at the system level of testing); by immediately isolating those functions of low testability, we gain insight as to where additional testing resources are needed. Note that the degree to which we consider a function to be "worrisome" is a function of how much testing is considered feasible.

As you can see from the bar charts, there were many functions of low testability. This does not say that these functions are incorrect (recall that this program has never failed for NASA), but rather that these functions should receive special consideration during V&V. In our tool, there are ways of decreasing the recommended testing costs if the user knows that the regions of the code where the low testabilities occur are not hiding faults. Although such knowledge is difficult to obtain, it does provide the user with a justifiable way of performing code inspections and testing sufficiency.

VI. Static Metric Results

This section provides several sets of data that we collected from the B737 source code when we ran it through a commercial metrics package with the default settings. Table 1 and Table 2 display the results that were attained by running PC-METRIC 4.0 [5].

VII. Comparison of Results

Some software researchers and practitioners have equated testability with McCabe's cyclomatic complexity or some other static metric. We contend that such static measures do not capture the dynamic, data dependent nature that is fundamental to testing and our analysis of the effectiveness of testing.

In 1990, we introduced both a new definition of testability and a new method for measuring testability based on our definition. Still, we are frequently asked how our definition compares to cyclomatic complexity, which we feel is a valid question. In this section, we will try to show how these two measurement methods differ, and what these differences mean for the typical tester or QA manager.

As we have shown in Section 5, the B737 code had functions of high testability and lower testability. If the reader then considers Table 1, we immediately see that the VG1 values for the functions of B737 never exceeded 7, and for VG2, the functions never exceeded 10. According to the cyclomatic complexity measures, all of these functions are labeled as "not complex;" however sensitivity analysis has found that many of these functions are more likely to hide faults during testing than McCabe's numbers might suggest.

Our interpretation for why this is true is simply how the two metrics view a program; sensitivity analysis is based on the semantic meaning of the program, whereas cyclomatic complexity is based on an abstract and structural view of the program. It is true that the structural view has some impact on the semantics of the program, however during system level testing, we argue that the information provided by cyclomatic complexity is essentially useless in terms of how much testing to perform. Thus we conclude that for unit testing, cyclomatic complexity is an easy means of attaining a feeling for how good the structure of the program is (essentially as a "spaghetti" code type of measure), however for critical systems, we contend that the semantic perspective on testability provided by sensitivity analysis is far more valuable. Sensitivity analysis costs more, but the value added is also increased.

VIII. Conclusions

We contend that the preliminary results of experiments in software sensitivity are sufficient to motivate additional research into quantifying sensitivity analysis [13, 6]. Not only do we think that this technique may hold promise in assessing critical systems, but in Hamlet's award winning *IEEE Software* paper [3] and The National Institute of Standards and Technology's report on software error analysis [14], sensitivity analysis is acknowledged as a technique that should be further explored for its potentially enormous impact on assessing ultra-reliable software.

Although the subprocesses of sensitivity analysis will in all likelihood require minor revisions as more is learned about fault-based analysis, the ideas that motivate sensitivity analysis dispute the contention that software testing is the only method of experimentally

Procedure	N	$N^\wedge$	V	E	VG1	VG2
LIMITER	37	46	145	2186	3	3
LIM_180	59	72	259	4001	3	3
ONED	137	156	714	25760	3	7
INTEGRATE	45	57	188	1642	3	3
FOLAG	43	68	186	2243	2	2
WASHOUT	53	72	233	3215	2	2
STATE	17	24	56	329	1	1
EZSWITCH	62	113	301	4694	2	3
KOUNT	66	71	290	3508	4	4
MODLAG	130	185	701	16964	2	3
MODLAG_1	108	76	482	7706	7	7
MODLAG_2	35	64	149	1772	2	3
MODLAG_3	68	82	308	5263	3	3
DZONE	22	40	84	603	1	1
AUTOPILOT	276	610	1842	25716	2	2
MODE	172	323	1016	5166	1	1
MODES_2	178	351	1072	7309	1	1
MODE1	139	209	763	6940	1	10
MODE2	120	210	663	8948	1	8
CALC_GSTRK	120	134	605	9837	6	8
MODE3	109	168	576	7615	1	7
THROT	73	144	368	1432	1	1
ATHROT	180	310	1072	21174	5	7
PRE_FLARE	79	168	418	4840	2	2
VER_SINE	29	57	121	605	1	1
WIND_SHEAR	99	151	512	6763	1	1
SPEEDC	18	36	67	320	1	1
AFTLIM	58	97	273	3926	3	6
EPR_GAIN	43	72	189	2302	2	2
LONG_x	111	242	620	2391	1	1
LONGAP	226	399	1408	28817	3	3
CALC_HR	15	28	52	227	1	1
FLARE_CONTROL	135	185	728	12557	5	5
CALC_HDER	39	72	171	1485	2	2
PRE_FLARE_LONG	98	185	528	7332	2	2
IN_ON_BEAM	39	71	171	1225	2	2
GSE_ADJ	37	67	160	1270	1	1
BEFORE_GSE	51	96	240	2657	1	2
BANK_ADJ	17	33	61	457	1	1
PITCH_ADJ	39	81	176	1491	2	2
CAS_ADJ	25	53	102	657	1	1
LATERAL	115	270	656	2394	1	1
LATAP	192	362	1173	19388	2	2
LOC_ERROR	78	128	390	4095	2	2
CROSS_VEL	18	36	67	320	1	1
LOCCMD	76	128	380	5510	2	2
LOCINT	46	101	219	1884	1	2
LOCCF	78	117	383	4521	4	4
CROSSTKADJ	33	71	145	957	2	2
BANK	90	156	469	5894	3	4
PHICMDFB	83	145	426	5830	2	3
RtoA_XFD	103	191	559	7847	2	2
CALC_PSILIM	57	76	254	2898	3	3
SPOILER	51	81	231	2391	2	2
AIL_CMD	97	169	513	5287	4	4
RUDDER_CMD	75	145	385	3751	3	3
OUTERLOOPS	39	69	169	650	1	1
AUTOOL	69	117	339	3386	3	4

Table 1: Software Science Length (N) Estimated Software Science Length ($N^\wedge$), Software Science Volume (V), Software Science Effort (E), Cyclomatic Complexity (VG1), and Extended Cyclomatic Complexity (VG2) for B737.c

Metric	Score
Software Science Length (N):	4707
Estimated Software Science Length ($N^\wedge$):	4107
Software Science Volume (V):	42147
Software Science Effort (E):	7963380
Estimated Errors using Software Science ($B^\wedge$):	13
Estimated Time to Develop, in hours ($T^\wedge$):	123
Cyclomatic Complexity (VG1):	70
Extended Cyclomatic Complexity (VG2):	111
Average Cyclomatic Complexity:	1
Average Extended Cyclomatic Complexity:	1
Average of Nesting Depth:	1
Average of Average Nesting Depth:	0
Lines of Code (LOC):	3312
Physical Source Stmts (PSS):	2683
Logical Source Stmts (LSS):	569
Nonexecutable Statements:	861
Compiler Directives:	9
Number of Comment Lines:	1384
Number of Comment Words:	1985
Number of Blank Lines:	629
Number of Procedures/Functions:	58

Table 2: Summary of Static Metric Scores

quantifying software reliability. We believe that dynamic testability analysis is a new form of software validation, because it is quantifying a semantic characteristic of programs. We cannot guarantee that sensitivity analysis will assess reliability to the precisions required for life-critical avionics software, because as we have pointed out, low testability code can never be tested to any threshold that would strongly suggest that faults are not hiding. However, we do think it is premature to declare such an assessment impossible for all systems, and we feel that this topic deserves attention both from the avionics community as well as the software engineering and testing communities.

This experiment demonstrates important differences between static and dynamic analysis of how much testing is required. Admittedly, dynamic information is far more expensive to attain; but for the additional cost, the precision derived we feel is justified. This expense comes mainly from the fact that the input space and probability density function are also considered when assessing how much testing is necessary, not only the structure of the code. And this expense is in computer time, not human time.

We have felt that static software metrics are too assumption-based to be useful for predicting how to test critical systems. For this reason, we developed a new perspective on testability, a new way of measuring that definition, and commercialized a tool to perform the measurement.

Acknowledgement

This experiment was funded through NASA-Langley Grant NAG-1-884. The authors thank Carrie Walker for supplying us with the B737 auto-generated code from the ASTER tool.

Disclaimer

The code supplied to us was from NASA and not Boeing, and as far as we know, this code and the testability results do not reflect the quality of the software used in Boeing aircraft or produced by Boeing. Boeing is in no way affiliated with this experiment nor RST Corporation.

References

[1] E. Dijkstra. Structured Programming. In *Software Engineering, Concepts, and Techniques.* Van Nostrand Reinhold, 1976.

[2] M. H. Halstead. *Elements of Software Science.* New York:Elsevier North-Holland, 1977.

[3] D. Hamlet. Are We Testing for True Reliability? *IEEE Software*, pages 21–27, July 1992.

[4] O. J. Dahl, E. W. Dijsktra, and C. A. R. Hoare. *Structured Programming.* Academic Press, 1972.

[5] SET LABORATORIES INC. PC-METRIC User's Manual.

[6] J. VOAS, J. PAYNE, C. MICHAEL AND K. MILLER. Experimental Evidence of Sensitivity Analysis Predicting Minimum Failure Probabilities. In *Proc. of COMPASS'93.*, NIST, Washington DC, June 1993.

[7] J. VOAS, L. MORELL, AND K. MILLER. Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2):41–48, March 1991.

[8] J. VOAS AND K. MILLER. Improving the Software Development Process Using Testability Research. In *Proc. of the 3rd Int'l. Symposium on Software Reliability Engineering.*, pages 114–121, Research Triangle Park, NC, October 1992. IEEE Computer Society.

[9] L. J. MORELL. Theoretical Insights into Fault-Based Testing. *Second Workshop on Software Testing, Validation, and Analysis*, pages 45–62, July 1988.

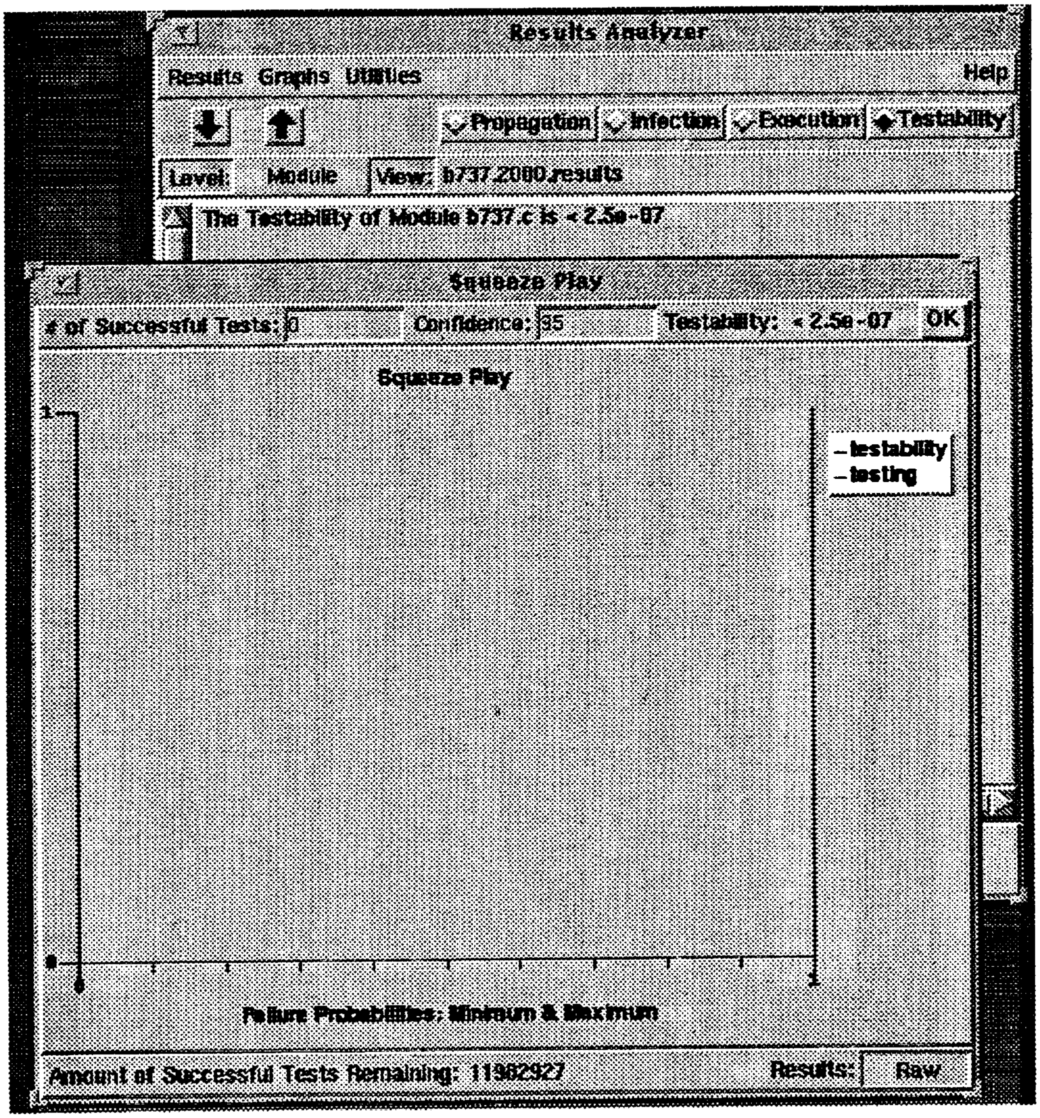
[10] L. J. MORELL. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, SE-16, August 1990.

[11] J. VOAS, K. MILLER, AND J. PAYNE. PISCES: A Tool for Predicting Software Testability. In *Proc. of the Symp. on Assessment of Quality Software Development Tools*, pages 297–309, New Orleans, LA, May 1992. IEEE Computer Society TCSE.

[12] J. VOAS, K. MILLER, AND J. PAYNE. Software Testability and Its Application to Avionics Software. In *Proc. of the 9th AIAA Computers in Aerospace*, San Diego CA, October 19-21 1993.

[13] J. VOAS. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, 18(8):717–727, August 1992.

[14] W. PENG AND D. WALLACE. Software Error Analysis. Technical Report NIST Special Publication 500-209, National Institute of Standards and Technology, Gaithersburg, MD, April 1993.



The Testability of Module b737.c is < 2.5e-07

# of Successful Tests: 0

Confidence: 95

Testability: < 2.5e-07

Amount of Successful Tests Remaining: 11982927

Results: Raw

Figure 1: Testing needed given "raw" testability score for autoland module.

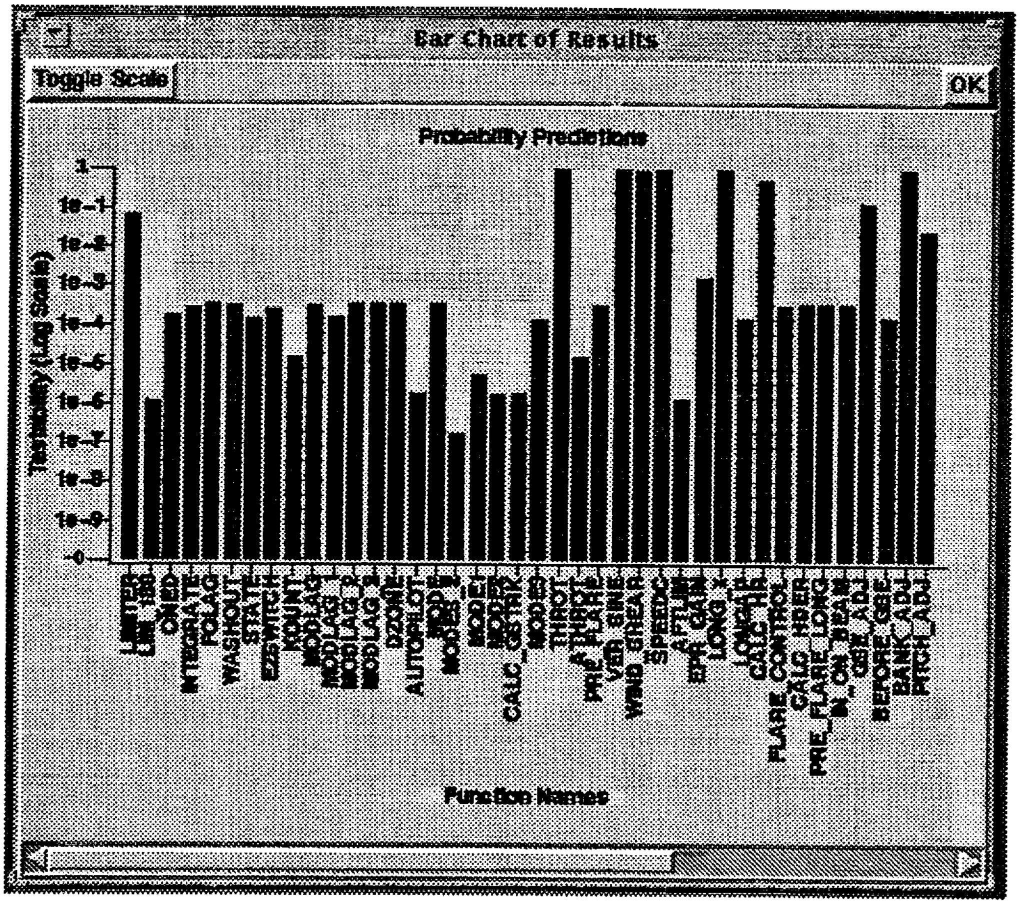


Figure 2: Function testabilities for autoland.

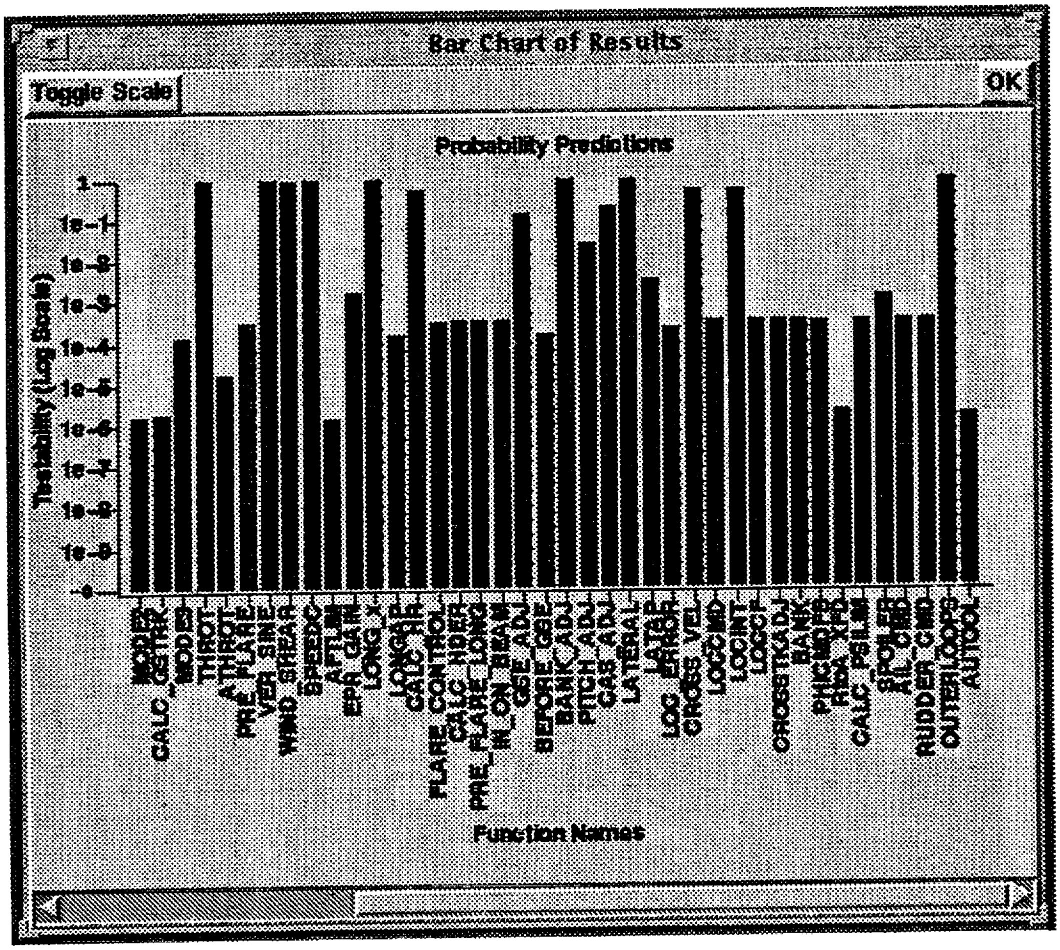
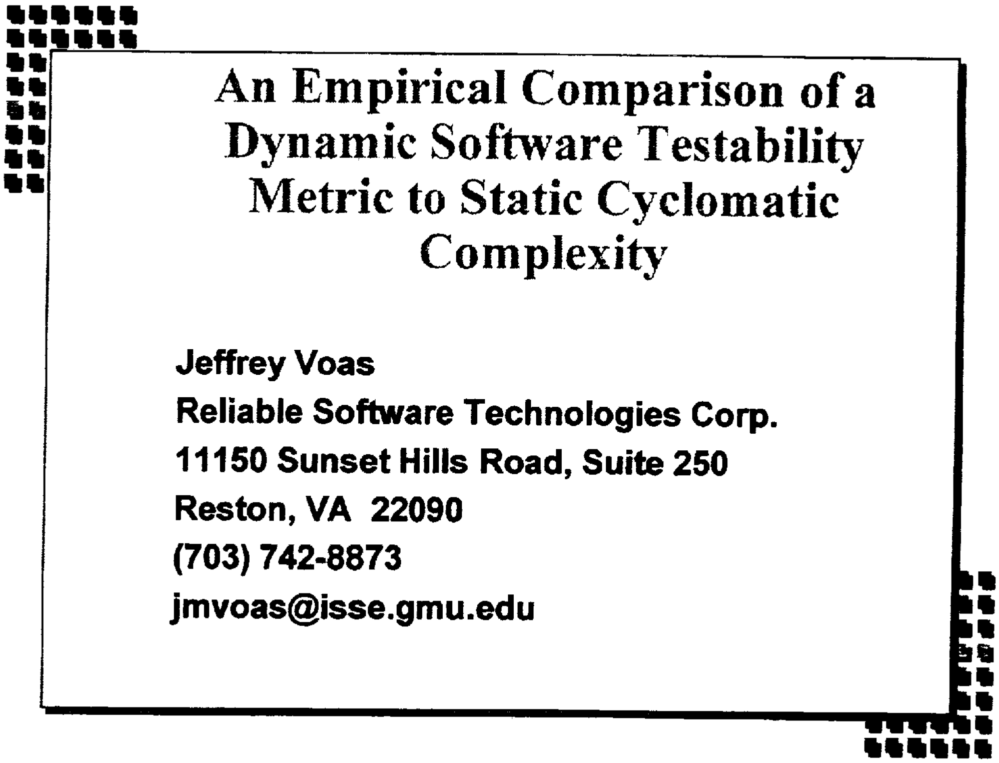
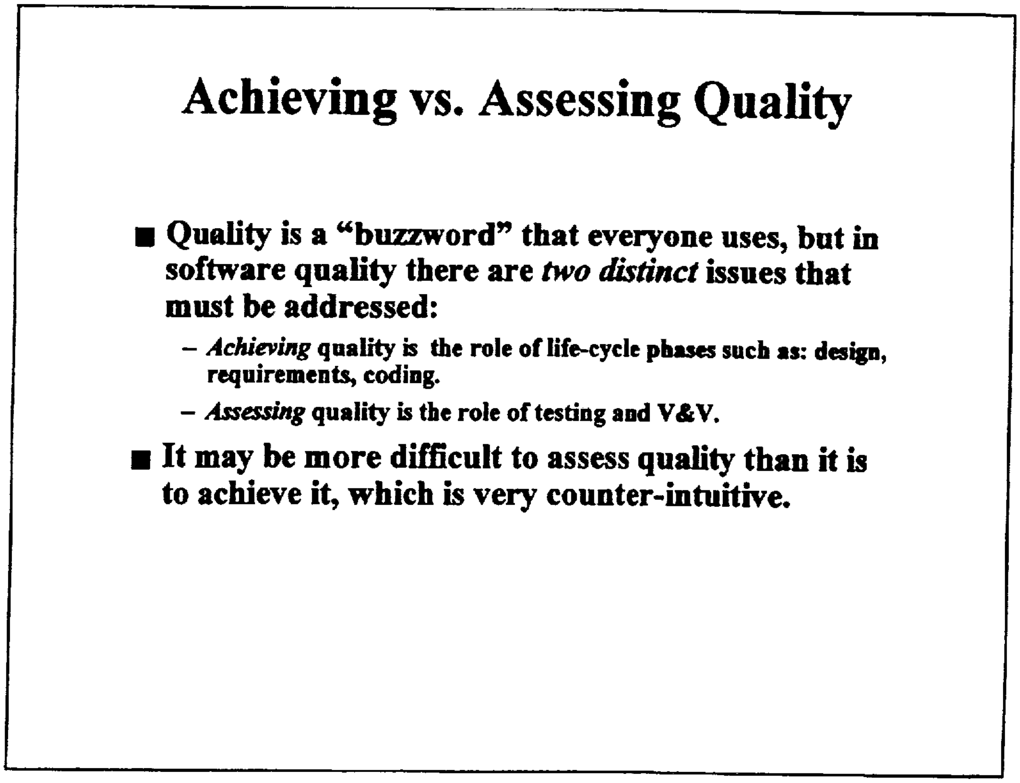


Figure 3: More function testabilities for autoland.



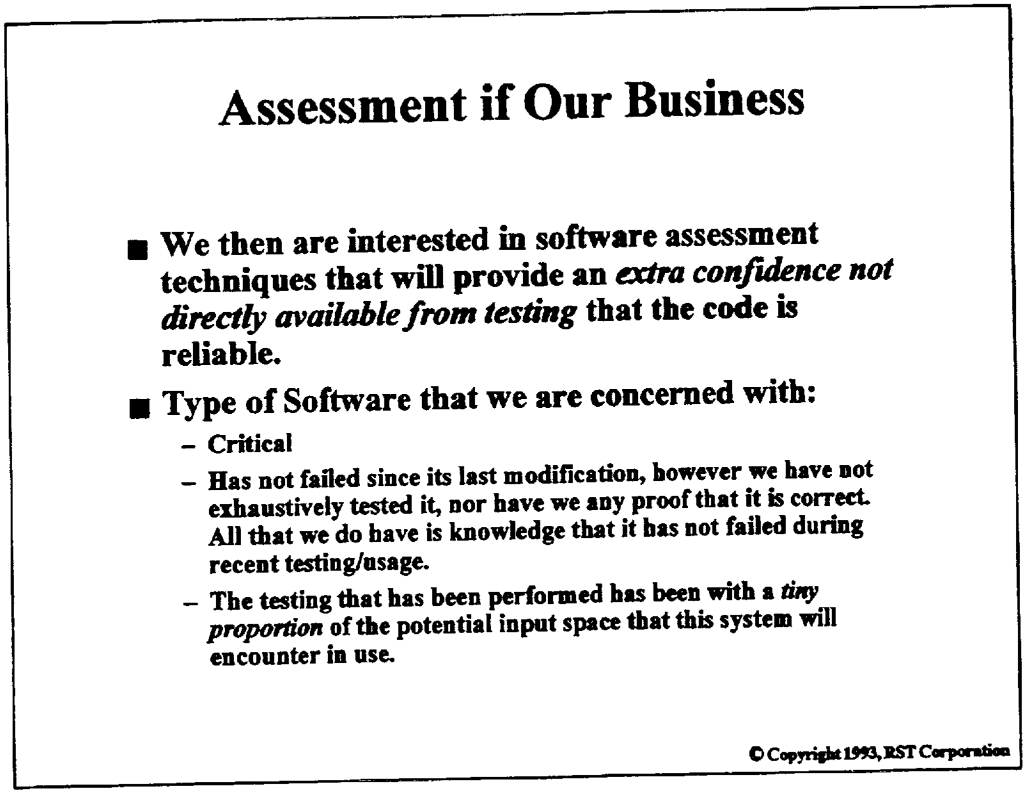
An Empirical Comparison of a Dynamic Software Testability Metric to Static Cyclomatic Complexity

Jeffrey Voas
Reliable Software Technologies Corp.
11150 Sunset Hills Road, Suite 250
Reston, VA 22090
(703) 742-8873
jmvoas@isse.gmu.edu



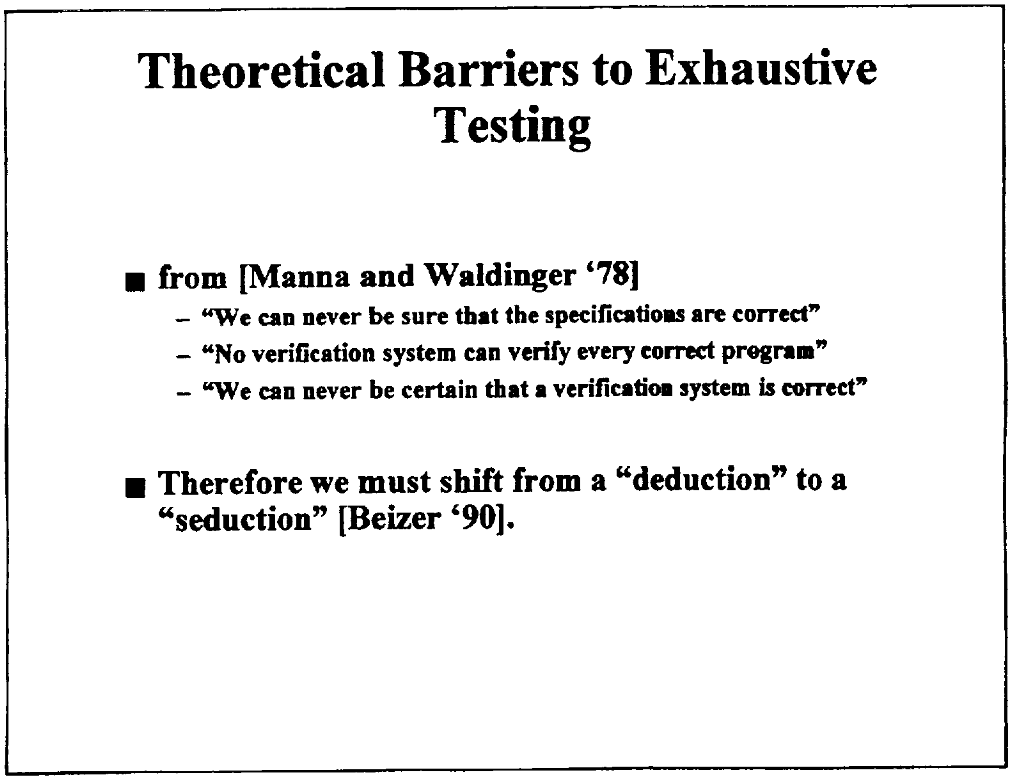
Achieving vs. Assessing Quality

- Quality is a "buzzword" that everyone uses, but in software quality there are *two distinct* issues that must be addressed:
  - *Achieving* quality is the role of life-cycle phases such as: design, requirements, coding.
  - *Assessing* quality is the role of testing and V&V.
- It may be more difficult to assess quality than it is to achieve it, which is very counter-intuitive.



Assessment if Our Business

- We then are interested in software assessment techniques that will provide an ***extra confidence not directly available from testing*** that the code is reliable.
- Type of Software that we are concerned with:
  - Critical
  - Has not failed since its last modification, however we have not exhaustively tested it, nor have we any proof that it is correct. All that we do have is knowledge that it has not failed during recent testing/usage.
  - The testing that has been performed has been with a ***tiny proportion*** of the potential input space that this system will encounter in use.



Theoretical Barriers to Exhaustive Testing

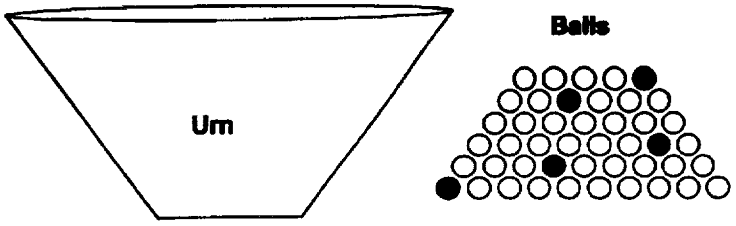
- from [Manna and Waldinger '78]
  - "We can never be sure that the specifications are correct"
  - "No verification system can verify every correct program"
  - "We can never be certain that a verification system is correct"
- Therefore we must shift from a "deduction" to a "seduction" [Beizer '90].

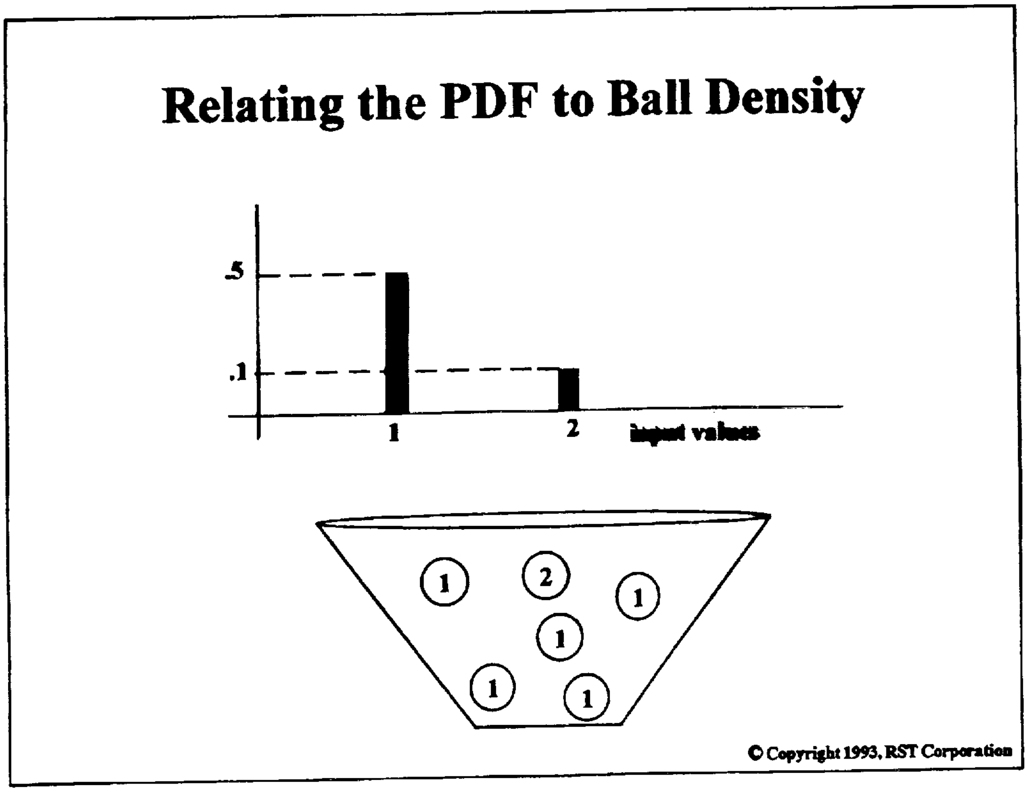
Difference between Testing and Testability

- Testing is defined with respect to some "authority" that asserts whether an output is correct.
- Testability says nothing about correctness, but rather thc likelihood of incorrect output occurring.
- This is a fundamental difference that needs to be understood.

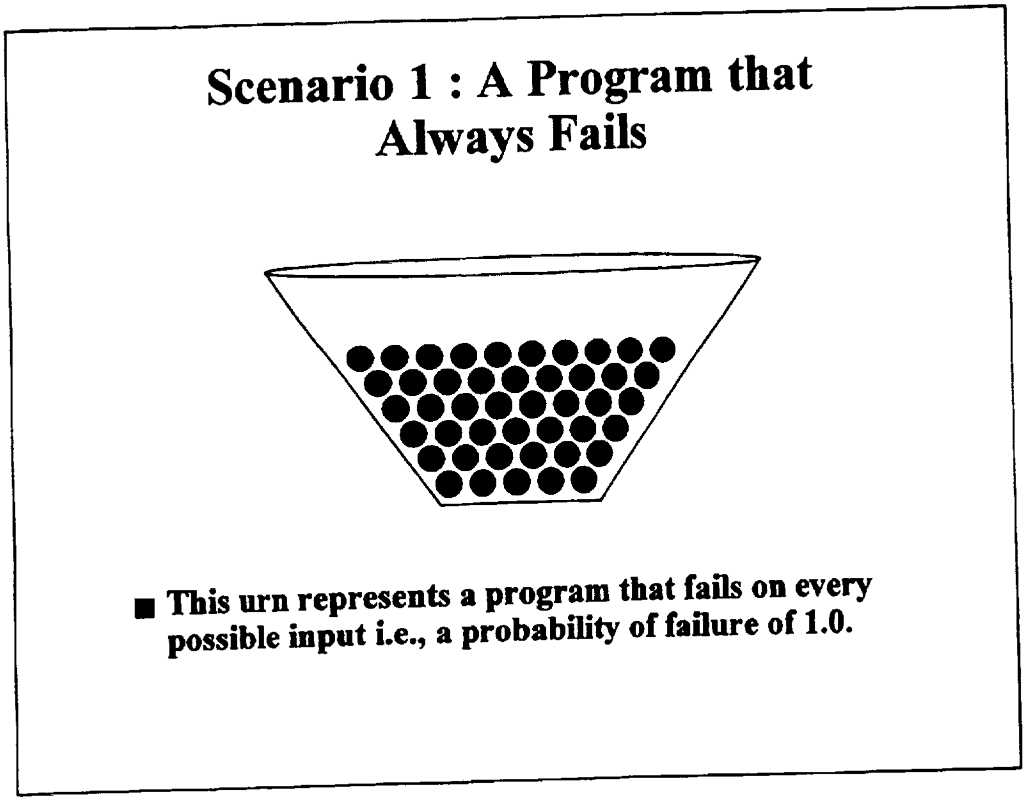
Balls and Urn

- Testing can be viewed as selecting different colored balls from an urn where:
  - Black ball = input on which program fails.
  - White ball = input on which program succeeds.
- Only when testing is exhaustive is there an "empty" urn.





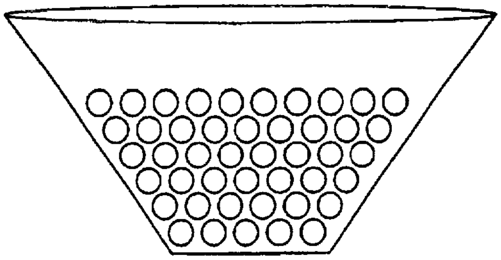
Relating the PDF to Ball Density



Scenario 1 : A Program that Always Fails

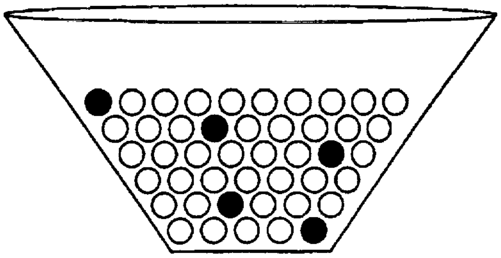
- This urn represents a program that fails on every possible input i.e., a probability of failure of 1.0.

Scenario 2 : A Correct Program



- This urn represents a program that succeeds on every possible input i.e., a probability of failure of 0.0.

Scenario 3: A Typical Program



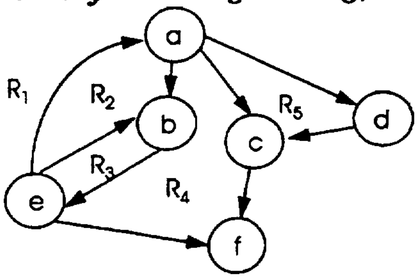
- This urn represents virtually all software in use today.

Traditional Definition of Software Testability

- <u>Definition</u>: the ability of a system to be easily and thoroughly tested, where thoroughly means that a particular coverage metric is achieved (e.g. statement coverage, branch coverage).
- Based on the weak assumption that covering code means no faults remaining.
- Example: McCabe's Cyclomatic Complexity Metric [*IEEE Transactions on Software Engineering*, 1976].

Cyclomatic complexity

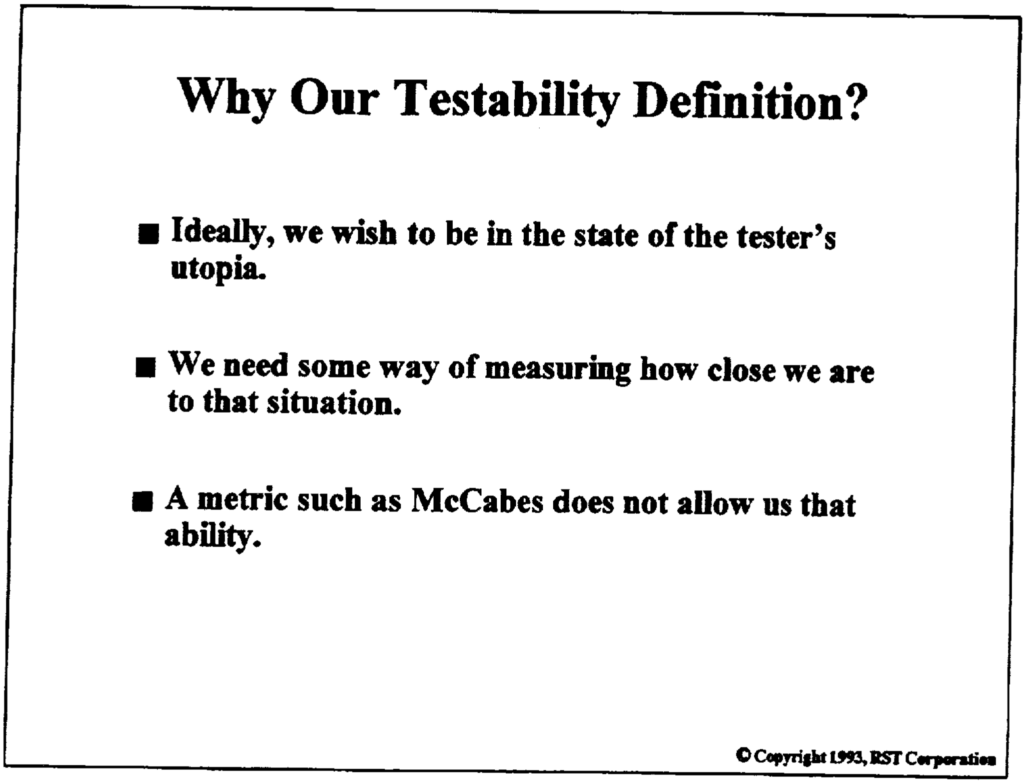
V(G) = # of regions
or
V(G) = L - N + 2P
where:
L = # of links
N = # of nodes
P = disconnected parts



V(G) = 5

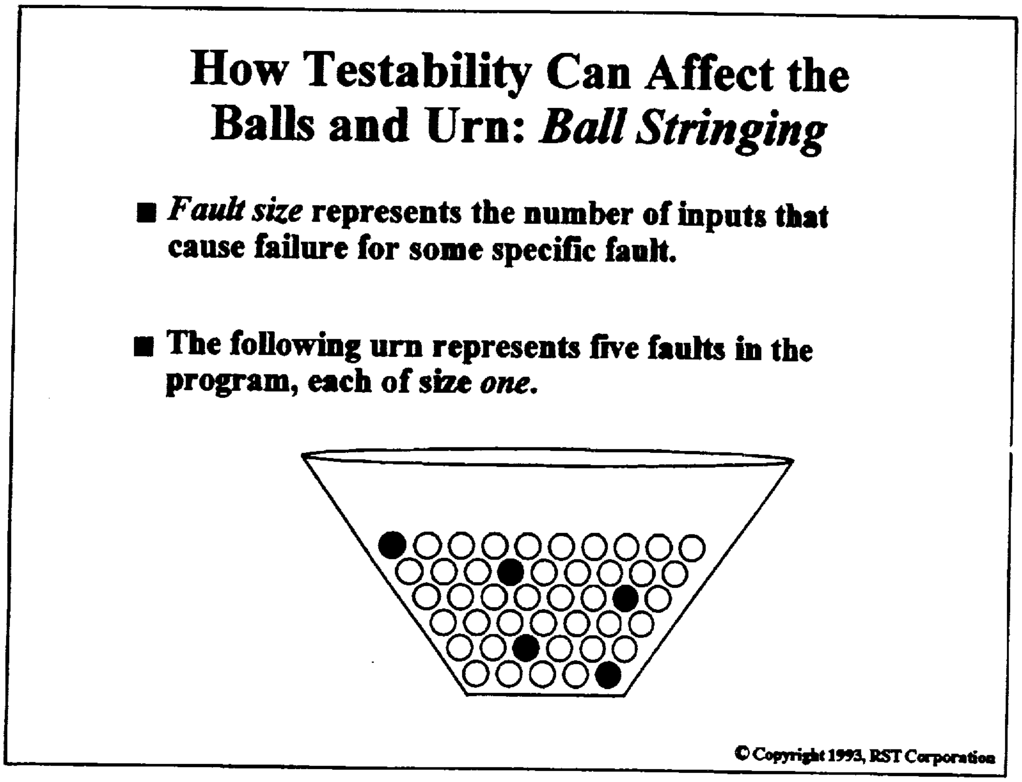
Our Definition of Software Testability

- <u>Definition</u>: a *prediction* of the probability that existing faults will be revealed during testing according to some testing scheme *D*. (*D* is some input representation)
- If there is a fault at a particular location, how likely is it you will see the fault as a failure during testing according to *D*.
- If faults are unlikely to cause failures then it is obvious that the fault will be difficult to detect during testing.



Why Our Testability Definition?

- Ideally, we wish to be in the state of the tester's utopia.
- We need some way of measuring how close we are to that situation.
- A metric such as McCabes does not allow us that ability.

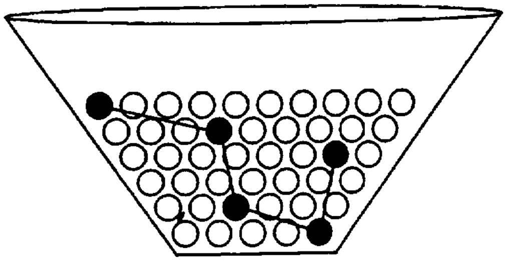


How Testability Can Affect the Balls and Urn: *Ball Stringing*

- *Fault size* represents the number of inputs that cause failure for some specific fault.
- The following urn represents five faults in the program, each of size *one.*

Ball Stringing (cont.)

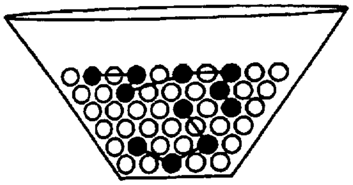
- This urn has five inputs that cause failure that are all caused by one fault in the program.
- Thus, this fault is of size *five*.



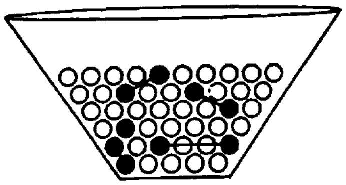
Testability and Balls and Urn

- Testability can be viewed as an assessment of how the black balls (if any) are distributed throughout the urn.
- With ***high*** testability, any string of black balls is ***long.***
- With *low* testability, any string of black balls is ***short.***

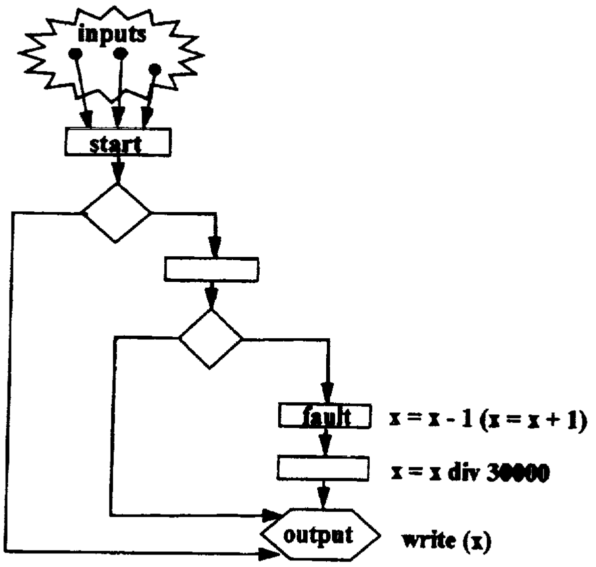
High: Shortest string is 4



Low: Longest string is 2

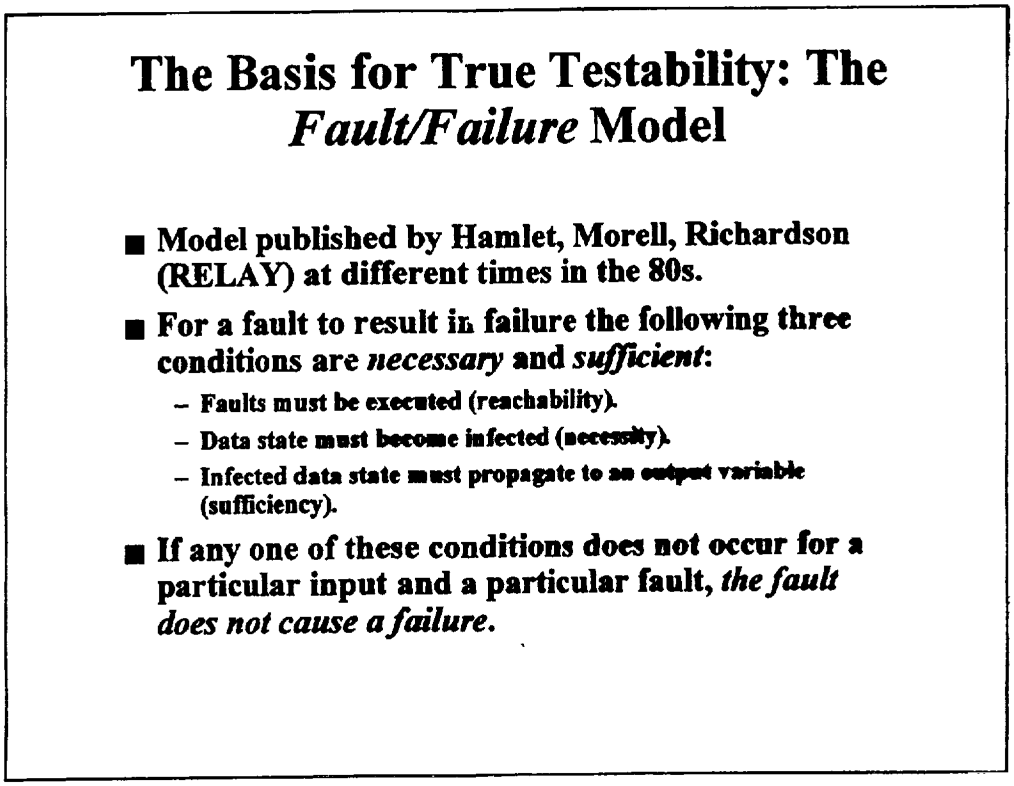


Example of Hiding Fault



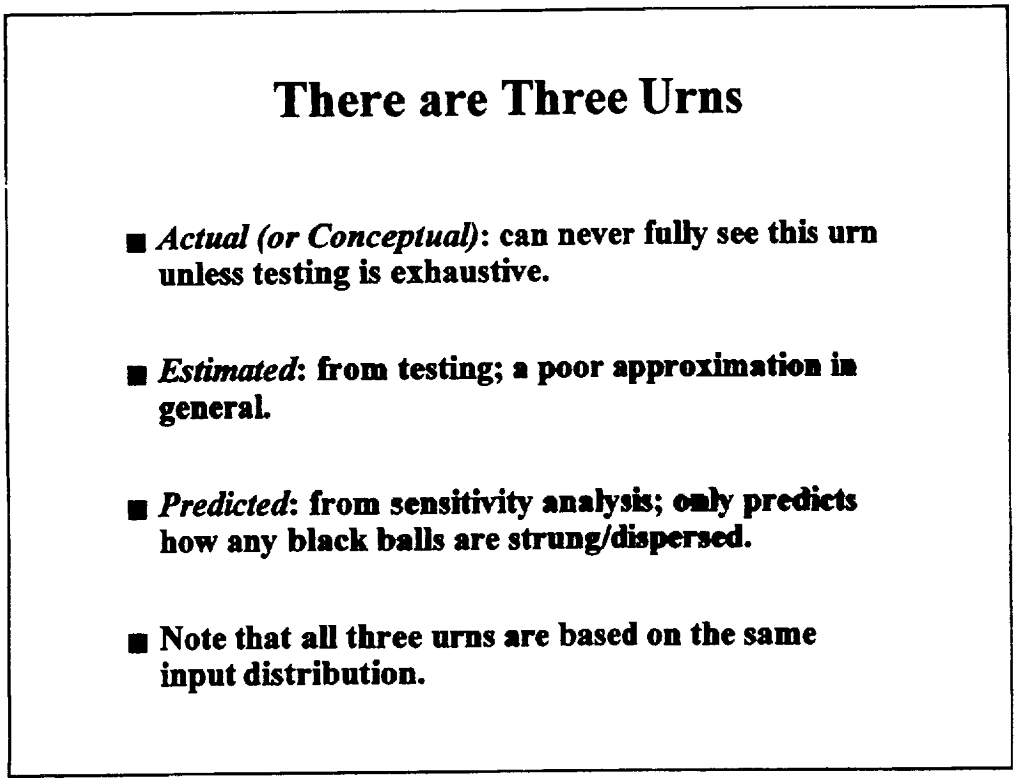
How Will We Predict the "Coloring" and "Stringing" Within the Urn?

- Use, lower level of code abstraction than McCabe.
- Use mutation analysis techniques.
- Approximate all 3 conditions of the fault/failure model.
- Increase error classes considered beyond those generally considered by fault-based/error-based techniques.
- Use dynamic code analysis instead of static code analysis.



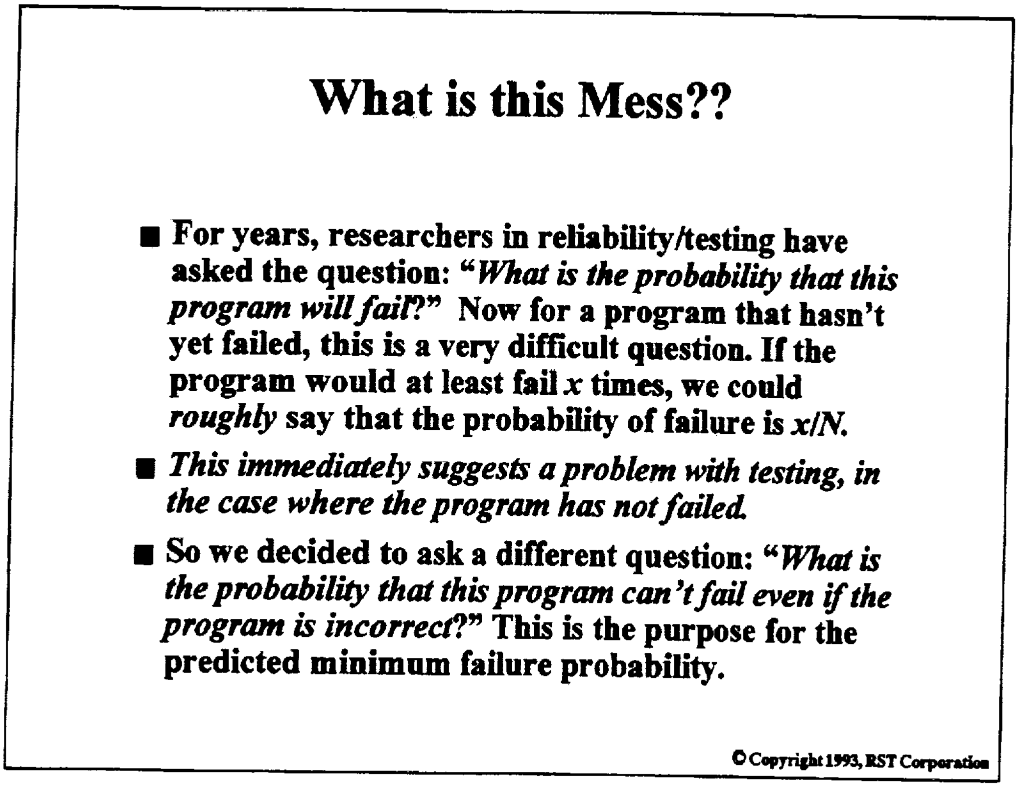
The Basis for True Testability: The *Fault/Failure* Model

- Model published by Hamlet, Morell, Richardson (RELAY) at different times in the 80s.
- For a fault to result in failure the following three conditions are *necessary* and *sufficient*:
  - Faults must be executed (reachability).
  - Data state must become infected (necessity).
  - Infected data state must propagate to an output variable (sufficiency).
- If any one of these conditions does not occur for a particular input and a particular fault, *the fault does not cause a failure.*



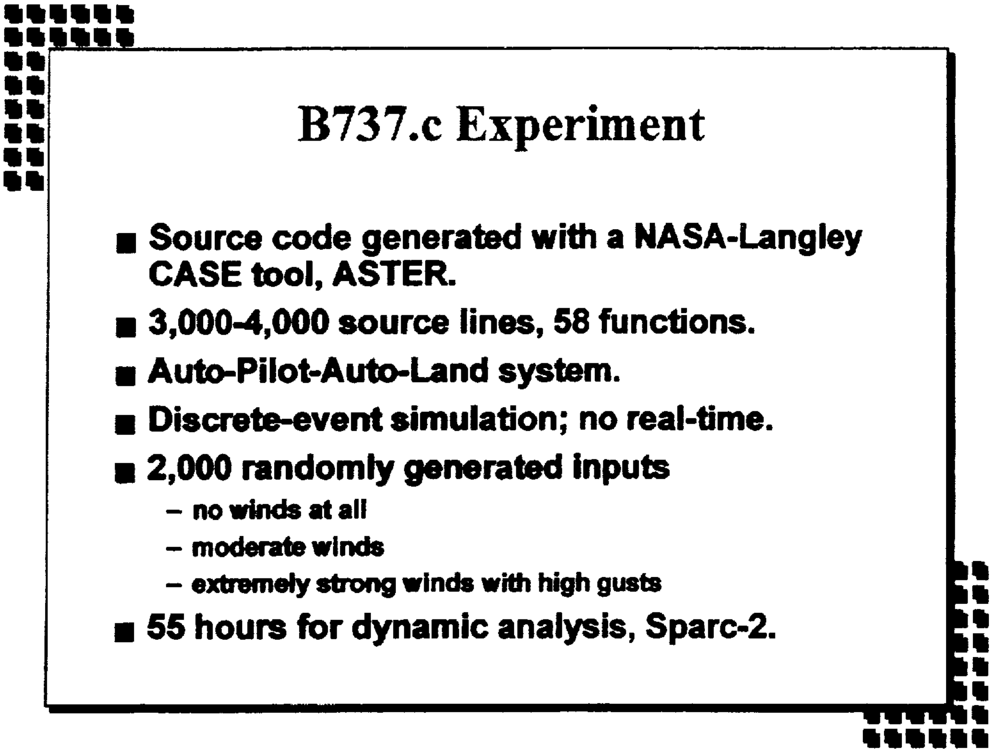
There are Three Urns

- *Actual (or Conceptual)*: can never fully see this urn unless testing is exhaustive.
- *Estimated*: from testing; a poor approximation in general.
- *Predicted*: from sensitivity analysis; only predicts how any black balls are strung/dispersed.
- Note that all three urns are based on the same input distribution.



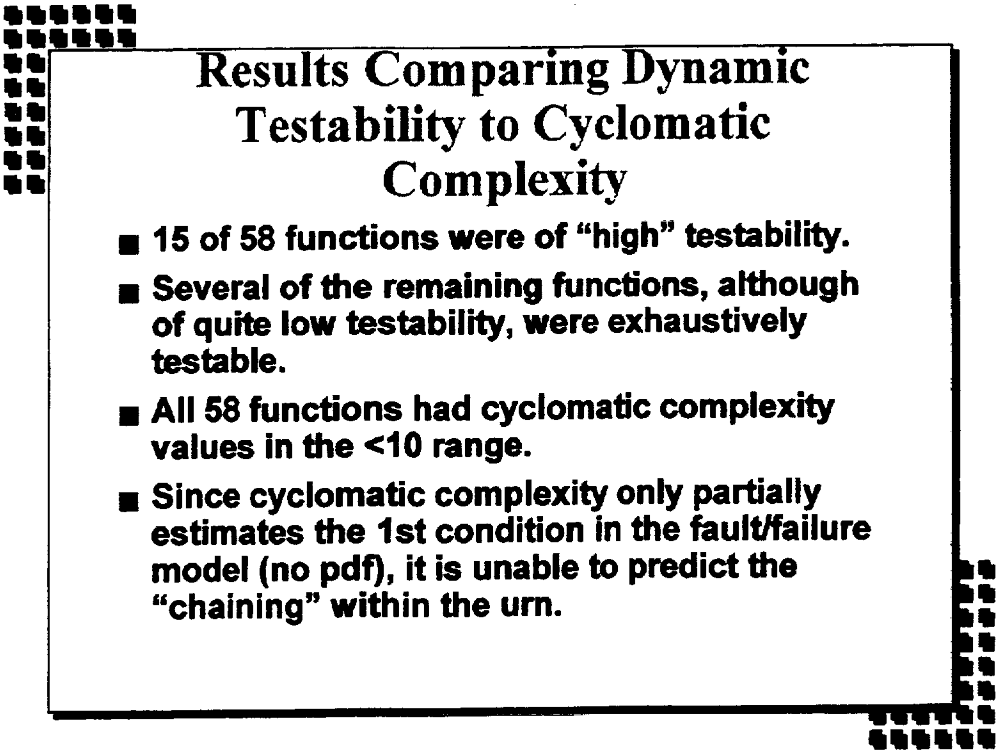
What is this Mess??

- For years, researchers in reliability/testing have asked the question: "*What is the probability that this program will fail?*" Now for a program that hasn't yet failed, this is a very difficult question. If the program would at least fail *x* times, we could *roughly* say that the probability of failure is *x/N*.
- *This immediately suggests a problem with testing, in the case where the program has not failed.*
- So we decided to ask a different question: "*What is the probability that this program can't fail even if the program is incorrect?*" This is the purpose for the predicted minimum failure probability.



B737.c Experiment

- Source code generated with a NASA-Langley CASE tool, ASTER.
- 3,000-4,000 source lines, 58 functions.
- Auto-Pilot-Auto-Land system.
- Discrete-event simulation; no real-time.
- 2,000 randomly generated inputs
  - no winds at all
  - moderate winds
  - extremely strong winds with high gusts
- 55 hours for dynamic analysis, Sparc-2.



Results Comparing Dynamic Testability to Cyclomatic Complexity

- 15 of 58 functions were of "high" testability.
- Several of the remaining functions, although of quite low testability, were exhaustively testable.
- All 58 functions had cyclomatic complexity values in the <10 range.
- Since cyclomatic complexity only partially estimates the 1st condition in the fault/failure model (no pdf), it is unable to predict the "chaining" within the urn.

SOFTWARE QUALITY: PROCESS OR PEOPLE

by

Regina Palmer
Martin Marietta Astronautics
P.O. Box 179, M/S S1008
Denver, CO 80201
(303) 977-5748
&
Modenna LaBaugh
R&FC Group
3076 So. Hurley Circle
Denver, CO 80227
(303) 986-3729

This paper will present data related to software development processes and personnel involvement from the perspective of software quality assurance. We examine eight years of data collected from six projects. Data collected varied by project but usually included defect and fault density with limited use of code metrics, schedule adherence, and budget growth information. The data are a blend of AFSCP 800-14[1] and suggested productivity measures in *Software Metrics: A Practioner's Guide to Improved Product Development.*[2] A software quality assurance database tool, SQUID,[3] was used to store and tabulate the data.

The projects represented varying degrees of programmer expertise, acquaintance with software engineering techniques, and languages including Ada, C, FORTRAN, LISP, Pascal, and Prolog. The programs evaluated were engaged in the production of simulation, R&D, mission operations, flight, or ground support software. The size of the programs ranged from small, $500K to $2 million, to large, in excess of $40 million.

Amongst the projects, we were able to track the responsiveness of different programmers to improving quality based on assessment and feedback. When quality goals and standards were established, and stressed by management, the compliance of all could be obtained. Knowledge of management expectation was especially important. This could be seen in how peers reviewed each other's work according to the tone set by their lead.

At a company such as Martin Marietta which develops software for various agencies of the government, each with its own concept of getting the job done and the related cost, flexibility of process is often sought. Such flexibility usually translates into eliminating budget for independent assessment

[1] AFSCP 800-14, Software Management Indicators, Management Quality Insights, Air Force Systems Command, 20 Jan 1987.

[2] *Software Metrics: A Practioner's Guide to Improved Product Development*, K. H. Moller and D. J. Paulish, Chapman and Hall, 1993.

[3] Software Quality Assurance Interactive Database, produced by R&FC Group.

of the quality of products and processes. It is easily argued that the engineers themselves are responsible for the quality of goods and will see to it. As soon as signs of engineering neglect appear, this course is overturned and outside evaluators play catch up on project. Such action is not cost effective, but can produce an accepted delivery. The most cost effective quality program, in our experience, is one that assesses the output of individuals early in the process then concentrates on those who show the least commitment or understanding of the quality of work expected.

In our case studies, deviation from expected output seems to occur amongst programmers who have only produced software for internal use that does not need integration or coordination with other software producers, those without a familiarization with programming standards and procedures, and those who believe rules were made for someone else. The first two are helped by a well defined process, the latter are a roadblock in any effort.

The success or problems of the process used on six programs is presented here. Each program is characterized according to lines of code effort, number of programmers, experience level of programmers, criticality of software, degree to which contract requirements or budget guided the process selected, and success of the process implementation as measured by the quality assurance effort. The projects reported range from the best of all possible worlds to the worst.

The best world is one where all (most) parties agree on the process and are committed to adherence. Next best has management in agreement with a restrained acceptance by the programming staff. The worst world has a process imposed upon other habitual methods causing rework costs to soar and pitched battles on a daily basis over budget and schedule.

Case study 1 was a model small program involving non critical ground support software written to perform on a personal computer. Data from it is included to show nominal cost of quality when the process produces the desired result with little rework.

Case study 2 involved a small project of four to seven programmers developing non critical software in C and Prolog. Response of the programmers to metrics collection was examined and its influence on their subsequent work analyzed.

Case study 3 was a large project involving a software engineering staff of 20 to 40 people using Ada to develop software for a space experiment with human interaction and support software. The performance of groups of programmers in the process defined for a full life cycle program is examined and related to varying management expectation.

Case study 4 is our worst case model, where everything is wrong and the solution requires replacement of staff. The programming languages were C and assembly.

Case study 5 was a medium sized project producing flight and ground software for a space experiment written in Ada and C. It had a well defined process and the study documents how much rework was involved related to programmers per their level of acceptance of the process imposed.

Case study 6 was a small project that was an engineering support task for one of the NASA centers. It is used to show the difference in performance of the programmers involved based on their past experience with software engineering discipline. No contract criterion was available but each was totally responsible for code to work on a particular platform.

DEFINITIONS

Productivity where reported was calculated from total software engineering hours, including support from or for systems engineering, systems testing, program management, software quality assurance, and direct support from areas like finance and planning. It represents the deliverable lines of code divided by these hours times an eight hour day.

Fault Detection is reported in a form to show what activity was used to find defects and in what phase they were found. It is also sometimes displayed to emphasize the quantity of defects found in house versus at the customer's site. Types of defects are reported if needed to explain the other data. The process of tracking discrepancies in software provides information to help improve productivity and efficiency. When problems are discovered during integration and system test, the priority of the error is examined in addition to what caused the error. Priority of errors can range from errors that make the system inoperable to errors that do not disrupt the running of a test. The following details the levels of priority used by this paper:

A - error in the code in which the software did not meet the requirements or design, an error which was a documentation error which caused the code to not meet the requirements or a code error which crashed the system making the system non operational until the error was fixed.

B - error which crashed the system but there was a work around and the system could be used.

C - error found in the code which did not interfere with the operation of the system.

D - a minor error in the code such as a typographical error in a help message.

The cause is examined to identify the reason for the error. The cause could be a requirements error, design error, coding error, hardware interface error, or a requirements change directed by the customer.

In the few examples where code metrics are used to emphasize the difference between programming groups, items found of value were the size of modules, and adherence to coding standards.

Schedule adherence was based on planned dates for major milestones versus actuals and slippage in the final delivery. A major milestone was usually a formal review or delivery. In program 3, which was cancelled before delivery and stretched out twice before then, adherence to internal schedules was used to compare team performance and responsiveness.

PROGRAM 1 was a model small program involving non critical ground support software written to perform on a personal computer. Data from it is included to show nominal cost of quality when the process produces the desired result with little rework. This program consisted of 7500 lines of mission operations software. There was a software lead, six software engineers, one systems engineer, and a test engineer. The 7500 lines of code were required to be developed and delivered in six months. The entire team was experienced in developing and testing software. The program philosophy was to use senior personnel to ensure that every task was completed on schedule.

The program began by producing a software development plan to document the process to be implemented. As the SDP was being developed, the program conducted tabletop reviews with the engineers that would implement the plan, SQA, test, systems, and the customer representative.

The tabletops were used to ensure that a process was developed that incorporated good engineering practices as well as being streamlined. Once the SDP was approved and agreed upon by everyone, the requirements were finalized. The customer was very involved assuring that the requirements were finalized in a timely manner because of the tight schedule. A requirements review was held to baseline the requirements with the customer. Design and code walkthroughs were held to ensure that the design and code implemented the requirements and that the design and code standards were adhered to. At each of the walkthroughs the software lead, SQA, test, systems, and a customer representative were present. This ensured that everyone was aware of the state of the software and agreed upon the results of the walkthroughs.

Figure 1 shows the planned versus actual schedule adherence by program 1. One week of slip in the schedule occurred during the code phase but that slip was recovered during system test allowing the program to still deliver the software on time to the customer.

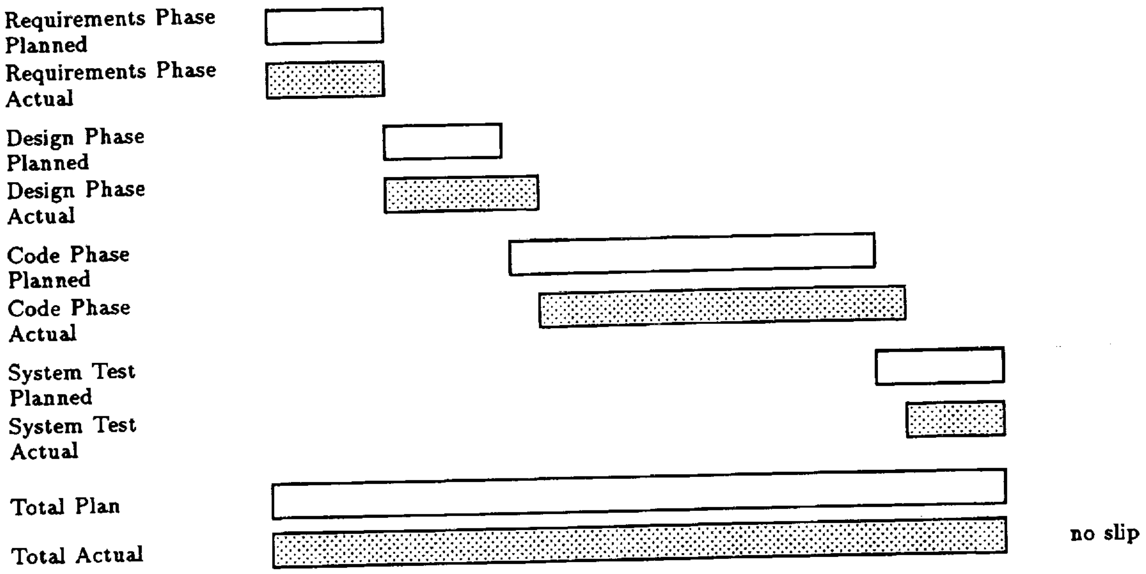


Figure 1 – Program 1 Schedule Adherence

The results of the reviews of the requirements document are shown in table 1. The requirements document improved after each review excluding the preliminary design phase which had a review cycle too short to allow incorporation of meaningful corrections.

Table 1. SRS Document Completion Index

Phase	DI Score (1.0 high)
Requirements	.67
Design (PDR)	.67
Design (CDR)	.77
Coding	.83

Figure 2 shows the breakout of the types of errors found in each phase of the life cycle. Most of the requirements errors were discovered during the requirements phase. The requirements errors found during the system test phase were due to the customer changing the requirements prior to delivery. Overall the errors found during the life of the program resulted in a fault density score of 0.8 discrepancies per 1K line of code. There were no errors found after delivery of the software to the customer site. 93% of the discrepancies were found before system test.

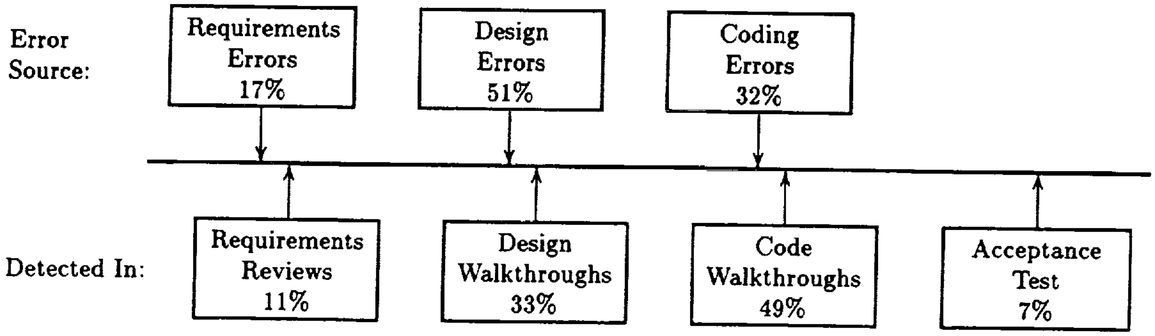


Figure 2 – Percentage of Errors Found by Software Phase

This small program was very team oriented. At the start of the program, the lead, software engineers, test, SQA, and the customer representative got together and decided that everyone would need to work together to meet the scheduled delivery date. All engineering staff were experienced in their fields and responded to action items from their peer reviews rapidly. The productivity was below expected standards but adherence to schedule and accuracy of the code were the drivers. Measured by those standards, the program was everything desired.

PROGRAM 2 was less than $2.5 million including delivery of commercial workstations used in development. Requirements and most design had been accomplished under a previous contract. The documentation produced under the previous contract included a system specification and functional description document. Effort reported was a planned two year implementation and system test with two deliverable prototypes and a final operating capability. New deliverable documentation to be produced to Air Force standards included program and database specifications, and the user's manual. Projected coding effort called for five to seven programmers, including three leads with more than two years experience at the company and the remainder being college graduates. Contract was bid with a 20 LOC per day goal for each engineer. 81KLOC were developed (70K C, 11K Prolog/LISP) and 199KLOC delivered (included legacy code from previous contract). The end product was software used in a lab environment.

The process involved informal walkthroughs involving the software lead, programmer, tester, and quality. Unit Development Folders were maintained till delivery. Programming standards described headers required for all code, commenting and self descriptive naming for variables. A guideline of less than 100 LOC per module was not enforced. A tester who was not part of the software development staff was used for Computer Software Component (CSC) integration into the prototypes and final delivery. Only test results of the CSC integration were reviewed. Discrepancy tracking was initiated at CSC integration.

Defects were measured only through the testing program. Software measures for code simplicity, self-descriptiveness, and conciseness were obtained on the C code. This was accomplished with a code reading tool that calculated the Halstead Measure,[4] branching complexity, lines of code,

[4] *Elements of Software Science*, M. Halstead, Elsevier, 1977.

commenting as a percentage of total non blank lines, and variable density. The numbers found for the code were not used as acceptance criteria. They provided a background from which to evaluate changes in the code resulting from error reports. Large changes in any score were viewed as cause to reconsider acceptability of proposed changes. Such a screening was used due to limited resources for people to review changes.

The programming standards on the project were dictated by two of the leads. They adhered to them, the third lead did not and the junior programmers did only after they became aware that lack of adherence was reported to management. Most programmers responded positively when management made metrics goals visible to them. In the first audit of code compliance to standards, two samples were taken representing code from senior programmers in sample A and less experienced programmers in sample B. 81% of sample A was above average in score, but only 59% was above average in sample B. Sample B ratings tended to be either very good or very poor with less than 20% of the modules falling in the middle. The results of the audit were distributed to the programmers and a limited amount of time was authorized for rework. The group represented in sample A reworked code that fell below the minimal acceptable level raising their mean score to 3.8 from 3.6. Sample B programmers reworked all code scoring average or below bringing up the mean score for sample B to 4.4 from 2.9. The entire sample rose to 4.1 (excellent) from 3.2 (good).

Table 2. Code Compliance Audit

Sample	Modules	Score	Rework
A	154	3.6	3.8
B	176	2.9	4.4
A + B	330	3.2	4.1

A major problem in the methodology used on this program was the lateness of finding the majority of errors. 85% of the errors in the code were not found till integration of the final deliverable though 60% existed in code baselined a year earlier. This occurred due to inexperience on the part of the integration tester and a flaw in the test philosophy of the development personnel. The tester assumed unit testing of low level functions had been performed by the developers. The software leads were more involved in code development than anticipated and did not exercise sufficient oversight of the unit test effort. Functional testing of the first baseline was not performed because it was legacy from the previous contract and assumed working because of acceptance at the customer's site.

Table 3. Discrepancies in the Baselines

Baseline	Size	Fault Density
IOC1	22KLOC	20
IOC2	33KLOC	3
FOC	70KLOC	3

After delivery of the second prototype, the test philosophy on the program changed. More unit testing was demanded before integration. Upon being told that code discrepancies were being tallied during integration testing, the programmers became very active in finding and documenting errors in the baselined code while performing unit testing prior to integration of their own code with that baseline.

The following table is in order of programming experience.

Table 4. Code Fault Density by Programmer

Programmer	Fault Density of Code
P1 - sw lead	8
P2 - sw lead	18
P3 - new graduate	10
P4 - new graduate	8
P5 - new graduate	17

P2 and P5 were reluctant to take time to test. P4 was used to do most correction of P2 code and P3 was used to correct P5 code because of the low fault density of code they wrote.

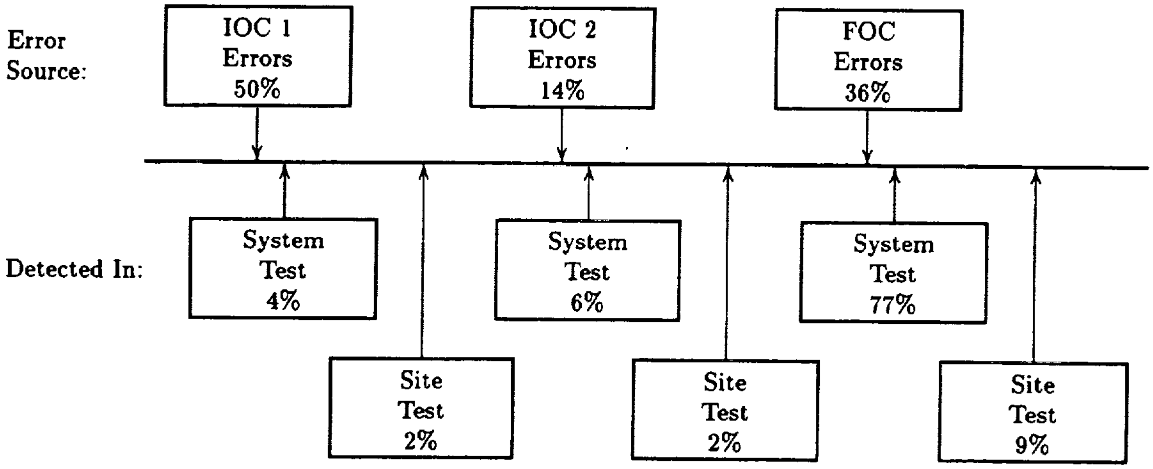


Figure 4 – Distribution of Errors in Baselines

The perceived error rate (that seen by the customer) was 1 fault per 1000 LOC. This was low in the customer's experience and the customer was pleased with the software and regularly used the prototypes from the first two deliveries.

Table 5. Error Detection Activity

Phase	Engineering Test Errors Found	Site Use Errors Found
IOC1	4%	2%
IOC2	6%	2%
FOC1	21%	8%
FOC2	56%	1%

Schedule slippage appeared after planned enhancements to old code were completed and newly developed code was nearing baseline. Up to three months before the planned delivery date for the final operating capability, the program manager was reporting the program was on schedule. Estimates made by the quality representative of test completeness projected that 90% of the errors in the code had been found. This was based on the completeness of scheduled testing by the test department and the assumption that the error rate established in the first baseline testing of 2 defects/KLOC would not grow to more than 4. Unfortunately, this estimate was in error as became apparent when attempts to verify the completeness of unit testing for the final delivery were initiated.

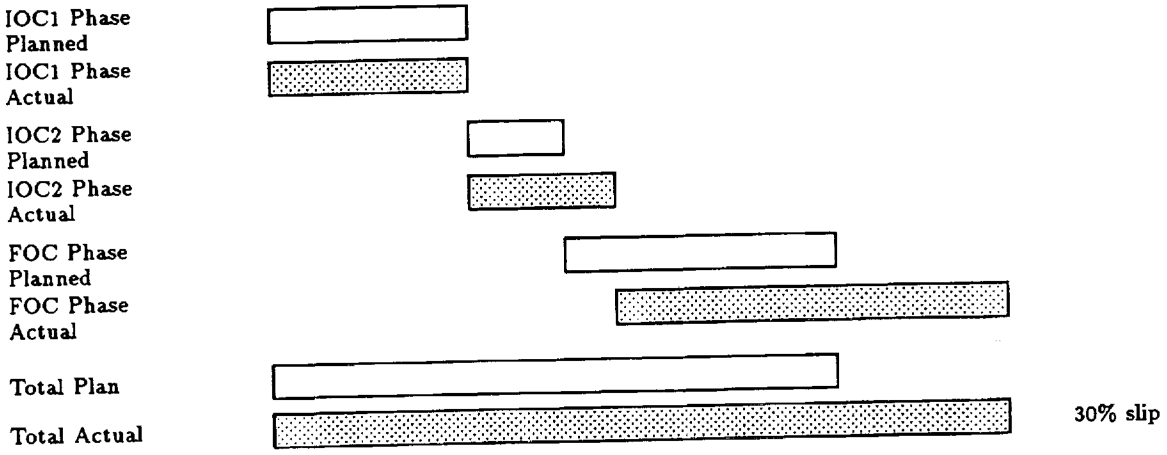


Figure 6 – Program 2 Schedule Adherence

The goal of 20 LOC per day per engineer was not met though at 16 LOC per day they did exceed the company expectation at that time for production of ground support software. General problems surfaced in relation to additional resources required to bring into compliance the code of the third senior programmer and to make functional legacy software that should have been working but was not. This effort, undertaken at the end of the development program, contributed to a significant cost overrun that ate all profit bid for the program plus additional company funds.

The planned quality assurance budget on the program was exceeded by 10% and engineering budget by 36%.

PROGRAM 3 was valued at more than $500 million including major space qualified hardware development but less than $30 million for software. Software effort was projected to include from 14 to 40 developers over three years. There were five leads with experience in the range of 5 to 15 years. 60 KLOC Flight and 50 KLOC test bed software were to be developed by mostly experienced programmers with subsequent updates for additional deliveries. 114 KLOC ground software were to be developed or flight code would be reused and modified by programmers of varying skill level. The programming language was Ada with less than 200 LOC of C used in the ground support software. SSP30000[5] was the required standard.

The process covered a full development life cycle, including formal reviews, massive documentation, independent test, and software system engineering at the start of contract conducted by a group separate from software development. A programming and procedures standard covered coding practices, defined the walkthrough process, software development folder contents, baseline activities, and unit and informal CSC integration testing. Independent tests at the top level CSC (TLCSC),

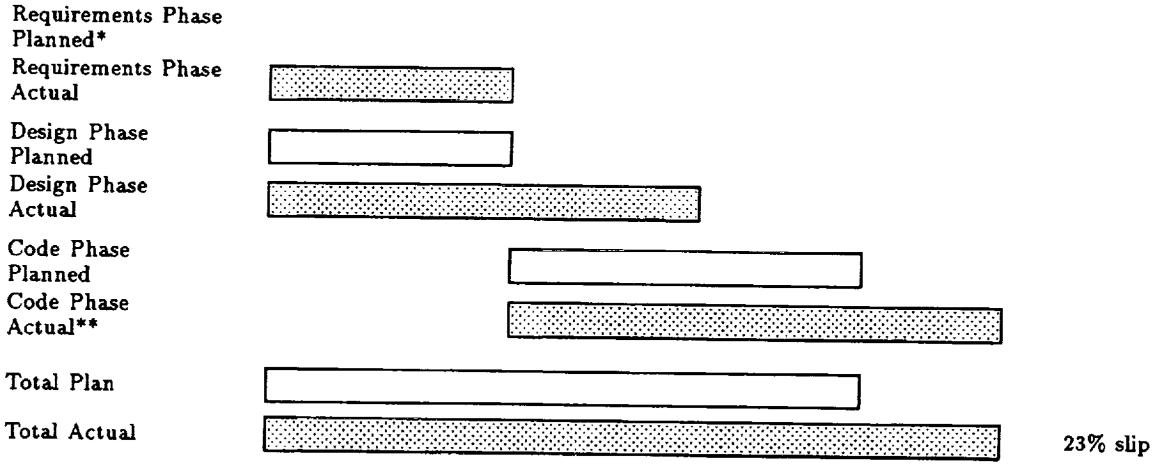
[5] SSP30000, Space Station Program Definition and Requirements, Section 2 Program Management Requirements.

CSCI and system level were to be conducted. At the outset of the program, a methodology with heavy involvement from groups separate from software development including system engineering, test, quality assurance and system safety was instituted. The on site customer representative became involved during the critical design phase.

Software documentation changes were controlled by the software manager after PDR through a software review board (SRB). Modification to the requirements and design documents was through redlines submitted to the SRB that were reviewed by the leads for all the CSCIs, software quality, software test, and system engineering.

Defect density was measured using the numbers of software change requests processed by the SRB and the action items generated by reviewers of walkthrough packages. An evaluation program run by software quality furnished a rating on the documentation and code produced which yielded a Document Completion Index value by phase.

The program was plagued by poor definition of requirements. It was a continuation of a previous Phase B study that supposedly brought the product to a PDR level. On the subsequent Phase C/D contract, more stringent requirements for process and products were imposed which necessitated regeneration of documentation thought to have been completed in Phase B, and presentation of a software requirements review (SRR). This had not been anticipated and directly impacted the CDR schedule.



* none planned

** program cancellation

Figure 7 – Program 3 Schedule Adherence

The increase in overall schedule included stretch out of the program. The slip due to requirements definition accounted for all the slip in the design phase and a third of the overall slip in the program through coding. The rest was a program replan dictated by funding constraints for the program.

Even with the three months of additional requirements definition, the requirements document did not improve. The following table shows the progression of Document Completion Index for the three major documents produced by the software group for the flight CSCI. The SRS went through 10 iterations before the coding phase but did not get 50% of the available points.

Table 6. Document Completion Index

Phase	SRS	SDD	STP	Project
PDR	.17	.55	.50	.40
CDR	.36	.74	.61	.57
Coding	.64	.74	.61	.66

The program was terminated due to loss of funding half way into the flight software development cycle. The above table shows a steady progression but clearly indicates that the SRS did not meet the standards required. Its improvement for CDR was obtained by diverting design personnel from the software area to support system engineering in rewriting the specification. This still did not overcome the reluctance of the document authors to specify testable requirements. The rise in its rating shown in the last score it received in review was obtained through the cumulative effect of SRB actions to remove implementation detail and replace with testable requirements.

The program was terminated before system test but the following chart shows the distribution of known errors and what activities were used to find them.

A large number of requirement changes were generated by the customer and hardware designers after the software design was baselined. Of the changes directed against requirements, 32% were change summaries from the hardware subsystems. Software on this project responded to requirement change with software fixes, since it seldom could demonstrate that a hardware change for a problem would be better than a software fix. Better on this project always meant cheaper or quicker.

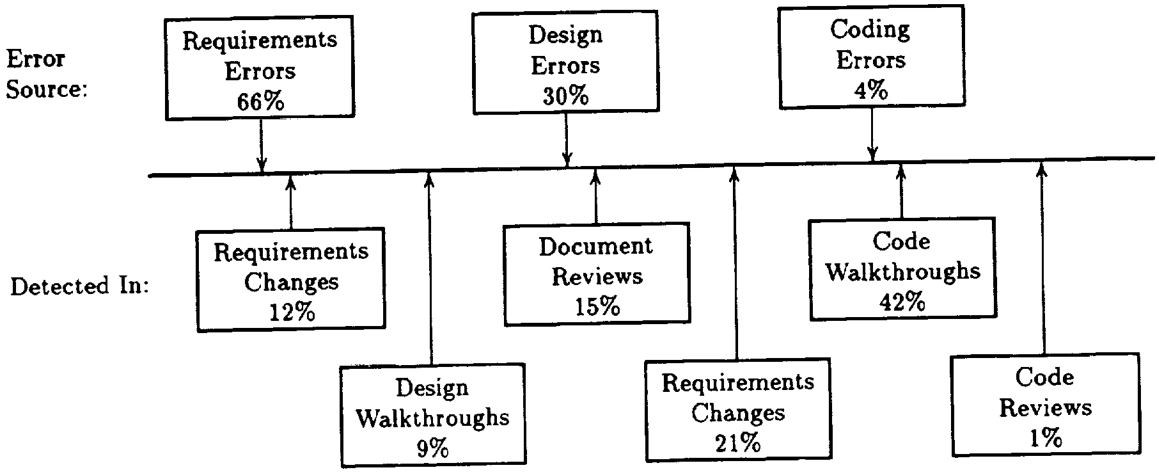


Figure 8 – Fault Detection

There was a noticeable difference in the attitude of the groups producing the four CSCIs. The three ground CSCI teams were reluctant to respond to action items and were lead by personnel who believed completion of code was the number one priority and if it worked all else would be forgiven. This corresponds to a black box mentality, i.e., the user should be happy with the result and not want to know what's inside. This is contrary to currently established specifications for software development.

Table 7. Action Item Response Time

CSCI Name	Number of Items	Average Response Time (days)
FLIGHT	24	33
EGSE	13	46
TRAINER	5	60
SIMULATOR	10	83

The items in the table, while not of top priority, were still non compliances to the contract which required correction.

The ground teams also were more reluctant to meet internally established dates for review of their work by peer groups.

Table 8. Code Schedule Adherence

CSCI NAME	W/T Dates Missed	Average Slip (days)	B/L Dates Missed	Average Slip (days)
FLIGHT	10%	19	23%	15
EGSE	26%	28	43%	21
TRAINER	14%	7	0%	0
SIMULATOR	18%	28	18%	14

At termination of the contract the quality assurance budget for evaluations was 173% over plan. The constant re-review of non compliant documentation had consumed 50% of total planned quality budget before the majority of code evaluations and testing were approached.

PROGRAM 4 is our worst case model, where everything seems wrong and the solution required replacement of staff. The programming languages were C and assembly. This program consisted of 10K of flight (RAM) software, 0.7K of flight (PROM) software, and 20K of Ground Support Equipment (GSE) software. There were four software engineers and one lead. The value of the program was approximately $22 million.

The software had a Software Development Plan (SDP) that documented the process that the engineers were to follow. This SDP was the model for programs at Martin Marietta and met the minimum standards. It consisted of the requirement for informal walkthroughs for requirements, design, and code. Unit Development Folders (UDF) were to be generated for both the flight and ground software during the requirements phase and updated with design, code, test cases and results, and problem reports. Design and coding standards were identified as well as standards for testing the software (i.e., unit, CSC Integration, CSCI testing). The program had a goal of 100 lines of code per module as part of the coding standards. Formal reviews were held for the system and software and consisted of a Preliminary Design Review (PDR) and Critical Design Review (CDR). There was a separate Acceptance Review for the software.

The software lead that started this effort was not an experienced software engineer, had no previous management position or training in software discipline. The lead was a hardware person that had done some analysis/simulation software in a lab environment and had never worked a deliverable software program. In an attempt to save money the SQA effort for this program was initiated after the beginning of the code and unit test phase though the original proposal had called for an assurance effort from contract start. Although the SDP required that walkthroughs be held on the software, none were conducted.

There were three builds of the flight software and then the delivery to the customer site. The testing consisted of unit testing, CSC integration, and CSCI testing performed by the software engineers. Discrepancy tracking was initiated just prior to system level test. A requirement for formal testing with Quality was not levied until build 3 of the software. The ground software consisted of two formal builds and delivery of the software to the customer. Parts of the ground software were baselined as test software since the hardware needed software for test.

Figure 9 shows the schedule adherence for program 4. The program had a 14% slippage in schedule that began in the requirements phase. One of the causes for the slippage in schedule was that the program was placed on hiatus for two years in which no work was done. After those two years the program restarted but the personnel that originally worked the program were no longer available and the program had to use time to restaff and come up to speed. Another problem that caused the slippage was the unknown state of the requirements. The requirements were continually being changed by the program and the customer. Since there were no firm requirements, the design was not baselined before coding and the code continually changed.

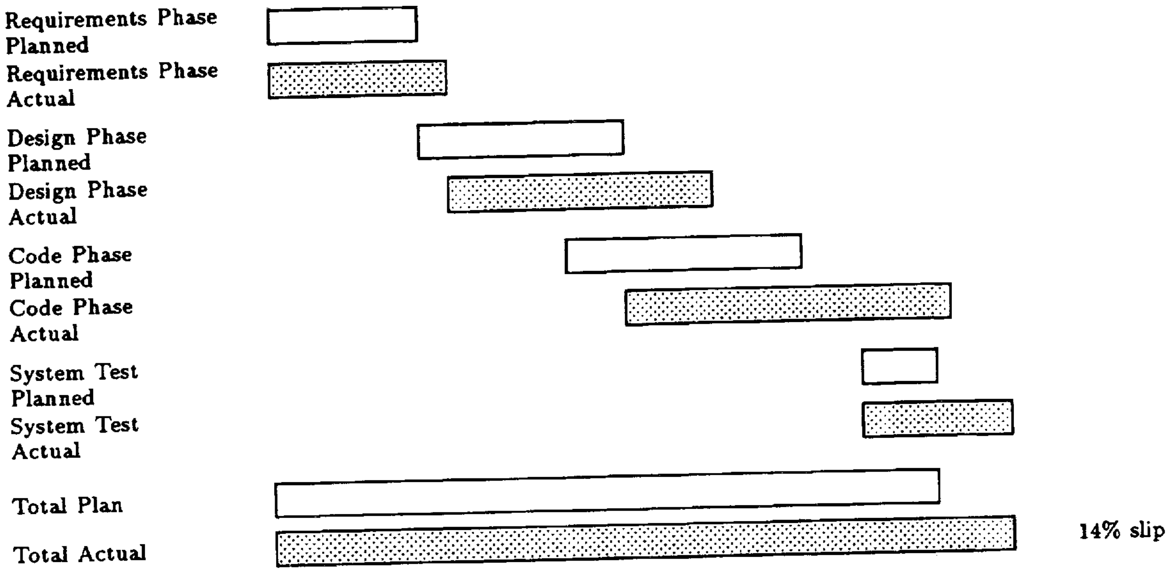


Figure 9 – Program 4 Schedule Adherence

Table 9 shows the various builds of the flight RAM software with the total lines of code for each build with the total number of modules and the average percent of comments for the CSCI. The increase in the fault density score during build 2 was a result of replacement of one of the engineers with a programmer who exercised greater test discipline.

Table 9. *Flight RAM Software*

	Total LOC	Average Percent Comments	Average Size Of Modules	Total Number Of Modules	Fault Density
Build 1	4665	19.11%	52	90	1.70
Build 2	7835	16.22%	61	128	6.15
Build 3	9459	19.42%	66	143	2.42
Delivery	9538	19.43%	67	142	0.84

Table 10 shows the various builds of the flight PROM software with the total lines of code for each build with the total number of modules and the average percent of comments for the CSCI for the C code. The large increase in the fault density score was directly related to the replacement of the PROM software engineer with a programmer who exercised greater test discipline.

Table 10. *Flight PROM Software*

	Total LOC	Average Percent Comments	Average Size Of Modules	Total Number Of Modules	Fault Density
Build 1	740	12.08%	49	15	10.81
Build 2	742	12.08%	49	15	32.35
Build 3	692	12.04%	46	15	5.78
Delivery	752	12.04%	50	15	0

Table 11 shows the various builds of the GSE software with the total lines of code for each build with the total number of modules and the average percent of comments for the CSCI. For build 1, of the 380 modules 18% of the modules had no comments at all; for build 2, 17% of the modules had no comments; and for the delivery, 18% of the modules had no comments. Only two of the files with no comments were changed after the initial baseline

Table 11. *GSE Software*

	Total LOC	Average Percent Comments	Average Size Of Modules	Total Number Of Modules	Fault Density
Build 1	18,624	8.1%	49	380	1.23
Build 2	19,317	7.9%	49	392	0.10
Delivery	19,002	7.8%	48	398	0

After start of system test, there were 18% priority *A* discrepancies, 23% priority *B* discrepancies, 53% priority *C* discrepancies, and 6% priority *D* discrepancies. Since the testing prior to baseline relied on the software engineers, a large number of high priority discrepancies show the lack of rigor used in unit testing. The number of priority *A* and *B* discrepancies found during system test could indicate insufficient time for the engineer to test the code in sufficient detail before turning over modified code, or it could indicate insufficient testing of the software prior to software and hardware integration.

Figure 10 shows the types of errors found throughout the program. There were several types of errors found. There were Coding Errors, Design Errors, Requirements Errors, Hardware Errors that resulted in the software being changed, and Requirements Changes. Coding Errors represented 69% of all errors Even after the start of system test, 2% of the errors were in design or requirements.

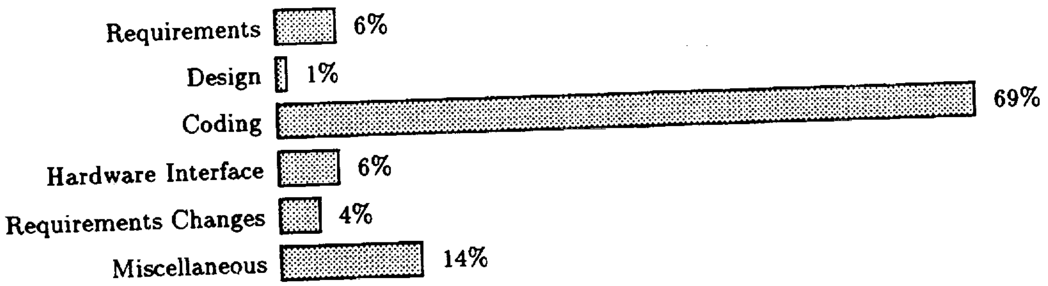


Figure 10 – Percent Errors by Type of Discrepancy

The rapid rise in Figure 11 from build 2 to build 3 is directly related to the independent test program instituted at this time. The original requirement for testing was that the software engineers perform the testing to their own comfort level. During various reviews and evaluations it was discovered by the new software lead that the level of testing was insufficient. Software had not been retested after modifications. The new software lead recommended to the customer that a more rigorous test program be implemented on the software under the cognizance of Quality. The customer agreed to this and the large number of discrepancies after build 2 is shown in the following figure.

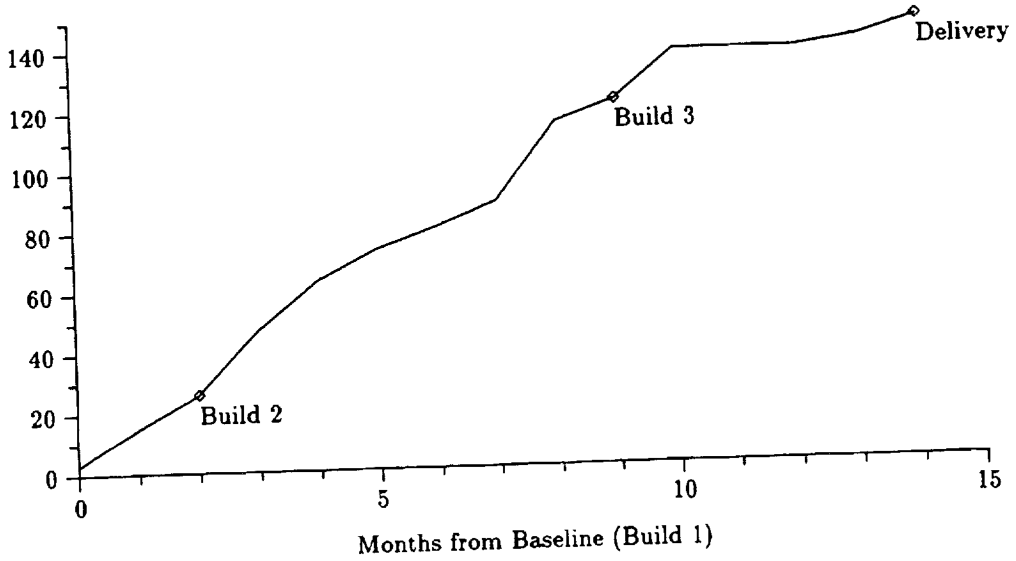


Figure 11 – Cumulative Flight Software Discrepancies

After delivery of the Flight RAM software, eight errors were discovered in the software. Of these errors one was a hardware error that resulted in the software being changed, one was a requirements error, and the rest were coding errors. Nine of the errors in Build 3 were a result of enhancements to the software requested by the customer. One of the causes for the errors in the flight software, was that the breadboard hardware used to simulate the flight hardware and used for testing the flight software had several differences between it and the flight hardware, which resulted in the software behaving differently between the two. Once the breadboard was changed to match the flight hardware, the software was corrected to operate on the flight hardware. During the phase of Build 3 to Delivery of the flight PROM software, 18 errors were found in the software. These were coding errors with the exception of one requirements change from the customer. Four of the errors were customer requested enhancements to the software. The fault density of the flight software after delivery to the customer site was 0.78.

Table 12 shows the number of discrepancies for the GSE software by build. The lack of requirements and design errors can be directly attributed to the fact that the requirements and design were changed to match the software as a result of the software engineer not adhering to the requirements or design documents. After delivery of the GSE software, no errors were found at the customer site.

Table 12. Types of GSE Software Discrepancies by Build

Build	Coding Errors	H/W Interface Errors
Test Software	17	0
Build 1	20	1
Build 2	5	0

The software development effort resulted in an average of 12 lines of code being developed per day. There was an increase of 53% of software engineering hours from the original proposal submittal and an increase of 47% of SQA hours. The 53% increase in the software development effort was directly related to the fact that several of the software engineers and the software lead were replaced nine months prior to delivery of the software. The software did not meet schedule, the software did not work with the hardware, and none of the documentation met the documentation standards or matched the software. The problems with the engineers that were replaced entailed not testing the software sufficiently or not testing the software at all. The software lead, not having worked a deliverable software program, did not enforce the SDP and ensure that the requirements were met or that the software was tested prior to integrating the software with the hardware. The software lead was replaced as well as two of the engineers, one ground and one flight. There was an overall attitude between these three that their software was perfect, therefore there could not possibly be any errors in the software and so it did not need to be tested.

The ground software engineer exhibited a lack of regard for the established process. The ground software that was developed did not meet requirements or design and the documentation was changed to match what the engineer had done after the fact. This engineer was replaced when the new software lead found out that the software written did not meet requirements or design and that the software had not been tested because the engineer did not think the ground software was important enough to take time to test. The lack of discrepancies in the GSE software is due to the requirements and design being changed to match what the software engineer had done. The customer was apprised of the situation and agreed to changing the documentation to match the software, and waived the unit and CSC integration testing of the software to prevent a schedule impact.

The PROM software engineer had the attitude that the software did not need to be retested even if the software had been changed by more than 50%. This software engineer was replaced and the change in the fault density shows the significance of the new test program. The fault density of the PROM software went from 10.81 to 32.35 and then back down to 5.78 with the new software engineer.

PROGRAM 5 was a medium sized project producing flight and ground software for a space experiment written in Ada and C. It had a well defined process and the study documents how much rework was involved related to programmers per their level of acceptance of the process imposed.

This program consists of 13K of Flight software and 18K of GSE software. The value of the program is approximately $20 million. The flight software is written in Ada and GSE in C. There are two leads and five programmers. The software lead has 10 years experience of software development, has not had a software lead position before but has developed deliverable software and knows the process and the importance of defining requirements and testing the software sufficiently. The software engineers that are used on the program have experience in software development but three of those engineers were replaced on program 4.

The software lead has established the rules for developing the software on this program and is ensuring that the software is developed and tested sufficiently. The software lead from program 4 has been given a position of testing the flight software on the breadboard unit and it has not been determined if the testing is sufficient. The software was required to be developed to European Space Agency (ESA) standard PSS-05-0[6] and ESA PSS-01-21.[7]

The program has an SDP which documents the process for developing the software and documentation. The SDP requires informal walkthrough for requirements, design, and code as well as formal reviews, (i.e., SRR, PDR, CDR). Software Development Folders (SDF) were required and initiated during the design phase. These contain the requirements, design, code and unit test cases. SQA's involvement with this program started at Authority to Proceed (ATP) and has continued throughout the program life cycle.

SQA participates in the various walkthroughs and reviews the requirements for traceability, testability, completeness, consistency, correctness, and understandability. SQA reviews the design for traceability of requirements, conformance to contractual requirements, compliance with design standards, completeness, correctness , understandability, and consistency. SQA reviews the code for compliance to coding standards, implementation of the design into the code, traceability of requirements, completeness, correctness, documentation of the code (i.e., comments), and consistency.

Figure 14 shows the breakout of the schedule for program 5 and the 17% slip in schedule. A slip in the schedule started in the design phase as a result of customer changes and impacted the Preliminary Design review date but was mostly made up in the Critical Design phase. The coding effort has also experienced a slip due to changes in design and additional requirement changes. This program will soon enter system test.

[6] ESA PSS-05-0 ESA Software Engineering Standards.

[7] ESA PSS-01-21 Software Product Assurance Requirements for ESA Space Systems.

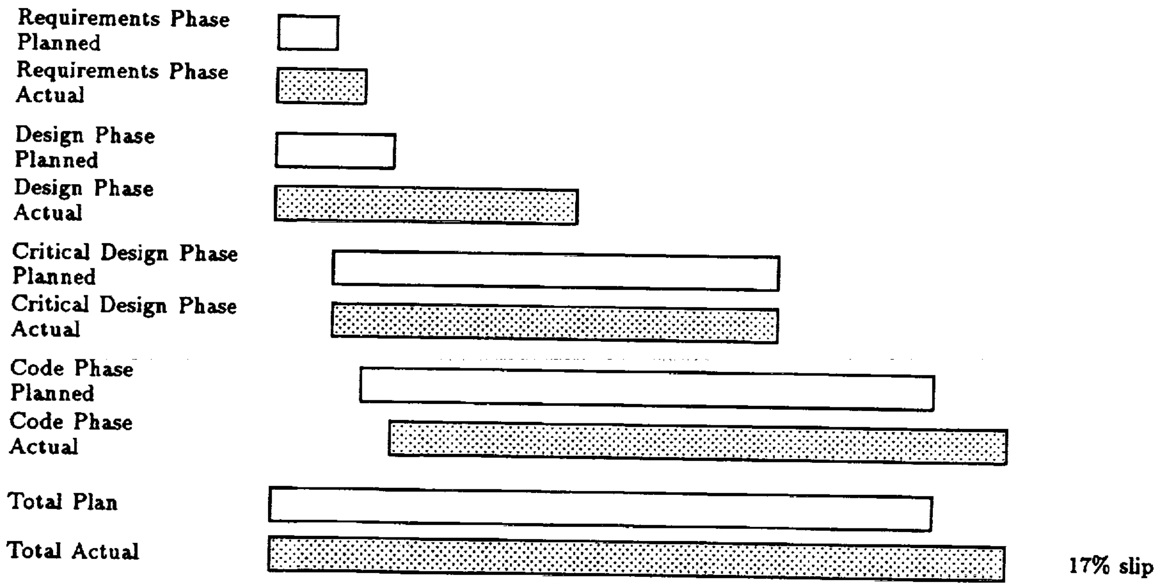


Figure 14 – Program 5 Schedule Adherence

As shown in Figure 15, the majority of the requirements errors to date were found during the requirements phase.

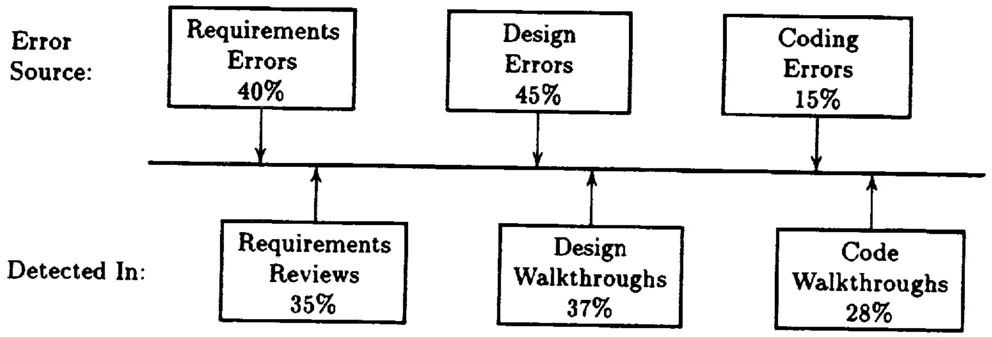


Figure 15 – Percentage of Errors Found by Phase

So far, this program seems to be on the right track. The only question mark is the fact that three people who demonstrated problems in adherence to process on program 4 are on this program. As shown in the following table, the average time to close action items of one of those inherited programmers is double the others.

Table 13. Programmer Responsiveness

Programmer	W/T Action Items	LOC	Average Time To Close
P1	49	9325	4 days
P2	31	1190	2 days
P3	70	1449	2 days
P4	45	3227	2 days
P5	10	1120	2 days

One flight programmer from program 4 seems to have learned the lesson from being replaced and has exhibited a different attitude throughout this program. The engineer has placed a great deal of emphasis on retesting software that has changed and when the customer brought up the issue if this testing was necessary, this engineer emphasized the importance of testing changes to the software. The ground support software engineer does not seem to have learned the lesson for implementing the requirements and design and of testing the software. The software lead has had to intervene more often to gain compliance to the process from this engineer. The other experienced engineers — ground and flight — have been working to the process documented in the SDP. The process for this program seems to be working in relation to the walkthroughs, implementation of requirements and design, and testing of the software. The software is still a little behind schedule but that is partially due to the hardware requirements changing.

The evaluation of the flight software requirements specification shows the evolution of the document for the program. The requirements document is reviewed throughout the software development cycle. Table 14 shows the results of the document reviews performed by SQA during the requirements, design (PDR and CDR), and code phases. The requirements document was first reviewed during the requirements phase and was found to be unacceptable due to the lack of testability and traceability of the requirements. The baseline of the document during the requirements phase was an improvement. The document continued to evolve throughout the development cycle and improved slightly with each new revision. This is due to the fact that the requirements were understood better than in the requirements phase and the hardware requirements were more firm than in the requirements phase of the software.

Table 14. SRS Document Completion Index

Phase	DI Score (1.0 high)
Requirements	.50
Requirements (Baseline)	.60
Design (PDR)	.65
Design (CDR)	.73
Coding	.75

PROGRAM 6 was an engineering support task for one of the NASA centers. It is used to show the difference in performance of the programmers involved based on their past experience with software engineering discipline. No contract criterion was available but each was totally responsible for code to work on a particular platform. This program was a small research program that developed software for a power system. The program consisted of a program manager, a software lead that also acted as the deputy program manager and a programmer, and two software engineers. The software lead (P1) was fresh out of college with a masters degree and had never worked a program before, one of the engineers (P2) was fresh out of college but had worked as a summer hire for Martin Marietta, and the last engineer (P3) had several years experience in developing deliverable software. The software was developed primarily in Lisp on a Unix based machine with some software developed on a PC in Pascal. The software was divided into two CSCIs, application software which totaled 116,000 lines of code and lower level processor (LLP) software which totaled 5800 lines of code. The program had a series of change orders which documented the requirements changes to the software and the hardware.

Although this program was a small research program, Martin Marietta has minimum standards that all software programs are required to meet. This consists of a software development plan, documented requirements and design, testing, and configuration control. The program generated a SDP that reflected how the program was going to operate while still meeting the minimum Martin Marietta requirements. A requirements document was generated for the software and maintained through the life of the program. The design for the software was documented in monthly progress reports to the customer. The software was tested at the system level but did not include lower level functional testing. A formal software test was run at the Martin Marietta facility prior to delivery and system test at the customer site. The level of testing was determined by the engineer. There were no coding standards per se, therefore style was dependent on the individual software engineer.

There were no slips in the schedule for program 6 due to the program's statement of work being written such that whatever software was developed at the time of delivery was what the customer accepted. As long as the software had the required functionality, the customer was satisfied.

Table 15 shows the two CSCIs with the total lines of code, the average size of a module and the fault density of the software. The majority of the application software was developed by programmers P1 and P3. The LLP software was developed by P2. The difference in fault density of the two types of software shows the level of unit testing that was performed by the engineers. Programmer P2's software was not sufficiently tested prior to baselining. There is a significant difference between the two engineers, P1 and P2, although they were both new graduates from college when they first started on the program.

Table 15. Program 6 Software

Software	Total LOC	Average Size Of Modules	Total Number Of Modules	Fault Density
Application	116,712	451	259	1.15
LLP	5,857	345	17	3.07

There were 152 discrepancies found in the software of which 12% were requirements changes by the customer and 88% were coding errors.

Table 16 breaks out the errors by priority. The *A* priority can be requirements errors/changes or crashes in the software.

Table 16. Program 6 Percent Discrepancies by Priority Level

Priority	Percent
A	13%
B	13%
C	70%
D	4%

Programmer P2 moved from the LLP software to the Application software when the development was done. When the total number of crashes (*A* priority) are reviewed 53% of the crashes (software that was not tested sufficiently after modification and incorporated into the system) were introduced by programmer P2.

As shown in Figure 16, the majority of the errors were found during engineering test. A large portion of the discrepancies were requirements changes requested by the customer. This resulted in a fault density of 0.05 after delivery of the software to the customer and 1.24 for the program overall.

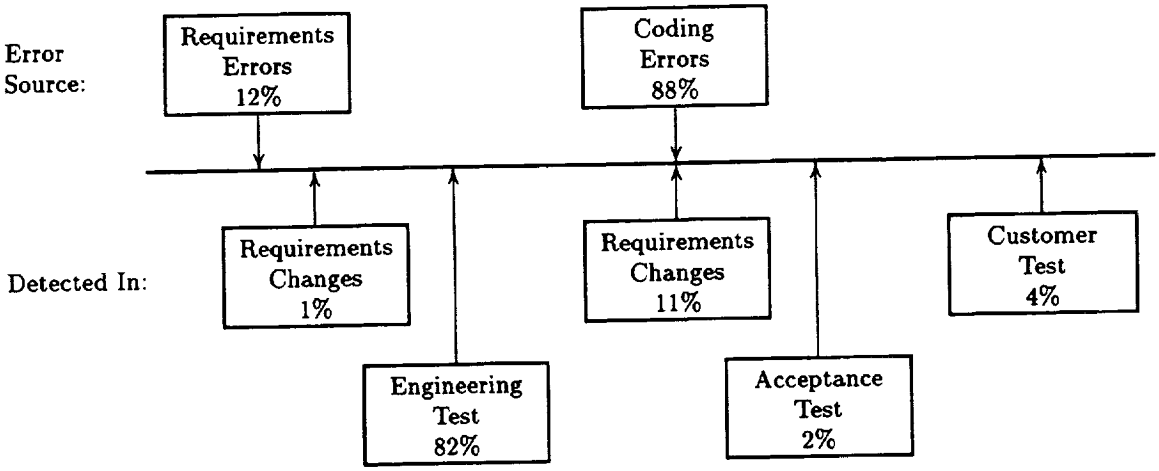


Figure 16 - Program 6 Percentage of Errors

For program 6, there were 122,569 lines of code developed over a five year span. This software changed drastically from the initial development effort because of change of platforms. The software originally started with a Xerox computer, a VME-10, and a Symbolics. The Xerox software was rehosted on a Solbourne, the VME-10 on a 386, and the Symbolics on the Solbourne. The lines of code developed per day could not be computed since the software effort was redefined through requirement changes including rehost efforts for developed software.

The difference in programmer discipline is shown on this program through the fault density measure. The software written by programmers P1 and P3 had a fault density of 1.15 and code written by programmer P2 had a fault density of 3.07. This difference is significant because of the large number of lines of code in the application software that were rehosted compared to the other software. The LLP software rehosted was 5.8K lines of code compared to 116K. Although the process defined for this program was a minimum set of standards, the programmers implemented the process differently. Programmers P1 and P3 were more conscientious in testing of their software than programmer P2. This was reflected in the number of *A* and *B* priority discrepancies found in the application software after programmer P2 moved to the application software. The majority of the rehosting of the application software was complete before programmer P2 moved over. Programmer P2 introduced double the number of *A* and *B* priority discrepancies as the other two programmers.

CONCLUSIONS

Table 17. Program Comparisons

Item	1	2	3	4	5	6
Involvement with Process Definition	YES	NO	NO	NO	YES	N/A
Stable Requirements	YES	YES	NO	NO	NO	YES
Adequate Unit Test	YES	NO	NO	NO	YES	NO
Quality Oversight for Complete Program	YES	NO	YES	NO	YES	NO
Schedule Slip	NO	30%	23%*	14%	17%	NO
Engineering Overrun	NO	36%	*	53%	NO**	NO
Quality Overrun	NO	10%	*	47%	NO**	NO
Planned Quality Budget versus Engineering	5%	13%	14%	12%	10%	2%
Actual Quality Budget versus Engineering	5%	10%	*	14%	**%	2%
Productivity Planned	6	20	9	22	7	25
Productivity Actual	6	16	*	12	**	25

* Program cancelled, data unavailable
** Program in process of completing

1. Following the Process

In all these programs a development plan was required by company standards and, on some, by contract requirement. Those programs that participated in defining the contents of the plan were more likely to adhere to it. Participation certainly strengthens understanding the contents which goes a long way toward following the process defined. Programmers accustomed to meeting standards adjust to new process definitions and respond to changes in their development environment by getting on with the job,

It is helpful to establish early who follows the process, so that additional resources can be brought to bear to gain adherence and reduce rework and action items.

Program 3 was, in the experience of the authors, a classic example of how direction from the software leads can effect the adherence to process. The worst adherence to schedule was shown by the group under the guidance of what we would describe as a whiner who wanted no rules, no oversight, and no process. The best adherence and best response for time to fix was demonstrated by the group directed by a person who expected adherence to schedule and expected the group to follow the agreed upon methodology as a team.

Having the people involved in the work responsive to the process is superior to bringing in the heavy artillery, i.e., upper management, to dictate compliance.

Program 6 is an example of what individuals bring to the job. This small program had little structure, no documentation other than task descriptions of a very high level, and the individual programmers chose their own method of test and evaluation of the results. One set of software had a significantly higher fault density than the other, though neither rating was excessive. The programmer who produced code with the higher fault density also had the most errors causing system crashes. The programmer producing code with the lower fault density approached the development of the application in a structured manner going from requirement to design to code and test. The other sat at the terminal and wrote code till it appeared to work.

2. Changing Requirements

As discussed under programs 3, 4, and 5, changing and/or nebulous requirements handicap a program. Program 3 overcame poor software requirement definition by establishing preliminary design based on the system specification and continuing to work the software requirements through the design and coding phase. This represented the design and development personnel overcoming the process. The reluctance of the requirements group to meet their responsibilities because of an overall program problem of requirements flux was a losing situation. The process dictated that this group produce and it did in volumes of bad documentation. Eventually, the responsibility for the product migrated to a more focused group under the control of the end users that allowed the evolution of an acceptable product.

Program 4 requirement changes came mostly from the customer and usually involved functionality in the ground support software. Such changes are more easily accommodated than hardware or design change and had little impact on delivery of the flight unit. A problem on program 4 involved not keeping the breadboard used to test the software current with flight hardware design changes. This caused considerable consternation and finger pointing when system integration testing could not start because the software would not run the hardware. The lack of communication and understanding of need was corrected by replacing the software lead with someone experienced in development of software for an embedded system.

Program 5 still has time to overcome the program slip introduced in its design and coding phase caused by requirement changes.

Performing document evaluations and reporting the document completion index provides visibility into progress toward meeting program goals. This is especially important with the requirements document since inadequacy in it is felt through the following phases.

3. Discipline in Informal Testing

Informal test lacking objective goals accomplishes little but can give a false sense of security. Programs 2 and 4 suffered from too little structured unit testing. This is shown in the large jump in defects found when test philosophy and responsibility changed. Both programs experienced considerable engineering overruns and schedule slip.

Program 2 personnel adjusted to the new test philosophy and informally competed with each other for best time to fix and fixing it right the first time. Program 4 personnel were so burdened with ego that to accomplish adequate unit test and institute correctness of fixes programmers had to be replaced.

On Program 3, the considerable number of slips in scheduled code walkthroughs and baselines for the ground CSCI were caused by additional coding effort needed to complete informal integration of the CSC into the CSCI. Except for the leads on the ground CSCIs, the programmers were used to developing stand alone code and lacked an understanding of meeting interface requirements.

The unit test on program 6 was defined and executed by the individual programmers involved. As reported previously, lack of rigor in unit testing surfaced in system test as more software crashes were introduced by one of the programmers than the total for the other two.

4. Reviewing Early

Trying to save budget by avoiding the use of early objective reviews as shown in program 4 can be self defeating. An inspection or review process that can point out errors early but not effect their correction, as shown on program 3, is a poor process in terms of cost and quality. The thorough review process used on program 1 was integral to keeping on a very tight schedule.

5. People

Pinpointing who is most likely to cause rework and frustration is an effective way to control rework. It is a matter of tracking number of action items and response to action items. If a problem with responsiveness or understanding of the importance of compliance to the process is identified, management must accept the responsibility of replacing the problem, redefining standards, or strong arming compliance.

6. Collecting Data

Using a database tool to pool data makes sense. However a project evolves, it will have reviews/evaluations of some type, discrepancy reporting and status to schedule. Collecting the result of objective reviews and other defect data should be a high priority for complete quality records. Current status of development progress is needed to flag problem areas and replan work to make up for known slips. The tool used to collect the information on these programs, SQUID, was designed around the Air Force pamphlet, Software Management Indicators, Management Quality Insights, Air Force Systems Command, 20 Jan 1987 and practical knowledge based on twelve years of performing software quality on projects. Reporting from a database is superior to digging through data retention boxes. Use of a tool can give structure to multiple quality programs and allow meaningful comparisons between different types of projects.

Each program undertaken yields a better idea of what is a meaningful measure. Program 2 was the first program that the authors had the opportunity to collect metrics on during the life of the program. Defect density and coding complexity measures were used because involvement began after coding start. After this experience the authors started looking more at early fault detection via walkthroughs and reviews. This in turn lead us to look at responsiveness of individuals in correcting deficiencies as a major driver in software quality.

In summary, we believe our observations support the conclusion that good programmers produce good code. A good programmer in the context of this paper is a programmer who is committed to project goals, highly disciplined, and responsive to constructive criticism that is based on meeting those goals. Of itself, a process does not make a product. The best a process definition can do is let producers know what is expected before they are evaluated for method and output. Proper execution of a process requires the cooperation of the participants. The more readily this cooperation is given the less the cost of rework.

Software Quality: Process or People

Regina Palmer
Martin Marietta Astronautics
P. O. Box 179, M/S S1008
Denver, Colorado 80201
&
Modenna LaBaugh
R&FC Group
3076 So. Hurley Circle
Denver, Colorado 80227

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Agenda
    - Introduction
    - Background of Programs
        - Process
        - Success
        - Problems
    - Lessons Learned

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Introduction
  - The Paper Describes the Following for Each Study
    - Involvement with process definition
    - Stability of requirements
    - Thoroughness of unit and system test
    - Degree of Quality oversight
    - Schedule Adherence

      Variance from planned completion dates
    - Overruns – Engineering and Quality
    - Planned Quality Budget as percent of Planned Engineering
    - Actual Quality Budget as percent of Actual Engineering
    - Planned Productivity

      Lines of code per engineering day
      Includes Program Management, Engineering, and Quality hours
      Excludes Business Operations and Property Management — not readily available to authors
    - Actual Productivity

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 1
  - Very involved with process definition.
  - Very stable and well defined requirements. Clearly defined end of phase before beginning of next. As shown by the high Document Completion Index scores.

Phase	DI Score (1.0 high)
Requirements	.67
Design (CDR)	.77
Coding	.83

  - Thorough unit and system test
  - Quality oversight from beginning of contract
  - Schedule Adherence – no slip
  - Overruns – none
  - Planned Quality Budget as percent of Planned Engineering – 5%
  - Actual Quality Budget as percent of Actual Engineering – 5%
  - Planned Productivity – 6 LOC
  - Actual Productivity – 6 LOC

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 2
  - Little involvement with process definition. Programmers were unfamiliar with testing rigor and programming standards. Table is in order of programming experience.

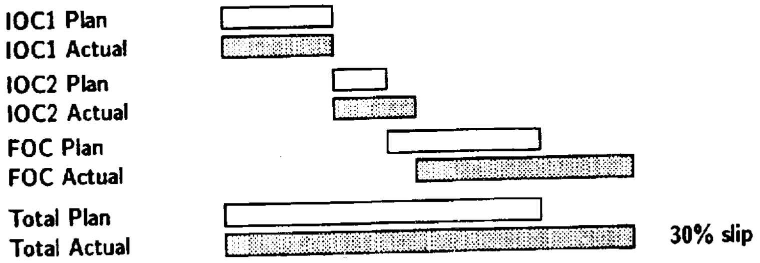
Programmer	Fault Density of Code Produced
P1 - sw lead	8
P2 - sw lead	18
P3 - new graduate	10
P4 - new graduate	8
P5 - new graduate	17

  - P2 and P5 were reluctant to take time to test. P4 was used to do most correction of P2 code and P3 was used to correct P5 code because of the low fault density of code they wrote.
  - Disagreement late in program on requirements
  - Quality oversight on program from beginning of coding

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 2 (continued)
  - Lack of discipline in unit test - 60% of errors found in final testing were in the first baseline.
  - Schedule Adherence - 30% slip in final delivery of software. Legacy code required upgrades to meet customer expectation of usability that was not anticipated in contract bid.



12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 2 (continued)
  - Overruns
    - Engineering exceeded plan by 36%
    - Quality exceeded plan by 10%
  - Planned Quality Budget as percent of Planned Engineering – 13%
  - Actual Quality Budget as percent of Actual Engineering – 10%
  - Planned Productivity – 20 LOC
  - Actual Productivity – 16 LOC

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 3
  - No involvement with process definition except for defining coding standards
  - Very unstable and poorly defined requirements. Document completeness scores for documents highlight this.

Phase	SRS	SDD	STP	Project
PDR	.17	.55	.50	.40
CDR	.36	.74	.61	.57
Coding	.64	.74	.61	.66

  - Unit test had only begun at contract close
  - Quality oversight from beginning of project but no authority for action item resolution

CSCI Name	Number of Items	Average Response Time (days)
FLIGHT	24	33
EGSE	13	46
TRAINER	5	60
SIMULATOR	10	83

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 3 (continued)
  - Schedule Adherence – 23% slip due to requirement definition during design phase
    - Attitude of software leads to maintain internal schedules varied.

CSCI NAME	W/T Dates Missed	Average Slip (days)	B/L Dates Missed	Average Slip (days)
FLIGHT	10%	19	23%	15
EGSE	26%	28	43%	21
TRAINER	14%	7	0%	0
SIMULATOR	18%	28	18%	14

  - Overruns
    - Engineering – Program cancelled, unable to compute
    - Quality – at beginning of code phase had exceeded evaluation budget by 173%.
  - Planned Quality Budget as percent of Planned Engineering – 14%
  - Actual Quality Budget – Program cancelled, unable to compute
  - Planned Productivity – 9 LOC
  - Actual Productivity – Program cancelled, unable to compute

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 4
  - No involvement with process definition
  - Unstable requirements – 33% of change traffic for the last build before delivery were enhancements requested by the customer.
  - Lack of test discipline in unit test – Large number of errors found after build 1 when other programmers were brought in to validate software.

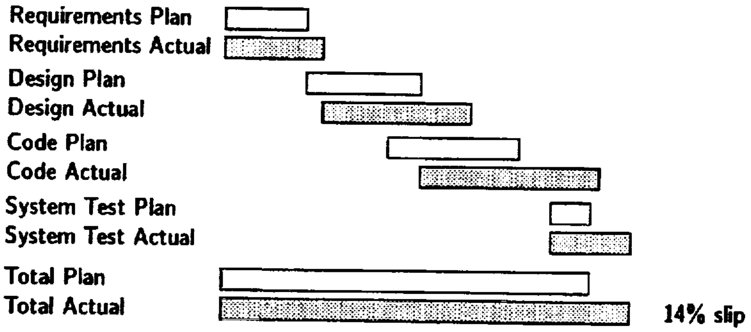
	Number of Defects
Build 1	16
Build 2	72
Build 3 thru Delivery	35

  - Quality oversight begun in coding phase
  - Overruns
    - Engineering – 53%
    - Quality – 47%
  - Planned Quality Budget as percent of Planned Engineering – 12%
  - Actual Quality Budget as percent of Actual Engineering – 14%

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 4 (continued)
  - Schedule Adherence – 14% slip

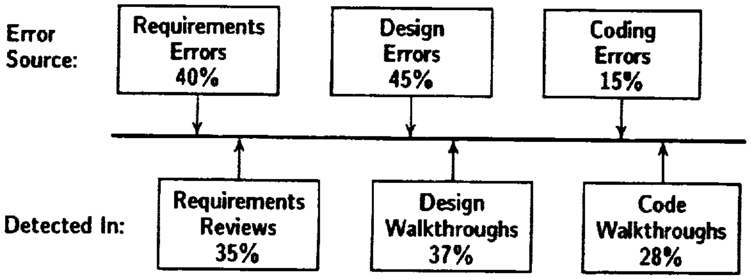


  - Planned Productivity – 22 LOC
  - Actual Productivity – 12 LOC

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 5 ·
  - Very involved with process definition. Reviews used in process very successful for finding errors early.



Error Source: Requirements Errors 40% | Design Errors 45% | Coding Errors 15%

Detected In: Requirements Reviews 35% | Design Walkthroughs 37% | Code Walkthroughs 28%

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 5 (continued)
  - Responsiveness of individual programmers to the corrective action process is shown in the table.

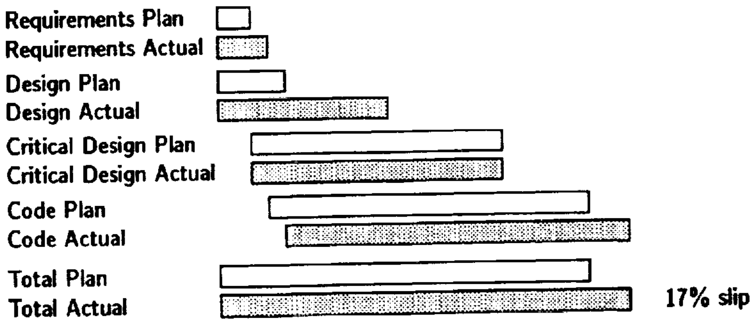
Programmer	W/T Action Items	LOC	Average Time To Close
P1	49	9325	4 days
P2	31	1190	2 days
P3	70	1449	2 days
P4	45	3227	2 days
P5	10	1120	2 days

  - Stability of requirements – requirement changes were imposed during the design phase impacting Preliminary Design schedule but time was made up during critical design. Hardware requirements changed after design baseline.
  - Thoroughness of unit test due to stress of software lead
  - Quality oversight from beginning of program
  - Planned Quality budget as percent of planned Engineering – 10%
  - Actual Quality Budget as percent of actual engineering (to date) – 10%

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 5 (continued)
  - Overruns – Program has not completed code phase yet, not calculated to date
  - Schedule Adherence – 17% slip appeared in code phase, to be made up in test



  - Planned Productivity – 7 LOC
  - Actual Productivity – Program not due to complete till next year

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 6
  - No process definition
  - Requirements changes caused new work order
  - Unit test performed at a level defined by the individual programmer. The difference in testing thoroughness is shown in the fault density of the software as measured by discrepancies found during system test.

Software	Total LOC	Average Size Of Modules	Total Number Of Modules	Fault Density
Application	116,712	451	259	1.15
LLP	5,857	345	17	3.07

  - Quality provided configuration control only for delivery to customer
  - Schedule Adherence – as a level of effort contract the schedule is always met
  - Overruns – N/A, level of effort
  - Planned Quality Budget as percent of Planned Engineering – 2%
  - Actual Quality Budget as percent of Actual Engineering – 2%
  - Planned Productivity – 25 LOC
  - Actual Productivity – 25 LOC

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Lessons Learned
  - Process definition must involve the participants to assure acceptance.
  - People who do not accept the process cause rework expense. Those likely to cause rework or delay are identifiable by tracking action items and response times.
  - Programmers must be made aware of objective goals for unit testing.
  - Fault density by itself is a deceptive measure. Using it, program 2 was estimated to be on schedule two months before the final delivery. Before end of test it became obvious that the unit test program had been inadequate and a 7 month slip occurred.
  - Programs which only collect metrics during test miss opportunities for early detection of problems.
  - Program knowledge disappears soon after each milestone on a program unless someone collects it as it happens.
  - Each program undertaken yields a better idea of what is a meaningful measure within a company culture.
  - Data for this paper were scattered amongst individuals involved on the programs reported and not readily available till input into the database tool used.

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

Program Comparisons

Item	1	2	3	4	5	6
Involvement with Process Definition	YES	NO	NO	NO	YES	N/A
Stable Requirements	YES	YES	NO	NO	NO	YES
Adequate Unit Test	YES	NO	NO	NO	YES	NO
Quality Oversight For Complete Program	YES	NO	YES	NO	YES	NO
Schedule Slip	NO	30%	23%*	14%	17%	NO
Engineering Overrun	NO	36%	*	53%	NO**	NO
Quality Overrun	NO	10%	*	47%	NO**	NO
Planned Quality Budget To Engineering	5%	13%	14%	12%	10%	2%
Actual Quality Budget To Engineering	5%	10%	*	14%	**%	2%
Productivity Planned	6	20	9	22	7	25
Productivity Actual	6	16	*	12	**	25

* Program cancelled, data unavailable

** Program in process of completing

12/2/93 GSFC Software Engineering Workshop

Session 6: Software Engineering Issues in NASA

omit

Dana Hall, Science Applications International Corporation

John Bailey, Software Metrics, Inc.

Marvin V. Zelkowitz, University of Maryland

Profile of NASA Software Engineering: Lessons Learned from Building the Baseline

Eighteenth Annual Software Engineering Workshop
2 December 1993
Goddard Space Flight Center

Dr. Dana Hall
Science Applications International Corporation
and
Frank McGarry
NASA Goddard Space Flight Center

It is critically important in any improvement activity to first understand the organization's current status, strengths, and weaknesses and, only after that understanding is achieved, examine and implement promising improvements. This fundamental rule is certainly true for an organization seeking to further its software viability and effectiveness. This paper addresses the role of the organizational process baseline in a software improvement effort and the lessons we learned assembling such an understanding for NASA overall and for the NASA Goddard Space Flight Center in particular. We discuss important, core data that must be captured and contrast that with our experience in actually finding such information. Our baselining efforts have evolved into a set of data gathering, analysis, and cross-checking techniques and information presentation formats that may prove useful to others seeking to establish similar baselines for their organization.

Role of the Baseline in Process Improvement

We use the term "baseline" to mean a relatively detailed understanding of the software engineering products, processes, and environment characteristic of an organization, large or small, in a given period of time. It is a snapshot of current product attributes and of present software engineering processes and the environment within which those processes operate. The fundamental objective is to gain understanding and not to judge that the way the organization performs its software development, maintenance, management, and assurance is right or wrong. This understanding is then used in two principal ways; first, to help identify and define potential improvements and, second, to serve as a reference against which future comparisons are made as candidate improvements are prototyped and implemented.

Establishing the baseline is the mandatory first step of any process improvement program. Determining the organization's software and software engineering characteristics before proposing and trying an improvement requires discipline. It is reasonably analogous to the discipline required to first understand a software problem's requirements and design before jumping in to write code. Although some random "improvements" might prove correct, experience has repeatedly shown that most are off the mark and are almost always short lived, frustrating, and wasteful of people's interest and resources. Thus, it is very important that the time and energy be taken to first gain insight and to understand the what's, how's, and why's of an organization's way of doing software business.

The understanding step provides the foundation for all of the process improvement program. The figure on page 4 illustrates the iterative and chronological relationship between the three fundamental steps of the basic process improvement paradigm. The Figure shows that gaining understanding precedes and then parallels the assessing and packaging steps. Examples of the types of insight comprising the understanding step are shown and will be further discussed below. Note that the understanding process is never completed. It continues on through repeated cycles of update as well as probing into lower levels of detail when and as needed to facilitate focused assessing and packaging. The assessing step uses the insight gained from the understanding activities to identify and define focused improvements that appear to be beneficial and cost-effective. The assessing activity includes prototyping and experimentation. Examples of such experiments are trying an improved inspections process or alternate testing technique. Those improvements that do prove helpful are then packaged as policies, standards, or guidebooks for promulgation back into the organization. Over time and with continued attention, the modifications become a routine part of the organization's software culture and the fundamental software engineering baseline of the organization thus will have advanced.

The figure on page 4 also helps to show that policies, standards, and guidebooks evolve from hands-on experience and usage. As a result of actual implementation in the culture of the organization, the policies, standards, and guidebooks serve as a means for communicating and helping to achieve greater uniformity. The people in the organization own and recognize the "rules" as simply their way of doing business. Experience has shown that this bottoms-up process is much more effective and useful than is adopting or tailoring a process from another organization or from some top-down mandate.

Core Data You Want to Capture

The organization's software characteristics baseline consists of four categories of information. These categories are:

1. Insight about the organization's application domain(s)
2. Characteristics of the end items; i.e., the products and/or services the organization provides
3. Attributes of the process(es) the organization uses
4. Insight about the organization's environment; i.e., its supporting tools and computing and networking infrastructure.

The figure on page 6 presents the core data that comprise these four information categories. This is the basic data you want to find in order to achieve a first order understanding of the organization and its software practices. Most fundamentally, insight is required about what the organization's application domain or domains are. In other words, what does the organization do, what people and budget resources does it have, and where is the organization trying to go (what are its goals?). Software development and maintenance often are only a part of its purpose so insight must be gained about the organization's overall roles and its software work as a subset of those roles. Further, since many organizations perform work in more than one domain, understanding must be gained about the allocation of resources across those domains.

Using the knowledge of the organization's application domain(s) as a foundation, insight is then gathered about the its products, processes, and computing environment.

Product insight is most readily quantifiable. The amount of software under development and the amount being maintained, languages in use, and error characteristics of the operational software are examples of key product attributes. As illustrated on page 7, we learned in our baselining activities at the NASA Goddard Space Flight Center (GSFC) that of the total civil service and support contractor population (a community of some 12,000 people), roughly one third spend the majority of their time doing software engineering-related work. By "software engineering-related", we mean performing one or more of the functions of software management, requirements definition and analysis, design, coding, testing, configuration management, and quality assurance. That community is responsible for the maintenance of at least 43 million source lines of presently operational code. We have found that across NASA all software activity groups into one of six major application domains: flight/embedded software, mission ground support software, general support software, science processing software, administrative software, or simply (for lack of a better title) research software. The distribution of existing operational software at the GSFC into the major NASA software application domains is shown in the figure on page 8. By way of definition, mission ground support software is the ground software necessary for the preparation and conduct of a space flight mission. Examples are command management software and software for determining vehicle orbital position. General support software includes engineering models, simulations, tools, and similar operational software. While pie charts are a useful analysis aid, other formats are also helpful. The graphic on page 9 uses a histogram format to show some of our baseline about software language preferences at the GSFC. This particular figure compares the language characteristics of currently operational software and the preferences of the developers of new software. (Neither the GSFC overall nor any of the organizations that comprise the space center have mandated or advocated a software language or set of languages. Each developer or project typically selects the language they prefer.) It is interesting to note the sharp decrease in FORTRAN use and the significant increase in the preference for C and C++. (Our baselining did not, unfortunately, distinguish between C and C++. We may look more specifically into the use of those two languages in the near future.) Ada use has grown at GSFC, but not to a prominent amount.

An organization's process characteristics may prove more difficult to determine. The availability of such insight in any reasonably quantified way is, in our experience, a good indicator of the organization's software engineering maturity. The managers of an organization with immature software engineering practices are not able to easily describe and prove usage of their policies and standards, the usual allocation of resources by software phase, tools used and the usefulness to the organization of those tools, or other key process attributes such as frequency and characteristics of reviews, staff training practices, and project software configuration control activities. An example of understanding an organization's process characteristics is shown in the schematic of page 10. This graphic portrays the usual GSFC software project's consumption of resources by each of the major phases that make up the software development process. This figure is presented here simply as an example of process insight, but it is also an interesting figure to briefly discuss. We found this allocation of resources to be roughly constant at GSFC regardless of activity development approach or size. While we have, for simplicity of presentation, mapped resource usage into a classical, one pass software development model, many projects at GSFC (especially the smaller ones) develop their software through an iterative, build-it and try-it approach. Almost invariably, these one person and small team efforts claimed not be following any particular development model and certainly not to be doing anything as formal as "requirements definition", but in reality, we observed that their work processes do cycle through the basic four phases, albeit in an informal, less structured way. So, our baselining indicates that the resource usage summarized by the schematic on page 10 is fairly typical of most software work at the GSFC.

A fundamental rule associated with establishing an organization's software engineering practices baseline is to not judge during the baselining process whether an observed practice is good or weak. The objective is simply to observe and record. One of several classical software engineering rules of thumb, for example, says that the front end requirements and design processes should receive 40 percent of the development resource budget while coding receives 20 percent, and testing the remaining 40 percent. GSFC software work varies some from that guidance. We do not at this stage argue or even examine whether some alternate resource pattern might be more effective for the typical GSFC software effort. However, comparisons such as this help to highlight further exploration to be done during the subsequent Assessing phase.

The fourth area of required insight concerns the organization's computing environment. This part of the baselining focuses on attributes such as the types of computers available, how networked the organization is, and how integrated are its software tools. An objective is to measure the organization's computing environment relative to the current state of practice generally in place across the software industry and to identify and characterize constraints that limit the organization's ability to continually or periodically upgrade. Aside from budgetary pressures, a constraint in the GSFC environment, for example, is the large quantity of currently operational software that must be maintained. Taking care of that installed base of some 43 million source lines of code inhibits the modernization of much of the computing infrastructure. We noted, as a consequence, continued reliance on centralized processing and limited introduction of more recent advances such as client-server architectures and powerful desktop computers. Understanding the organization's constraints with respect to its computing environment helps to set the practical context within which incremental improvements are possible.

What Data Can Be Captured?

While a core of data is critical as the foundation for an organization's software engineering improvement baseline, the reality is that often much of even that core data is not available. This is especially the case in an organization whose software engineering practices are relatively immature. Immature is meant as a descriptor of organizations where software engineering practices are largely driven by individual preferences, where little or no organized measurement is performed, and where there is little uniformity and sharing across the organization. The Software Engineering Institute at Carnegie Mellon University labels such organizations as "level 1" in its five level capability maturity model.

Establishing a baseline is an incremental process in and of itself. As the figure on page 4 emphasized, understanding begins as the first step and then is a process that continues in parallel as the organization experiments with and implements focused improvements. The data gathering process is one of iteratively gaining sufficient insight to identify and define promising improvements. Detailed accuracy and depth are not necessarily needed at least initially for the organization-wide baseline. As candidate improvement areas are identified, however, more indepth probing will usually help to understand weaknesses and to shape the nature of candidate changes. More will be said a little later in this paper about the balance between resources put into the baselining process and the level of depth and accuracy of the baseline.

In our GSFC and NASA-wide baselining work, we have been reasonably successful gathering insight about software languages in use, budgets, and quantities of software (as measured by lines of code and people involved). We estimate that our results for these types of measures are accurate within 25 percent of the true amounts. Twenty five percent is admittedly a wide margin, but it is adequate for our software process improvement needs at this early stage in the NASA Software Engineering Program. We have not been as successful identifying other less tangible core data. Examples of such data include effort distribution by phase, the operational lifetime (longevity) of software, error statistics of any kind, productivity measures, and the amount of resources typically invested in the key "overhead" functions of software quality assurance, configuration management, documentation, and project management. As discussed later in this paper, we believe that we have directly or indirectly gathered data from about ten percent of the GSFC software community. Very few of the managers and staff with whom we interacted had any quantified data pertaining to these less tangible measures. Most could offer only qualitative guesses. While such insights are probably better than no insight at all, we put a wide margin of error of 50 percent on that subset of our characteristics baseline. The point of this discussion is to emphasize that although a core family of process attributes are important to your understanding baseline, practical circumstances may dictate that you settle, at least in the early stages of an improvement program, for approximations and opinions for some of the desired data.

Techniques for Establishing the Baseline

We have found over the past year and six months that a combination of four methods works best to gather, cross-check, and understand the software domains, products, processes, and environment characteristic of a subject organization. The four techniques that comprise our integrated data gathering mechanism consist of administered survey vehicles, informal roundtables, review of selected project data and documentation, and one-on-one interviews. These techniques reinforce each other. Our experience indicates that all four are necessary in order to truly understand the what, how, and why of an organization's software business.

Surveys are a key instrument. We developed, tested, and placed under configuration control a comprehensive eight page survey and a single page, special subset. The single page version was used in two ways. One application was to help introduce our purpose to senior management, garner their support and approval, and elicit their insight about the software engineering process in their organization. The other way we used the single page version was as a verification mechanism with various individuals throughout the organization that had not been interviewed using the longer survey.

The eight page, main survey was widely applied. We found the most effective administration method to be a technique similar to that of a census-taker. At a pre-arranged meeting time and location, our data gatherer met with the single or occasionally several respondents. After introductions, a short explanation would be given of the NASA Software Engineering Program and the role of the characteristics baselining activity within that Program. The data gathering meeting would then proceed in the style of a question and answer session using the survey as the central script. A simple "don't know" or "not available" was entered as the response for those questions where the respondent could not easily answer. We had structured our questions so that a knowledgeable respondent would not need to invest time researching and compiling answers. Typically, the data gathering session consumed about 1.5 hours, although related discussions would sometimes extend the duration. This method of meeting with the respondent and walking down through the survey not only used minimal time, but also assured us of data return and data interpretation consistency. Since we usually used only one and occasionally two data gatherers, we were able to maintain a relatively consistent interpretation and level of detail. Early in our baselining, we tried and discarded easier techniques that relied upon survey mailouts and telephone calls.

Prototyping of the survey mechanisms proved very important. Such testing helped identify confusing questions, inconsistent definitions, and to polish our gathering techniques. We found, for example, that entries for descriptive data introduced too much variability and thus complicated our data reduction job. A more effective approach was to only use questions with definitive answers such as yes/no or response ranges (for example, <35%, 35-70%, or >70%).

The roundtable sessions were used to help check the insights gathered from the administered surveys and to gather more subjective opinions and advice. The roundtables were conducted using a structured set of five major questions; specifically:

1. Participant background and experience?
2. Business goals and objectives of the organization?
3. What software process is used?
4. Major strengths and weaknesses in developing software?
5. What could be done to improve the software engineering process in the organization?

Separate roundtables were conducted for managers and technical personnel and for civil servants and support contractors so that each set of participants could speak more freely. As with the survey and interview results, we have taken care to protect the privacy of the responses. In no case have observations been attributed back to individual participants.

While our survey administration method was in reality a one-on-one or occasionally one-on-two interview, we also used the interview technique principally as a means of pursuing insight that appeared, after analysis and comparison, to be contradictory or in some way particularly different from what we were learning was the norm in the organization.

Resource practicalities obviously precluded talking with every member of the software community so we tried to orient our data gathering energy to the major "pockets" of software work. Senior management proved very helpful in pointing to those software intensive groups within the overall organization, but our knowledge of NASA was also key. This brings up another valuable lesson. It is our opinion that the software engineering baselining process can only be done by individuals familiar with the target organization and its culture. That insight has proven highly important to our efforts at interpreting terminology, understanding roles and functions, and interpolating and extrapolating the data samples to represent the overall GSFC and NASA software communities. We do not believe that we would be very effective if we attempted to conduct this critical activity in unfamiliar domains or, conversely, if someone not familiar with NASA tried to conduct a NASA software baselining.

How Many People Should You Talk To?

No easy answer can be provided to this question. The amount of interaction depends upon the variability within the organization of interest and on its software engineering process maturity. The problem of sample size is probably amenable to statistical analysis. We have relied upon our extensive NASA experience base as our primary guidance for determining our sample size. As the graphic on page 13 shows, we sampled approximately ten percent of the GSFC software community and from that sample size, believe we have extracted insight sufficient to both guide our next round of focused improvement thrusts and to serve as a yardstick for future comparison. We cannot judge that ten percent is a proper sample size for improvement endeavors in other organizations, however.

How Much Will the Baseline Cost?

We have invested approximately eighteen person-months in the baselining activities focused on the GSFC software community. Our efforts first concentrated on the largest and most software intensive Directorate at GSFC and then broadened to encompass all of the GSFC, but at a lesser level of detail. As the data on page 14 shows, a cumulative six person-months was used gathering insight using the integrated four method approach previously discussed. This six months was preceded by two person-months of survey development, testing, and refinement. We found the archiving investment to be extremely important. This function helped to maintained order and organization amidst the inflow of large quantities of data. We estimate that about six person-months have been invested extracting and deriving information and insight from the survey, roundtable, and interview data. This function includes recognizing and dealing with data overlaps and gaps as well as performing analyses and comparisons. A total of two person-months has been expended so far packaging our results into two profile reports (one for the major software Directorate and one for the GSFC overall) and into several briefings.

Page 14 concludes with a short synopsis of our next steps. We are now transitioning from exclusive concentration on just gaining understanding to an effort balanced between continued understanding activities and focused assessments and prototyped incremental improvements. The software engineering baselining performed to date has highlighted several process areas as promising candidates for improvement.

Training and standards are high on that attention list. Several comments in this regard may be helpful. The GSFC training office has an excellent record of responding to managers' requests for specific training classes. Our observations conclude, however, that a more comprehensive software training activity may be cost-effective. We are interested in examining the advantages and problems surrounding an integrated software engineering training program; a curriculum that routinely prepares personnel for upcoming software roles. We also want to explore whether training effectiveness can be improved by expanding the delivery of training from the traditional classroom to include reinforcement mechanisms such as easy access to help and information from each person's office desktop machine.

The standards area apparently requires a considerably different approach from that used within the GSFC to date. (Since we haven't completed our NASA-wide baselining, we can't fully conclude that the same issue applies across all of the Agency, but our insight so far leads us to think that it does.) Several observations are particularly relevant. First, the existence of advocated software engineering processes and supporting standards is very inconsistent. Few organizations that do software work have any recommended approach at all despite the importance of software engineering to their existence and to the credibility of their products. Second, within those few organizations that do have a recommended software process and supporting standards, managers and staff either claim not to know of the recommended approach or freely take broad license to tailor and selectively apply elements of it. Related to this issue is the tendency for civil service personnel to require the contractor community to follow a particular process and standards while they themselves know very little about it and exercise no similar discipline on their own software activities. There is a distinct lack of ownership by the using community of processes and standards imposed from outside their immediate organization. We think these interrelated issues sum to the conclusion that most current approaches to developing, advocating, and using software process and engineering standards simply do not help and instead actually hinder, frustrate, and waste resources. Since current software standards methods largely don't work, our next steps in this part of the improvement program will be dual thrusts of continued detailed understanding and prototyping of alternate techniques. A very promising overall approach is that represented by the three layer, iterative model previously shown on page 4. This technique basically plays back into the organization the methods and practices the organization itself or a subset of the organization such as a particular project, has found helpful. These "packaged" methods and practices are the organization's processes and standards and since they are based on actual experience within the organization, they are owned and used with much greater effectiveness.

Lessons We Learned

In summary, the understanding activity is a mandatory and continuing element of any organizational software engineering improvement program. We have now completed the initial understanding baseline for software across the 12,000 person GSFC community. Several lessons from that work may be beneficial to others seeking to establish similar baselines for their organization.

A primary lesson is to be objective. The purpose of the understanding activity is to learn and not to judge that current practices are good or bad. As the understanding builds, a change in perspective can occur to identify candidate areas for improvement, but the key is that shift in perspective be based on facts rather than speculation.

It is important that insight be gained from personnel at all levels and roles within the organization. We found that interacting initially with upper management was especially important. Not only could the upper manager orient us to the types of work and the responsibilities of the organization, but we also gained the manager's approval of our efforts which in turn helped gain time and attention from the staff within the organization. Another benefit of starting with upper management was the important aspect of buy-in by the upper manager to the concept of software engineering improvement. This acceptance becomes especially important downstream when the initial understanding is achieved and the task of defining and experimenting with promising improvements gets underway. Gaining multiple perspectives from personnel throughout the organization does come with some problems though. The chief difficulty is recognizing and resolving overlaps in data and insights.

Lesson 3 on page 15 recommends layering the baselining effort. In other words, gather insight that is truly representative of the way the organization does its software work, but go only into as much depth as is needed for the current stage of the improvement effort. As candidate improvement areas are identified, more indepth investigation can be done concentrated on the aspects of that candidate area. As in any data gathering exercise, it is very easy to become overwhelmed with data and not be able to discern from all the data the useful information. Keeping a carefully organized archive helps, but the real key is to maintain a perspective of "peeling away layers of insight" as is most useful to your stage of improvement.

As the understanding builds, package the insight into some type of communicative medium. We have used both briefings and reports (which we call "software engineering profiles") as convenient repositories to store our insight and facilitate further discussion and progress within the organization. To help these products mature, give members of the organization opportunities to read and comment. This will likely be a challenging activity, however, because your baselining work will have identified weaknesses and problems which the organization may not want to hear or see on paper. Again, the support and commitment of the upper managers to process improvement become important contributors to the success of your efforts.

Finally, we have evolved to a combination of four methods to gather and verify process insight. Surveys work well if administered in person, if thoroughly tested for completeness and consistency, and if conducted by a single or at most very small team of personnel knowledgeable about the kinds of work the organization performs. Roundtables are a means of gathering in more subjective opinion and perspective. Reviewing selected documentation and data and follow-up interviews serve as tools for verifying and clarifying important items in your evolving understanding baseline.

References

1. Software Engineering and Assurance Plan, draft, July 23, 1993, NASA Headquarters Office of Safety and Mission Quality.

2. Profile of Software Within Code 500 at Goddard Space Flight Center, NASA Software Engineering Program, December 1992.

3. Profile of Software at the NASA Goddard Space Flight Center, NASA Software Engineering Program, draft, December 1993.

Profile of NASA Software Engineering: Lessons Learned from Building the Baseline

Dana Hall
Science Applications International Corporation

Frank McGarry
NASA/Goddard Space Flight Center

Eighteenth Annual
Software Engineering Workshop
December 2, 1993

Topics

- Role of the Baseline in Software Process Improvement
- Core Data You Want to Capture
- What can be Captured?
- Techniques for Establishing the Baseline
- Lessons We Learned Assembling the GSFC/NASA Baselines

(2)

Role of the Baseline in Process Improvement

- Objectives -

1) Establish the Baseline
   - Snapshot present attributes of the software itself (Software Product)
   - Snapshot present software engineering practices (Software Process)

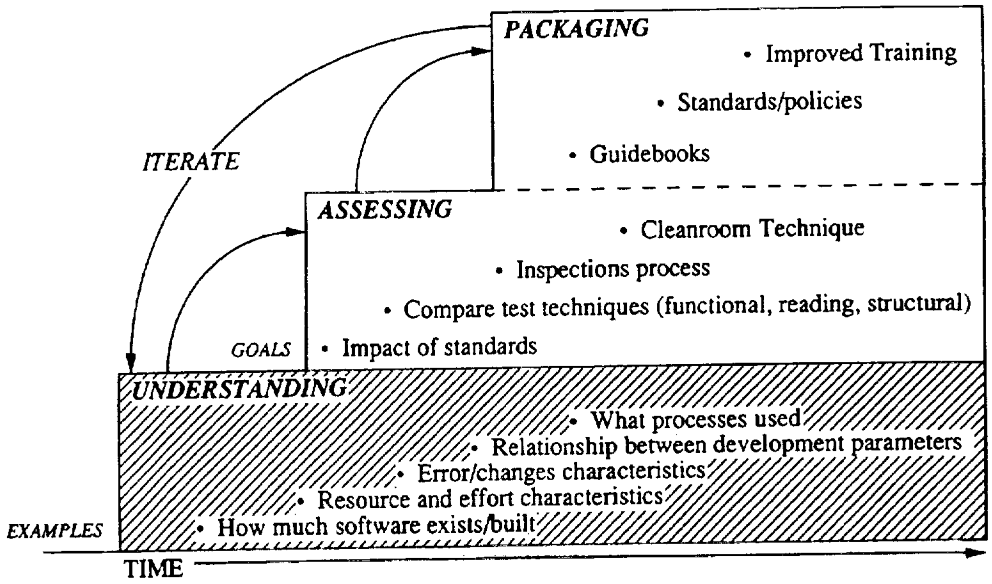
Basic objective is to understand; not to judge right or wrong

2) Baseline will be used to:
   - Identify and define potential process improvements
   - Make future comparisons to measure progress

Baselining Activity is Mandatory First Step of any Process Improvement Program

(3)

Evolving to an Effective "Process Improvement" Environment



PACKAGING
- Improved Training
- Standards/policies
- Guidebooks

ITERATE

ASSESSING
- Cleanroom Technique
- Inspections process
- Compare test techniques (functional, reading, structural)
- Impact of standards

GOALS

UNDERSTANDING
- What processes used
- Relationship between development parameters
- Error/changes characteristics
- Resource and effort characteristics
- How much software exists/built

EXAMPLES

TIME

Key Ongoing Step is to Understand

(4)

Most Significant Baseline Data

Core Data you Want to Capture

- Insight about the Application Domain
  "Characteristics of the Problem Addressed"

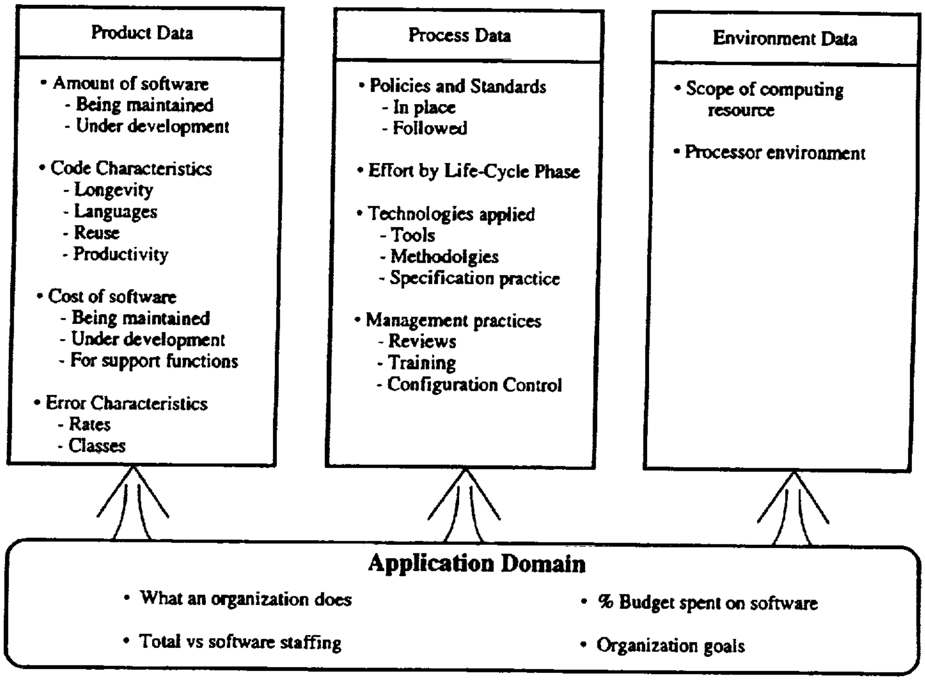
- Product Data
  "End Item Characteristics"

- Process Data
  "How is the End Item Developed and Maintained"

- Environment Insight
  "Supporting Tools and Infrastructure"

(5)

Core Data you Want to Capture



Product Data

- Amount of software
  - Being maintained
  - Under development
- Code Characteristics
  - Longevity
  - Languages
  - Reuse
  - Productivity
- Cost of software
  - Being maintained
  - Under development
  - For support functions
- Error Characteristics
  - Rates
  - Classes

Process Data

- Policies and Standards
  - In place
  - Followed
- Effort by Life-Cycle Phase
- Technologies applied
  - Tools
  - Methodolgies
  - Specification practice
- Management practices
  - Reviews
  - Training
  - Configuration Control

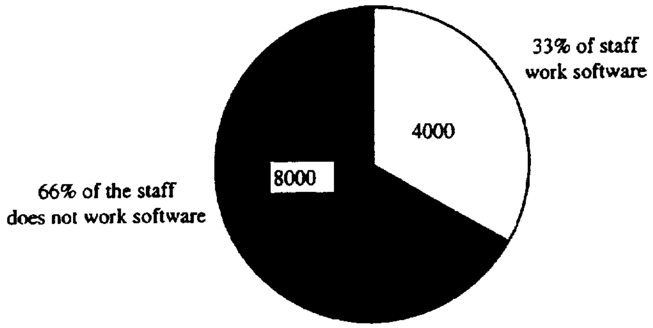
Environment Data

- Scope of computing resource
- Processor environment

Application Domain

- What an organization does
- Total vs software staffing
- % Budget spent on software
- Organization goals

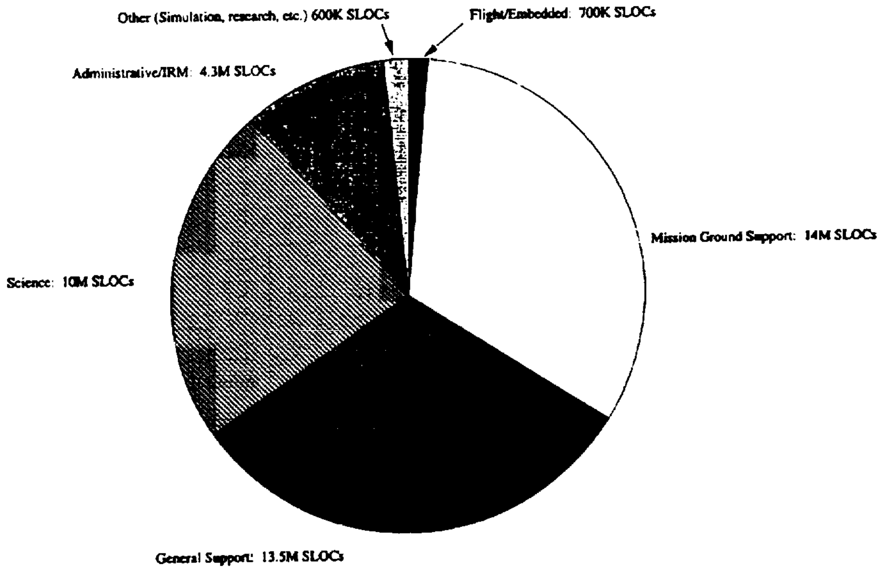
(6)



Total Software Staffing - 12,000
(civil servants & support contractors)

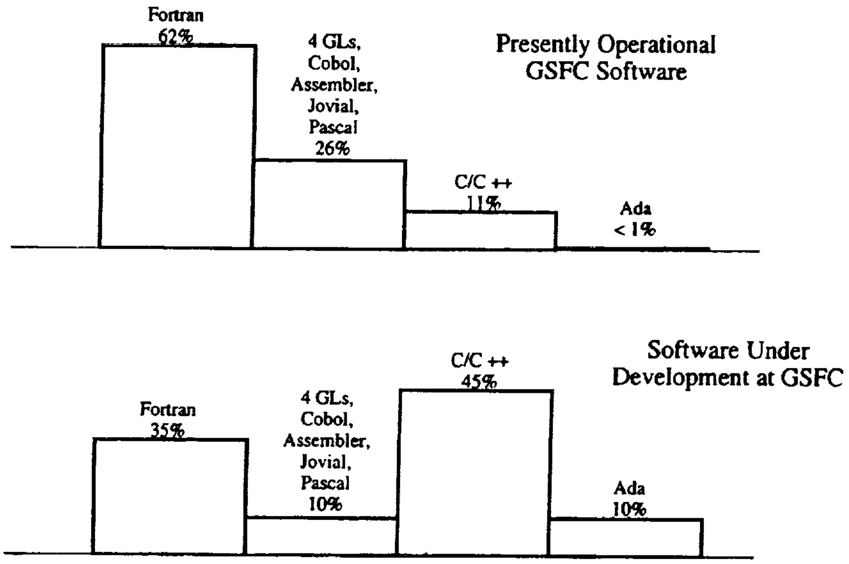
Percentage of Staff Devoted to Software

(7)



GSFC presently has about 43 million source lines of code

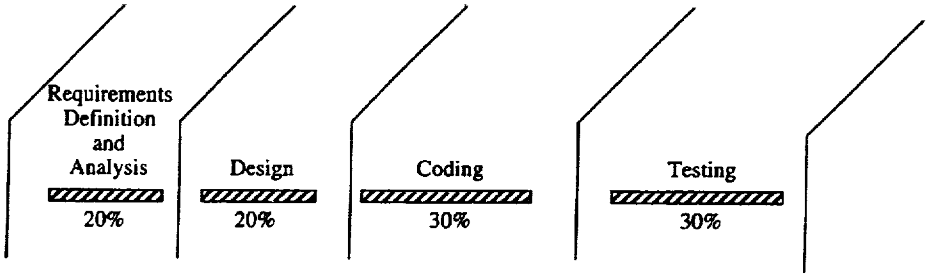
(8)



Language Preferences

(9)

Resource Consumption by Development Phase Typically at GSFC



(10)

What can be Captured?

- Conflicts will exist between data you want to capture and what is available/quantifiable

  Data availability is indicator of relative process maturity of the organization

- Accuracy takes long time and many projects (large, overlapping sample size)

- Data you probably can collect:

  Languages
  Budgets (cost)
  Amounts of software

  This data may be accurate ± 25%

- Less tangible data (depending upon organization's process maturity)

  Effort distribution by phase
  Software longevity
  Error statistics
  Productivity
  Investment in "overhead" functions (QA, CM, documentation, management)

  Our experience is accuracy ± 50%

(11)

Techniques for Establishing the Baseline

- Apply combination of four methods:

  - Administered surveys
  - Informal roundtables
  - Data and documentation review
  - One-on-one interviews

- Survey advice

  1) Must prototype/test the survey instrument
  2) Avoid descriptive entries; Make all responses quantities or checkmarks
  3) Use directed sampling

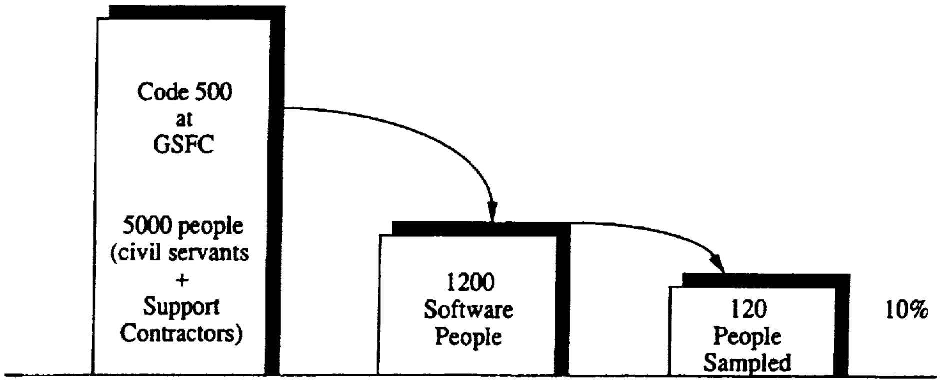
     -- Start with senior managers:
        * organization overview
        * awareness of your activities
        * pointers to "software pockets"
     -- Sample the pockets
     -- Cross-verify

  4) Only one data gatherer or small team (max of 3 people)

  5) Data gatherer(s) must know the organization

(12)

How Many People Should You Talk To?



Code 500
at
GSFC

5000 people
(civil servants
+
Support
Contractors)

1200
Software
People

120
People
Sampled

10%

Sample Size depends upon:

- Organization Uniformity/Heterogeneity
- How many software pockets (Approximately 20 "pockets" in Code 500)

(13)

How Much Will the Baseline Cost?

GSFC Baseline Experience

Survey Development/Testing:	2
Data Gathering (4 methods):	6
Archiving:	2
Data Analysis & Info Extraction:	6
Packaging:	2
	18 person-months

Next Steps: Focus on Most Promising Improvement Areas
1. Training
2. Helpful Standards
Assess/Experiment and Package

(14)

Lessons We Learned

1. Be Objective → Learn, Don't Qualify

2. Gather Perspective → Senior Management
   (Cross verify)
   - Lower Management
   - Developers
   - Testers
   - Quality Assurance

3. Layer your Baselining → Only go as deep as you need

4. Give the Organization Review Opportunity
   (but don't compromise your findings)

5. Use Combination of Methods: Administered Surveys
   - Roundtables
   - Data Review
   - Interviews

(15)

IMPACT OF ADA IN THE FLIGHT DYNAMICS DIVISION: EXCITEMENT AND FRUSTRATION

John Bailey
Software Metrics, Inc.
4345 High Ridge Rd.
Haymarket, VA 22069
703-385-8300

Sharon Waligora
Computer Sciences Corporation
10110 Aerospace Rd.
Lanham-Seabrook, MD 20706
301-794-1744

Mike Stark
NASA/Goddard
Software Engineering Branch
Greenbelt, MD 20771
301-286-5048

ABSTRACT

In 1985, NASA Goddard's Flight Dynamics Division (FDD) began investigating how the Ada language might apply to their software development projects. Although they began cautiously using Ada on only a few pilot projects, they expected that, if the Ada pilots showed promising results, they would fully transition their entire development organization from FORTRAN to Ada within 10 years. However, nearly 9 years later, the FDD still produces 80 percent of its software in FORTRAN, despite positive results on Ada projects. This paper reports preliminary results of an ongoing study, commissioned by the FDD, to quantify the impact of Ada in the FDD, to determine why Ada has not flourished, and to recommend future directions regarding Ada. Project trends in both languages are examined as are external factors and cultural issues that affected the infusion of this technology. This paper is the first public report on the Ada assessment study, which will conclude with a comprehensive final report in mid 1994.

INTRODUCTION

The Flight Dynamics Division (FDD) of the National Aeronautics and Space Administration's Goddard Space Flight Center (NASA Goddard) spends approximately $10M per year developing attitude ground support system (AGSS) and simulator software for scientific satellites. As the prime contractor in this area, Computer Sciences Corporation (CSC) develops most of this software.

Nine years ago, the FDD began investigating Ada and object-oriented design as a means to improve its products and reduce its development costs, with the intention of completely transitioning to an Ada development shop within 10 years. The FDD pursued Ada for reasons similar to those of other forward-looking software organizations in 1985. For example, Ada was considered to be more than just another programming language. Since it embodied several important software engineering principles and contained features to ensure good programming practices, its proper use was expected to lead to advances in the entire software development process. Furthermore, through increased reuse, reliability, and visibility, using Ada was expected to reduce costs, shorten development cycles (project durations), and lead to better and more manageable software products. These goals of greater reuse, lower cost, shorter cycle times, and higher quality became the expressed objectives for learning and using the Ada language at the FDD.

The FDD piloted the use of Ada on its smaller, less risky projects (satellite simulators) that were regularly developed on a VAX minicomputer where a reliable Ada compiler and tool set were available. Early pilots showed promising results and led to a complete transition to Ada for developing simulators in the VAX environment. The results of the early Ada studies and successes have been presented at previous Software Engineering Workshops as well as at Ada- and OOD-related conferences (Reference 1). Unlike those previous papers, which compare the measures of Ada projects with the existing FDD baseline measures in 1985, this paper compares the Ada projects with contemporary FORTRAN projects. The impact of Ada is assessed in the context of the evolving FORTRAN process.

Table 1 presents the high-level characteristics of the Ada and FORTRAN projects included in this study. Notice that all Ada projects were developed on the VAX minicomputers, whereas all the FORTRAN projects were developed on IBM mainframes. Not only are the projects small or, at most, medium-sized by industry standards, but they tend to be short-lived, with operational lives ranging from just a few months to a few years. We have factored these organization-specific software characteristics into the recommendations made in the last section about the future use of Ada and FORTRAN at the FDD.

Table 1. Elements of the FDD Environment

Application	Computing Environment	Language	Typical System Size
Ground Support	IBM mainframes	FORTRAN	200 KSLOC*
Simulators	DEC VAX	Ada	60 KSLOC*

* Thousands of source lines of code

THE ADA EXPERIENCE

Over the past 9 years, the FDD has delivered approximately 1 million lines of Ada code. Figure 1 illustrates the growth of Ada experience in this environment. The curve shows the accumulated amount of code as each project was delivered (the time before the first project delivery is foreshortened for clarity). The scale on this figure is in thousands of physical lines of source code, or KSLOC (i.e., editor lines or carriage returns), and therefore includes comments and blank lines.

Although SLOC is the traditional measure of software size in the FDD, we used statement counts to measure software size for this study. We chose statement counts (i.e., the number of logical statements and declarations) because they are not sensitive to formatting and because they are a more uniform indicator across the two languages both of functionality delivered and development effort expended (Reference 2). The average number of physical lines per statement varied somewhat over the period studied because of changes in programming style. By the last projects studied, the average number of lines per FORTRAN statement had risen from about 2 to more than 3, whereas the number of lines per Ada statement had fallen from over 6 to close to 4.

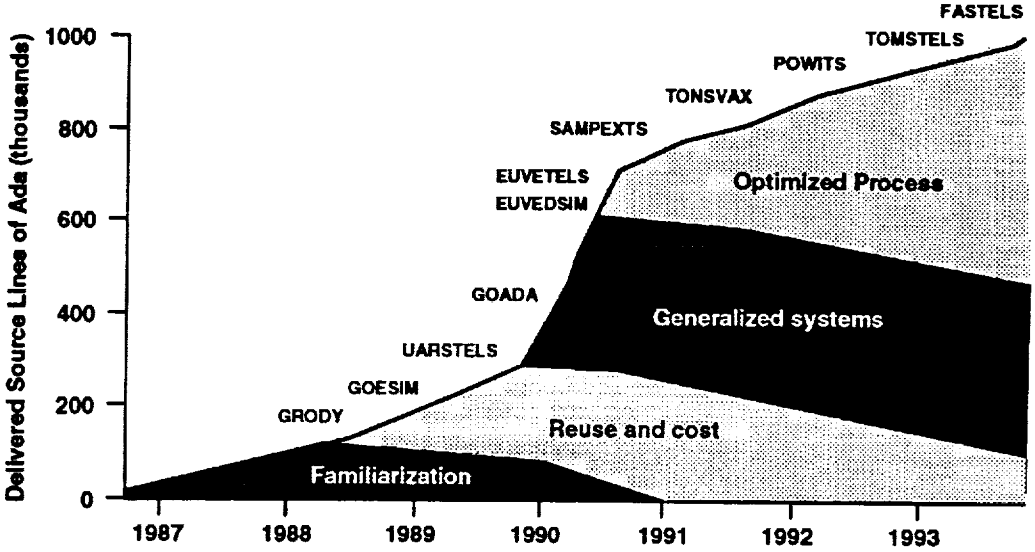


Figure 1. FDD Ada Experience and Focus

The four regions under the curve in Figure 1 give a rough approximation of the evolution of goals and objectives for the study and use of Ada in the FDD. Initially, the main concern was familiarization with the language, although the initial projects also stressed reusability as a major objective. Soon, the focus turned to the structured generalization of systems, and the success of these generalizations led to an overall improvement in the efficiency of the Ada software development process. Recently, there has been an additional focus on optimizing the development process specifically for use with the Ada language. This optimized process has been specified and documented in a recent supplement to the standard software development process guidebook used by the FDD (References 3 and 4). These Ada study goals for reuse, generalization, and process provided the framework for the evolution of the use of Ada in this environment. We discuss them further in the following section when we compare the Ada software with the FORTRAN software developed during the same 9-year period.

We examined the degree of usage of the many Ada language features on the various projects to verify that Ada developers were, in fact, using the full capability of this technology. We found evidence that the use of Ada has matured by looking at changes in the use of the language features over time. Figure 2 shows four views of the evolving language usage. Notice that the use of generics and strong typing increased, whereas the use of tasking decreased along with the average package size. Also, this maturation of language use appears to be leveling off, which suggests that sufficient thought has been given to using the language to enable a standard approach to evolve. These patterns indicate that the FDD developers have become skilled with Ada and have determined an appropriate style and usage for the language in this environment and application domain.

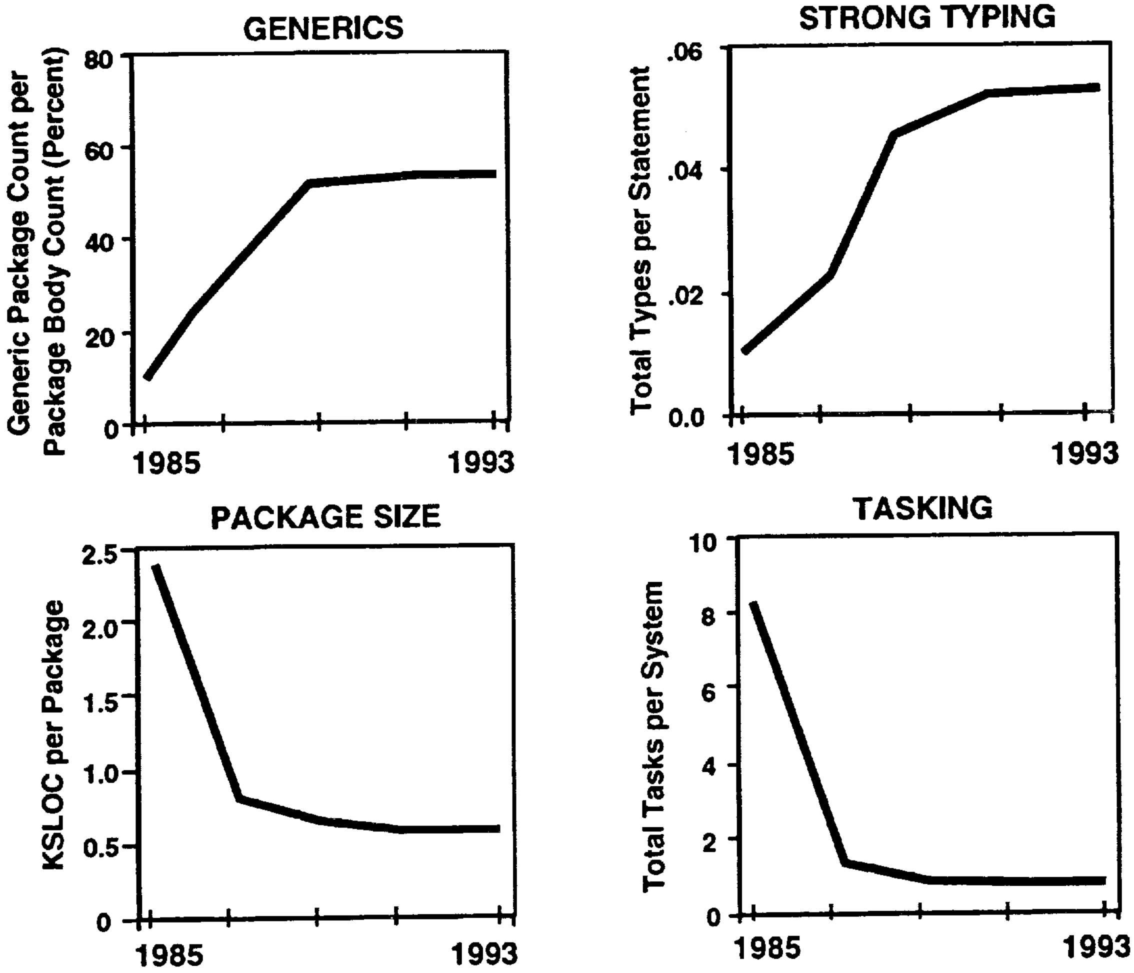


Figure 2. Maturing Use of Ada at the FDD

Comparing Ada and FORTRAN Baselines

The original FDD goals of increased reuse, lower cost (in terms of effort), shorter cycle times, and higher quality can be measured by comparing data from the Ada and FORTRAN projects. Previous papers (References 1 and 5) have documented improvement on Ada projects over the 1985 FORTRAN baseline. But, while the FDD was gradually maturing its use of Ada on the satellite simulators, the FORTRAN process also continued to evolve on the larger, mainframe projects. The following sections compare the evolving Ada and FORTRAN baselines between 1985 and 1993 in each of the initial four goal areas and in terms of the evolving software process. In this way, the improvements seen on Ada projects can be assessed within the context of the evolving FORTRAN baseline.

Reuse

During the 9 years that Ada has been used at the FDD, we have seen considerable improvement in the ability to reuse previously developed software on new projects. Figures 3 and 4 show, for Ada and FORTRAN projects respectively, the percentage of each project that was reused

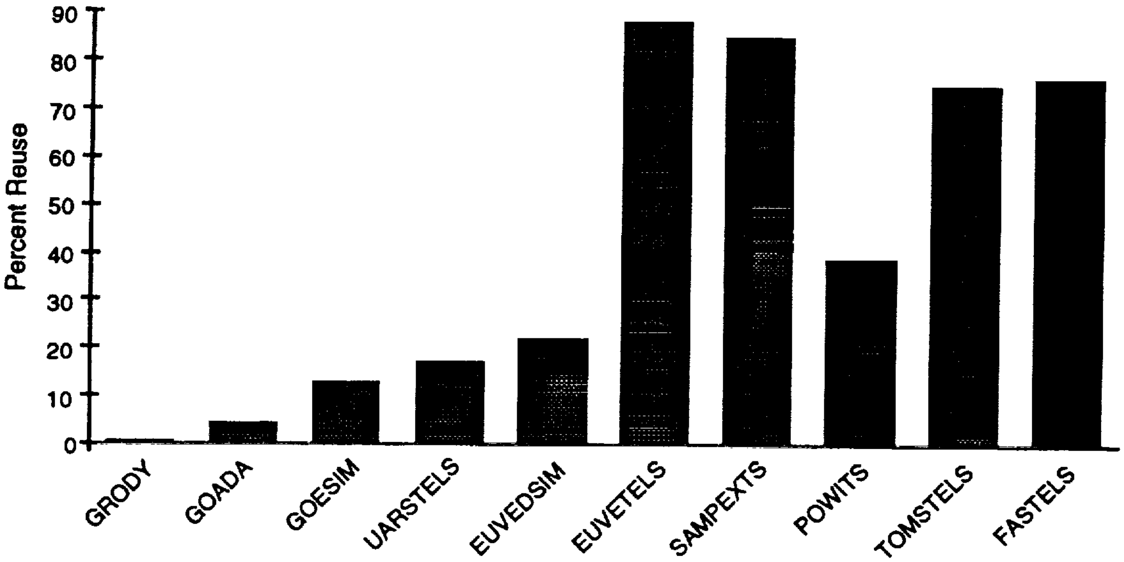


Figure 3. Verbatim Ada Reuse by Project

without change from previous projects. (The minimum unit of reuse is a single compilation unit; no credit is given if only a portion of a compilation unit is reused. The percentages are computed by dividing the total size of the reused compilation units by the total size of the project.) The first breakthrough in high verbatim reuse of Ada occurred in 1989 when a set of generics purposely designed for reuse were demonstrated to be sufficient to construct nearly 90 percent of a new project in the same domain.

The dip in the amount of reuse on the eighth Ada project was caused by a change in the domain that required modification to the Ada generics and additional new code development. Specifically, the original domain where high reuse was achieved was simulation software for

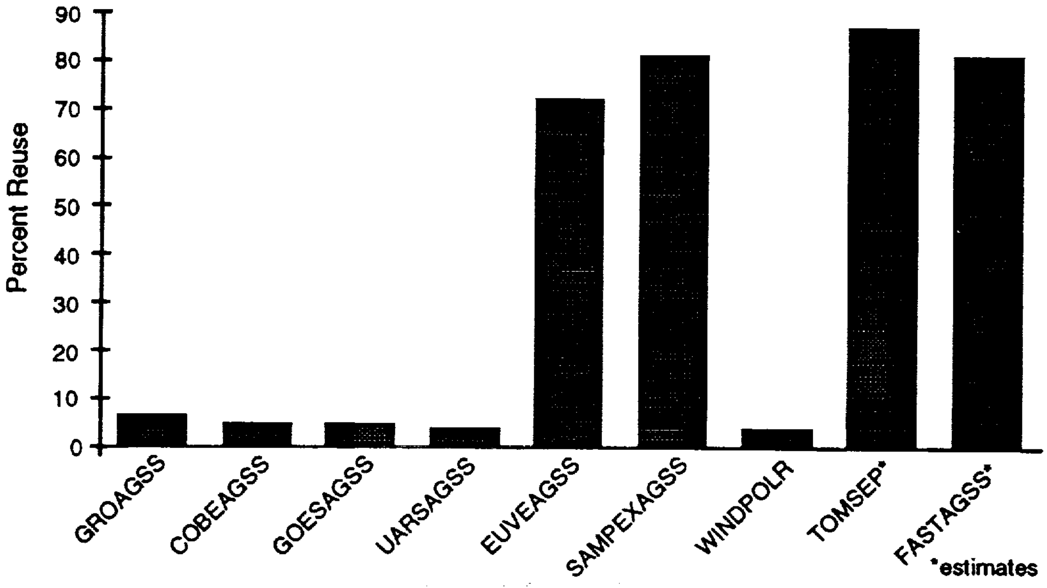


Figure 4. Verbatim FORTRAN Reuse by Project

three-axis stabilized spacecraft. When a spin-stabilized spacecraft was simulated for the first time, a substantial drop occurred in the verbatim reusability of the library generics. This incompatibility was rectified over time so that the generics can now accommodate either a three-axis or a spin-stabilized spacecraft. The slight drop in the most recent examples of reuse to around 80 percent, as compared with the earlier successes with high reuse that were closer to 90 percent, was caused by performance tuning on the latest projects. Performance issues are discussed again in the next major section of this paper.

Figure 4 shows the corresponding picture of verbatim reuse on the FORTRAN projects during the same 9-year period. At its peak, the amount of verbatim reuse achieved was nearly as great as with the reusable Ada generics, and the first successes occurred at nearly the same time as the first highly successful Ada reuse. (The first high-reuse FORTRAN project was the corresponding ground support system for the same satellite mission as the first high-reuse Ada simulator.) Again, a change in domain to spin-stabilized missions caused a drop back to the low levels of reuse observed on the earlier projects in the late 1980s.

The FORTRAN reuse approach differs from that used on the Ada projects, however. Instead of populating a reuse library with a set of generics that can be instantiated with mission-specific parameters, as was done in the Ada projects, the FORTRAN reuse library actually consists of two separate program libraries, together comprising nearly 400,000 source lines of code. One of these libraries is for three-axis stabilized spacecraft and the other is for spin-stabilized spacecraft. Subsystems from the appropriate library must be used in an all-or-nothing fashion. This style of reuse explains why there was a sharper drop in reuse when the change of domains occurred in the FORTRAN projects as compared with the Ada projects. In the Ada projects, it was still possible to reuse a sizable portion of the three-axis library without modification for the first spin-stabilized mission, whereas none of the large FORTRAN library to handle three-axis missions could be reused verbatim for a spinning satellite. The developers again achieved high levels of reuse in their FORTRAN projects by developing a separate complete subsystem library for spin-stabilized spacecraft that was analogous to the three-axis library. Since these two FORTRAN libraries embody over 80 percent of the functionality for any new ground support system, the FDD has since set up a special dedicated team to maintain them. This team is charged with keeping the software up to date with new requirements, while retaining its backward compatibility with previous systems. Estimates of the project-specific effort expended by this team are added to each FORTRAN project that reuses subsystems from the libraries.

An important distinction between the reuse styles adopted for the two languages is that the two FORTRAN libraries must be augmented as needed to handle new missions in their respective domains, whereas the Ada generics form a collective set of smaller components that requires little or no further modification to handle missions in either domain. To avoid the risk of introducing errors for existing clients, the maintainers augment the FORTRAN subsystems as necessary by adding new code rather than by generalizing or modifying their existing code. This causes the FORTRAN libraries to grow over time. In contrast, the Ada developers directly handle the generics needed for each project and further generalize them if necessary. By copying the generics into each project library that needs them, slight changes can also be made to eliminate unnecessary dependencies. The maintenance and configuration control disadvantages of having separate copies of the reusable components in each client project's library are less an issue with the simulators, which have short operational phases, than they would be with the longer-lived AGSS projects.

Cost Reduction

Figures 3 and 4 clearly show the points when dramatic improvements in reuse were achieved in both the Ada and the FORTRAN projects. The first Ada simulator and the first FORTRAN ground system to exhibit high reuse were both written to support the Extreme Ultraviolet Explorer (EUVE) satellite mission. Because of the nature of satellite mission support, the simulator is typically completed first so that it can be used to test the ground system. (In the case of EUVE, the Ada simulator was completed about 4 months ahead of the corresponding FORTRAN ground system.) Since these first successes with reuse almost coincided, and since they are associated with measurable changes in the development approach, we divided the Ada and the FORTRAN project sets into two groups depending on whether or not they were completed before the EUVE experience.

The left-hand side of Figure 5 shows the average costs in hours to deliver a statement of Ada, both before the successes in reuse and since (EUVE and subsequent projects). The figure shows that the productivity of delivering Ada software has doubled since high reuse has been achieved.

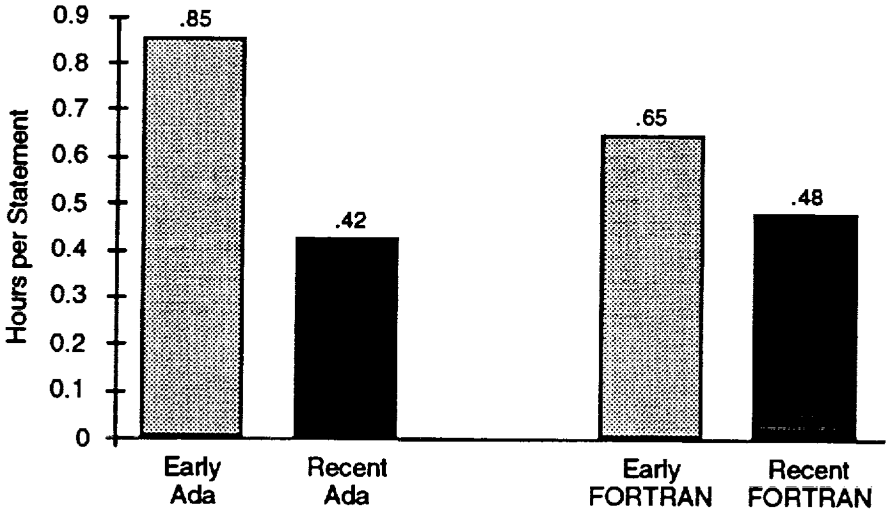


Figure 5. Effort To Deliver One Statement: Ada vs. FORTRAN, Early vs. Recent

The right-hand side of the figure shows the average costs in hours to deliver a statement of FORTRAN before and after the high-reuse process. Again, there is a marked improvement, though not as great a reduction as in the Ada projects. Although we have no hard data to normalize statement counts in Ada with statement counts in FORTRAN, we suspect they are roughly comparable measures of functionality (Reference 2). Assuming comparability, these results lead us to believe that Ada is somewhat more expensive than FORTRAN for conventional software development, but that reuse can lower the cost of an Ada delivery more than it can lower the cost of a FORTRAN delivery. We could conclude that FORTRAN is more cost effective for short-lived software but that Ada should be used for software that is likely to have a longer life through future reuse. We revisit this observation in the final section of the paper.

Shorter Cycle Time

Ada was also expected to lead to shorter cycle times or project durations. Figure 6 shows that this goal was met not only by the Ada projects but also by the FORTRAN projects. Again, the first high-reuse project in each language is the first project of each of the recent sets represented by the right-hand (darker) bars.

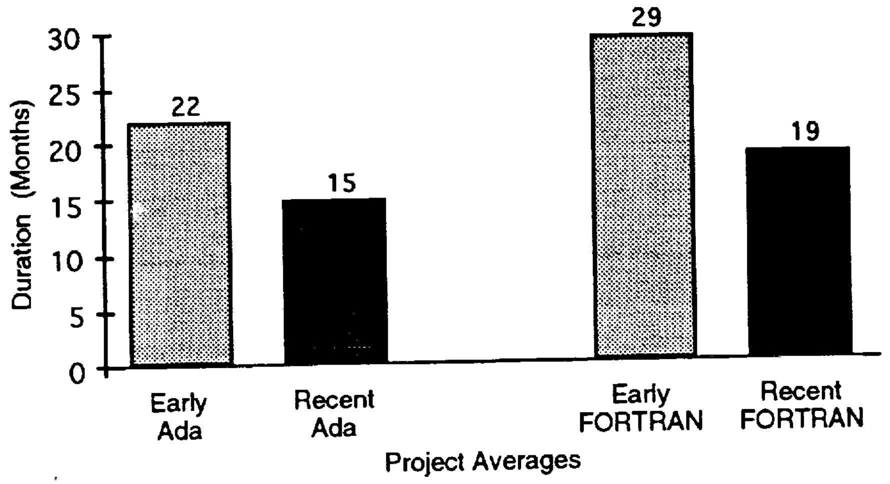


Figure 6. Average Project Duration: Before and After Reuse Process Adoption

The software development process did not change as suddenly as the reuse results, however. In fact, the schedule for the first high-reuse project in each language was more similar to earlier projects than it was to the subsequent high-reuse projects.[1] This is because the overall development process changed only after the EUVE project demonstrated that substantial savings could be achieved through large-scale reuse. To minimize risk, the first high-reuse project in each language was conducted using a more traditional schedule and staffing level. When management was able to observe the potential savings from reuse, procedural and scheduling changes were made to allow an expedited development process whenever high reuse was possible. So, whereas reuse can permit shortened project schedules, it is also necessary to accommodate this different behavior with an appropriately pared-down process. For example, the FDD now specifies a single design review in place of the traditional preliminary and critical design reviews whenever the majority of a new system can be constructed from existing code.

Reliability

The last explicit goal for the planned Ada transition was to increase the quality of the delivered systems. The density of errors discovered during development, measured on all FDD projects, is used to reflect system quality and reliability since quantitative operational data were not

[1] Because they behaved more like traditional projects, one might argue that the first high-reuse project in each language actually belongs to the early project set (represented by the left-hand bars of each pair). This would further accentuate the difference between early and recent projects.

collected.[2] Development-time errors are a useful reflection of quality because they reveal the potential for latent undetected errors and indicate spoilage and rework during development that, in turn, impact productivity and schedule.

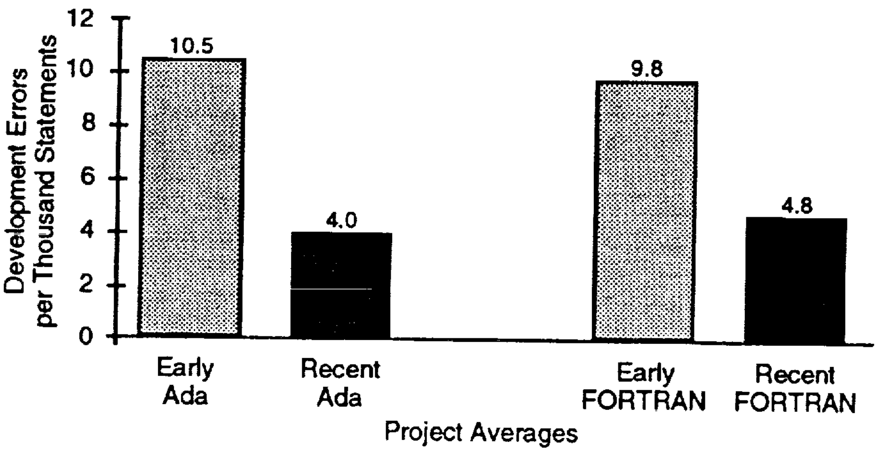


Figure 7. Reduction in Error Density in New or Modified Code: Ada and FORTRAN

The number of errors discovered per thousand statements of new and modified software before delivery is shown in Figure 7. Since the densities shown are based on only the new and modified code (verbatim reused code was not included in the denominator), we do not attribute these reductions to reuse. Instead, we attribute the reduced error rate to improvements in the development process that were instituted on all FDD projects during this period. These improvements included the use of object-oriented or encapsulated designs and the use of structured code reading and inspections. The fact that these process improvements were applied to projects in both languages is reflected by the similarity in the error-density reductions observed.

Process

An evolving development process was cited in the preceding discussion as being the reason for improvements in schedule and quality. We characterized the process by examining the distribution of effort across the various software development activities performed. The activity distributions shown in Figure 8 provide evidence that the more recent projects were conducted using a different process than the early projects. The figure shows the average number of staff-hours per project consumed by each of the four defined activities for software projects at the FDD. The dark bars for each activity show the averages for the first five Ada simulator projects, and the light bars show the average effort per activity for the five subsequent Ada simulators that achieved higher levels of reuse. (To fairly average the efforts for each activity among projects of varying sizes, the activity data for each project were first normalized by project size.) The savings exhibited for the later set of projects is due not simply to reuse alone

[2] So far, operational reliability, in terms of mean time to failure, has been adequate and it has therefore never become a measurement or improvement goal. The amount of maintenance effort expended on the operational systems is now being tracked, however.

but to the process change adopted to accommodate that reuse. The evidence for this is that the EUVE simulator (the sixth Ada project, see Figure 3) was the first to achieve a high level of reuse but, because it used a traditional process, its individual profile more closely resembled the earlier low-reuse group than the later high-reuse group.

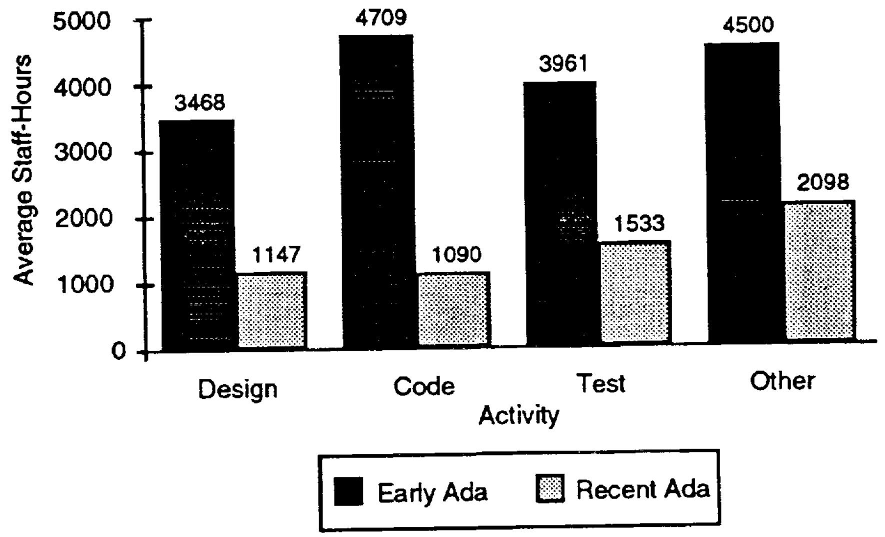


Figure 8. Average Effort by Activity: Early vs. Recent Ada Projects

Figure 9 shows the average effort by activity for the FORTRAN projects that were completed during the same period. Again, the effort magnitudes are normalized, and the projects are divided into an earlier group of lower reuse projects and a more recent group of higher reuse projects. As with Ada, a reduction in effort is shown for each activity when comparing low reuse with high reuse, although the net reduction is less in FORTRAN. Unlike the Ada results, however, most of the reduction occurs in the coding activity instead of being spread more evenly across every activity.

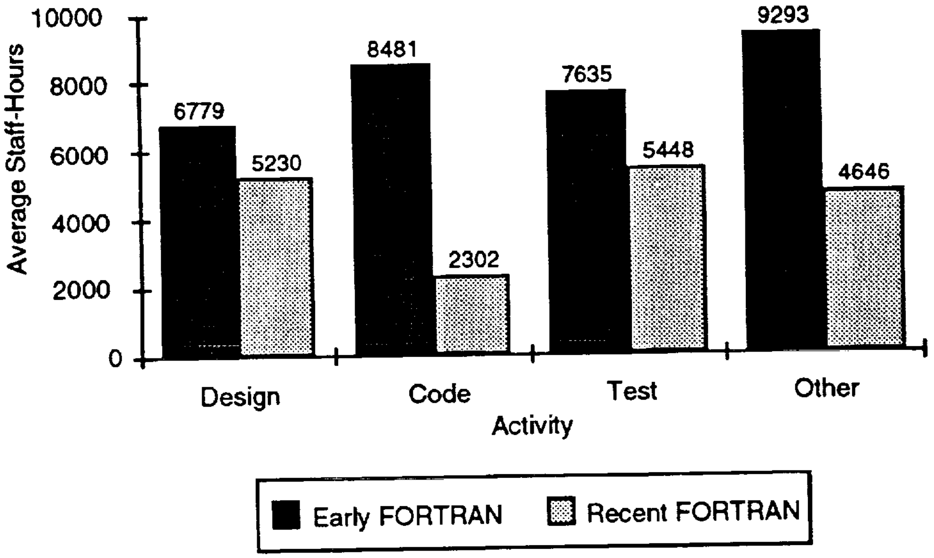


Figure 9. Average Effort by Activity: Early vs. Recent FORTRAN Projects

Notice that the shape of the activity distribution of the early projects in the FORTRAN set is virtually identical to the activity distribution of the early projects in the Ada set (the dark bars in Figures 8 and 9). However, as noted, the distributions for the recent, high-reuse projects differ between the languages. This suggests that the Ada and FORTRAN processes have each evolved in a different way even though they both had a common point of departure. The main lessons from this illustration are that the software process matured and improved during the time that Ada was used and that this evolution affected both the Ada work and the FORTRAN work, although in different ways. This process change is responsible for the reduced schedules shown in Figure 6. Also, the differing net reductions in effort reveal why Figure 5 showed a greater drop in cost for the Ada projects than for the FORTRAN projects when the benefits of reuse started to accrue.

Summary of the Comparisons

Quantitative data over the past 9 years show clear improvements attributable to the use of the Ada language in all of the initially specified goal areas. Many of these improvements were directly related to the increase in reuse. Interestingly, the FORTRAN systems showed comparable results over the same period, which leads to two possible interpretations. We could claim that because Ada did not provide a substantial improvement over FORTRAN, the FDD's continued involvement in the new language is unnecessary. However, Ada quickly did at least as well as the established language in the domain, and because it is a more modern language than FORTRAN, we could claim that Ada should be adopted for all future FDD development. Regardless of the interpretation chosen, it is safe to say that we found no quantitative evidence to indicate that Ada could not be used successfully on all FDD projects. The next section presents our additional findings beyond these quantitative results that affected the overall acceptance of Ada at the FDD.

Unforeseen Factors Impeding the Adoption of Ada

Given the encouraging Ada project results and the motivations of the forward-thinking managers who originally set out to transition to Ada, why hasn't the FDD successfully adopted Ada? As stated earlier, only a fraction of all new software developed at the FDD is in the Ada language. Our investigation discovered several unforeseen factors that have impeded the FDD's transition to Ada. These factors were poor initial system performance, the lack of adequate tool support, and developer bias. This section discusses these factors and their impact on the transition to Ada.

Performance

System performance was not an explicitly stated goal for the programs developed in Ada, but it turned out to be a major issue. By 1985, the programmers in the FDD had achieved such proficiency with FORTRAN software design and implementation that even the most complex flight dynamics systems performed adequately without paying any special attention to performance. Thus, performance had become an implicit consideration and was not addressed in software requirements, designs, or test plans.

Figure 10 depicts the relative response times of the delivered simulators between 1984 and 1993. A smaller response time indicates better performance. The figure reveals that the first

Ada simulator performed very poorly compared with predecessor FORTRAN simulators. Because Ada language benchmarks had shown that Ada executed as fast as equivalent FORTRAN programs and because performance was not an explicit goal, developers of the first Ada project paid little attention to performance. Instead they focused on learning the language and developing reusable software. It should come as no surprise that novice users of this fairly complex language did not produce an optimum design or implementation. But, because this system was delivered for operational use, the FDD users' first encounter with an Ada system was negative. This feeling was compounded by the fact that, because of scaled-down processing requirements, the FORTRAN simulator delivered immediately before the first Ada simulator was the fastest simulator ever delivered.

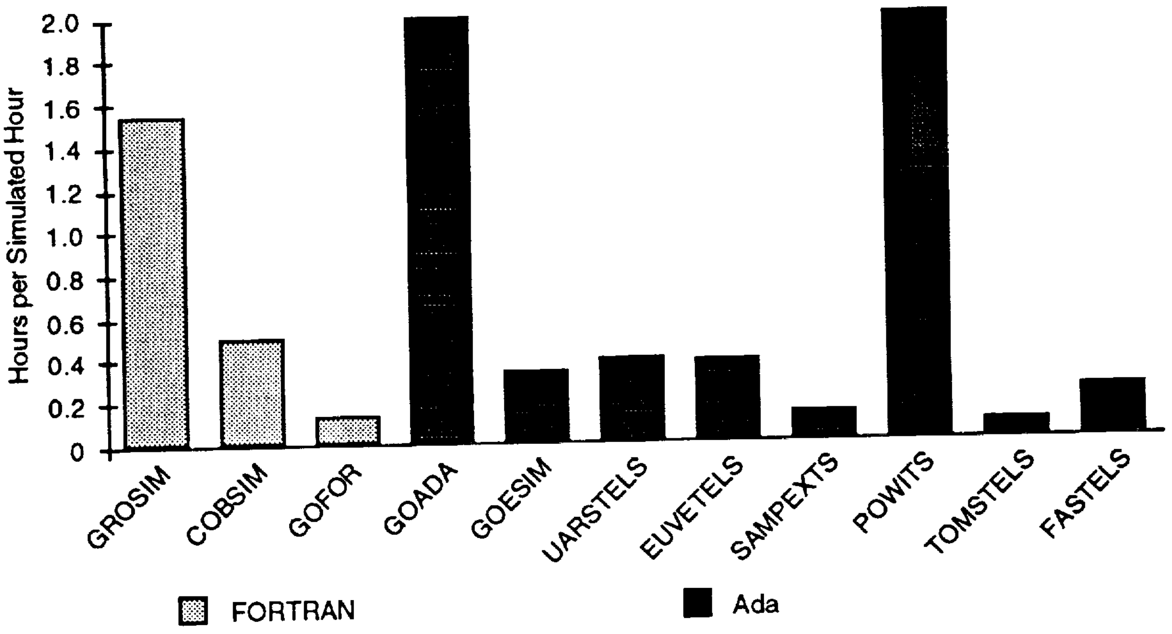


Figure 10. Relative Response Times for Ada and FORTRAN Simulators

In 1990, the FDD conducted a performance study (Reference 6) to determine why the Ada systems executed so much more slowly than the fastest FORTRAN system. The study discovered that some of the coding techniques practiced in FORTRAN to achieve high efficiency actually worked against efficiency in Ada and that some of the data structures around which the designs were built were handled very inefficiently by the DEC Ada compiler. The study resulted in a set of Ada efficiency guidelines (Reference 7) for both design and code that is now being followed for all new Ada systems. Interestingly, in order to follow those guidelines, the last two Ada projects in Figure 10 had to forgo a certain amount of reuse. (The slower POWITS simulator was completed before these guidelines were available and also had considerably more complex processing requirements.) As shown in the figure, the typical Ada simulator now performs better than most of the earlier FORTRAN simulators. However, first impressions are very important, and the perception of many of the FDD programmers and users is that Ada still has performance problems and that systems demanding high performance should not be implemented in Ada.

Ada Development Environments

Finding adequate vendor tools to support Ada development in the FDD was a major obstacle. In 1985, when the FDD began its work with Ada, most computer vendors were actively developing Ada compilers and development environments. The FDD believed that vendor tools would be widely available within a few years. But, consistently usable Ada development environments and reliable Ada compilers did not become available across the platforms used to develop and execute FDD software systems.

All the Ada projects included in this study were developed using DEC Ada on VAX minicomputers that were rated by FDD developers as having a good set of Ada development tools to facilitate the development process. However, 80 percent of the software developed in the FDD must execute on the standard operational environment, which is an IBM mainframe. And traditionally, FDD systems have been developed on their target platforms. Unfortunately, an adequate Ada development environment for the IBM mainframe was never found.

The FDD conducted several compiler evaluations and a portability study between 1989 and 1992, all of which declared the IBM mainframe environments unfit for FDD software development. In 1989, the FDD evaluated three compilers (Reference 8) and selected one for purchase and further study. Somewhat discouraged by this study which rated the best compiler as having only marginal performance for flight dynamics computations and no development tools, the FDD investigated an alternative approach. Since Ada was touted to be highly portable, they conducted a portability study to determine whether FDD systems could be developed in Ada on the VAX and then transported to the mainframe for operational use. This study (Reference 9) ported one of the existing operational Ada simulators from the VAX to the IBM mainframe using the Alsys IBM Ada compiler, version 3.6. The study found that relatively few software changes were required and that the resulting system performed adequately on the mainframe, but that rehosting was extremely difficult because of compiler problems and the lack of diagnostic tools and library management tools. Although rehosting the system required only a small amount of effort, it took nearly as much calendar time as was needed to develop the system from scratch.

In the fall of 1992, the FDD again conducted a compiler evaluation on what were supposed to be greatly improved products. This study (References 10 and 11) selected a different compiler than the earlier study, using the ported simulator as one of its benchmarks. Although the chosen compiler performed better than other candidates and was accompanied by a limited tool set, the study warned against using it to develop real-time or large-scale FDD systems because of its inefficient compiling and binding performance, immature error handling, and poor performance of file input/output. Finally, late in 1993, the FDD achieved some limited success developing a small utility in Ada (the FAST General Torquer Command Utility) on an IBM RS-6000 workstation and porting the software to the mainframe.

Figure 11 (a simplification of Figure 2) shows a sharp decline in the amount of Ada development late in 1990. It was at this point that the FDD had planned to begin developing parts of the larger ground support systems in Ada on the mainframes. But results of the early Ada compiler evaluation and the portability study made it clear that developing on, or even developing elsewhere and porting to, the mainframes was not feasible. Thus, continued growth in Ada stalled.

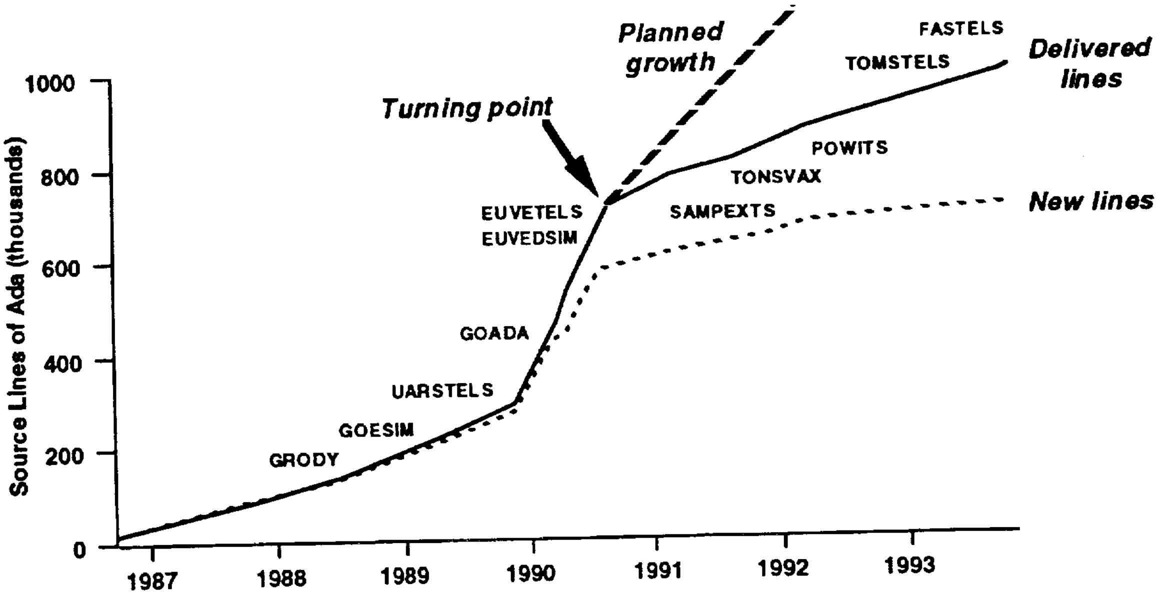


Figure 11. Drop in Ada Growth Coincides With Trouble Migrating to Mainframes

At the same time, the FDD's simulation requirements changed, reducing the number of simulators required to support each spacecraft mission from two to just one.[3] This change resulted in a reduction in the amount of software targeted to be developed in Ada. The net result was a significant reduction, instead of an increase, in the rate of Ada software delivery. The drop in Ada development is even more dramatic when you eliminate the amount of reused software and consider only the investment in new Ada code. The flatter dashed line below the curve for cumulative delivered size in Figure 11 shows only the number of new and modified lines being developed.

The unavailability of an adequate Ada development environment on the IBM mainframe was clearly a significant stumbling block to the FDD in its transition to Ada. We feel that as long as the mainframes remain the principal operational environment at the FDD and as long as mainframe development or deployment of Ada systems remains awkward, there appears to be no straightforward way to increase the use of Ada within the Division.

In addition to its disappointment with the mainframe development environments, the FDD also experienced only limited success in using Ada to develop an embedded system. As part of a separate research and development effort, the FDD developed an embedded application on a Texas Instruments 1750A machine using the Tartan Ada compiler. Unfortunately, the two-vendor situation led to interface problems between the hardware and software; the lack of diagnostic tools contributed to insoluble problems that resulted in an end product with reduced capability. This experience contributed to the general feeling among the FDD developers that there was not yet a satisfactory level of vendor support for Ada development.

[3] Before this point, both a telemetry simulator and a dynamics simulator were developed for each mission. Since 1990, only an enhanced telemetry simulator has been required.

Developer Perspective

As with any technology infusion effort, developers' bias for or against a technology can significantly facilitate or impede the transition. To get a reading on the current perspective of the developer community, we identified and interviewed 35 FDD developers who have been trained in or have developed systems in Ada. Each developer was asked which language they would choose for the next simulator project and which language they would choose for the next ground support system and why. Figure 12 shows their responses.

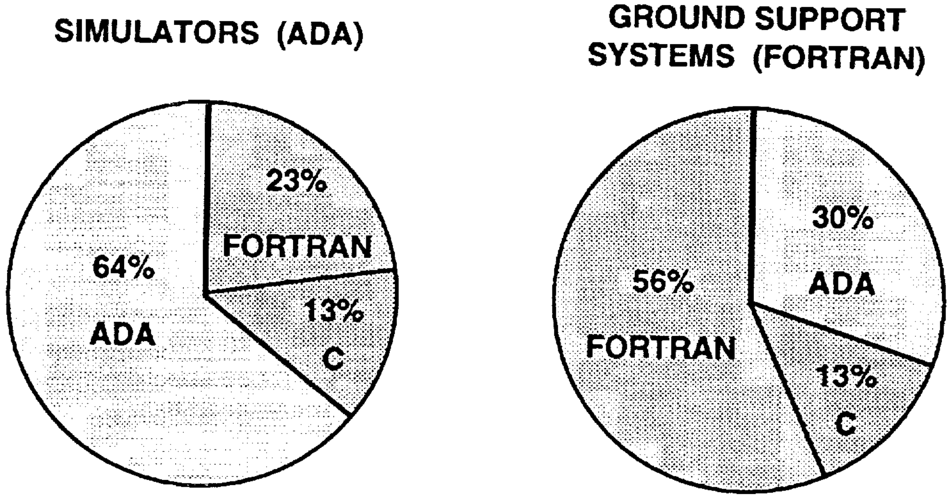


Figure 12. Developer Preferences for Programming Language

Most agreed that Ada should be used for the next simulator, whereas FORTRAN should be used for the next ground support system, citing the availability of reusable components and architectures as the deciding factors. But, also notice that 25 to 30 percent of the developers seemed to have a language bias because they chose the language not customarily used for each type of application. The 23 percent who preferred to use FORTRAN for simulators cited the complexity of the Ada language and poor performance as reasons to abandon Ada, while the 30 percent who preferred to use Ada to build the next ground support system felt that Ada was a better language for building larger systems.

Nearly all the developers pointed out that adequate tools are essential for efficient and accurate development using Ada, whereas FORTRAN development can be accomplished with little or no tool support. However, the two most enthusiastic Ada developers felt that they were sufficiently proficient with the language to overcome the lack of tools on the mainframe computers. Interestingly, at the other extreme, several the developers did not care one way or the other about which language they used for software development; two developers specifically commented that "Ada is just another language."

The two significant minority groups who were the most opinionated about language use, one in favor of Ada and the other opposed, have been fairly vocal and forthcoming with their views over the past several years. Thus, there may be an observable effect from these biases on the remaining group of developers who have not yet been trained in or exposed to Ada. We plan to investigate these possible effects through additional developer surveys before this study is concluded.

CONCLUSIONS AND RECOMMENDATIONS

In 1985 Ada was arguably more than just another programming language. However, by exposing the organization to the concepts of information hiding, modularity, and packaging for reuse, that which was "more than a language" was adopted, to the extent possible, by the FORTRAN developers as well as by the Ada developers. We hypothesize, and in fact have anecdotal evidence to support the theory, that Ada served to catalyze several language-independent advances in the ways in which software is structured and developed across the organization, and that these benefits have been institutionalized by process improvements.[4] Because many of the intended benefits of Ada have already accrued at the FDD and because the mainframe obstacle continues to hamper the complete adoption of Ada, we recommend that the FDD not mandate the use of Ada for all software development at the FDD.

However, we also recommend that the FDD continue to use Ada wherever there is a clear or anticipated advantage. This would include not only the continued use of Ada on satellite simulators but also the use of Ada on portions of any other projects that are expected to be long-lived and can be developed and deployed on an Ada-capable platform. As the FDD migrates away from mainframes and toward workstations, this will be an increasingly large segment of the software developed. Over the long term, Ada is likely to be a good candidate for future versions of the large reusable software libraries that are currently written in FORTRAN and maintained by a separate group of experts who are constantly augmenting the code's function in order to keep up with the needs of the client projects. Ada can be used to implement those subsystems, along with many other basic domain functions, as sets of separable and more maintainable abstractions, which would eliminate the high coupling found in the FORTRAN versions.

Finally, it is valuable to examine some of the original objectives behind the development of the Ada language. One of the major motivations for the DoD to develop and adopt Ada was to provide a common language across many projects, thus enabling the portability of programs, tools, and personnel. However, such commonality is not as important at the FDD where most of the software is already written in a single high-order language and is both developed and operated on a single platform by a seasoned team with low turnover. Besides providing the DoD with a common language, Ada was specifically designed to be beneficial for large system development and to be cost effective for systems that must be maintained over long periods of time. Neither of these situations describe the context in which Ada has been used at the FDD. If the FDD Ada projects had been large (closer to a million lines of code, for example) then we suspect that the use of Ada would have been a more important factor. Also, if the systems were long-lived, with maintenance cycles that were substantially longer than the development cycles, or even if the longer-lived ground support systems had been able to use Ada, we suspect that we would have observed more of the advantages of Ada that would not have been mimicked by the FORTRAN projects.

[4] These opinions were commonly held by the project managers of several of the Ada and FORTRAN projects included in this study.

THE AUTHORS

John Bailey is an independent consultant in the areas of software measurement and the Ada language. Sharon Waligora is a member of the Software Engineering Laboratory at Computer Sciences Corporation and has experience managing and working on the Flight Dynamics Division's software projects. Mike Stark is an employee of the Flight Dynamics Division at Goddard and also has experience leading and participating in many of the FDD Ada and FORTRAN projects.

REFERENCES

1. NASA/GSFC Software Engineering Laboratory, SEL-82-1206, Annotated Bibliography of Software Engineering Laboratory Literature, L. Morusiewicz and J. Valett, November 1993.

2. Institute for Defense Analysis (IDA), IDA Paper P-2899, "*Comparing Ada and FORTRAN Lines of Code: Some Experimental Results*," T. Frazier, J. Bailey, M. Young, November 1993.

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

4. ___, SEL-81-305SP1, *Ada Developers' Supplement to the Recommended Approach*, L. Landis, R. Kester, June 1992.

5. ___, SEL-91-006, *Proceedings of the Sixteenth Annual Software Engineering Workshop*, "Experiments in Software Engineering Technology," F. McGarry and S. Waligora, December 1991.

6. ___, SEL-91-003, *Ada Performance Study Report*, E. Booth and M. Stark, July 1991.

7. Goddard Space Flight Center (GSFC), Flight Dynamics Division, 552-FDD-91/068R0UD0, *Ada Efficiency Guide*, E. Booth (CSC), prepared by Computer Sciences Corporation, August 1992.

8. ___, "*Ada Compilers on the IBM Mainframe (NAS8040) Evaluation Report*, L. Jun, January 1989.

9. NASA/GSFC Software Engineering Laboratory, SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory*, L. Jun and S. Valett, June 1990.

10. Goddard Space Flight Center (GSFC), Flight Dynamics Division, "*IBM Ada/370 (Release 2.0) Compiler Evaluation Report*," L. Jun, September 1992.

11. ___, "*Intermetrics MVS/Ada Version 8.0 Compiler Evaluation Report*," L. Jun, October 1992.

Impact of Ada in the Flight Dynamics Environment: Excitement and Frustration

December 2, 1993

John Bailey, *Software Metrics, Inc.*

Mike Stark, *NASA/Goddard*
Sharon Waligora, *Computer Sciences Corporation*

SMI

Independent Assessment

- Flight Dynamics Division (FDD) began investigating Ada in 1985
- Expected to transition completely to Ada within 10 years
- Why is only 15% of new code being written in Ada today?
- What is the future of Ada in the FDD?
  - Mandate?
  - Abandon?
  - Change expectations?
  - Study further?

SMI

Why Use Ada in FDD?

"Ada is more than just another language"

- **Would lead to a major cultural change**
- **Would drive an integrated well-defined software engineering process**
- **Would help us build better products**
  - **Reduce life-cycle cost**
  - **Shorten project duration**
  - **Reduce number of errors**
  - **Increase manageability of software**

SMI

10015848-prese 3

FDD Environment

Organization Staff Level > 250

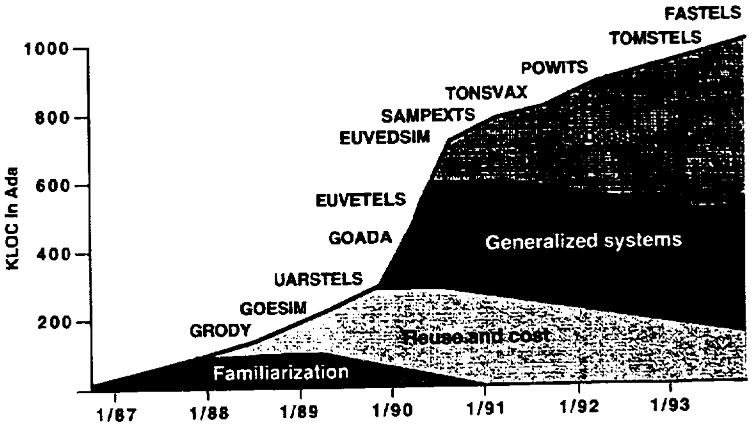
Characteristics	Applications	
	Mission Support	Simulators
Computing Environment	IBM mainframes	VAX
Language	FORTRAN	Ada
Typical System Size	200 KLOC	60 KLOC
Percent of Software	80%	15%

SMI

10015848-prese 4

Focus of Ada Study Evolved

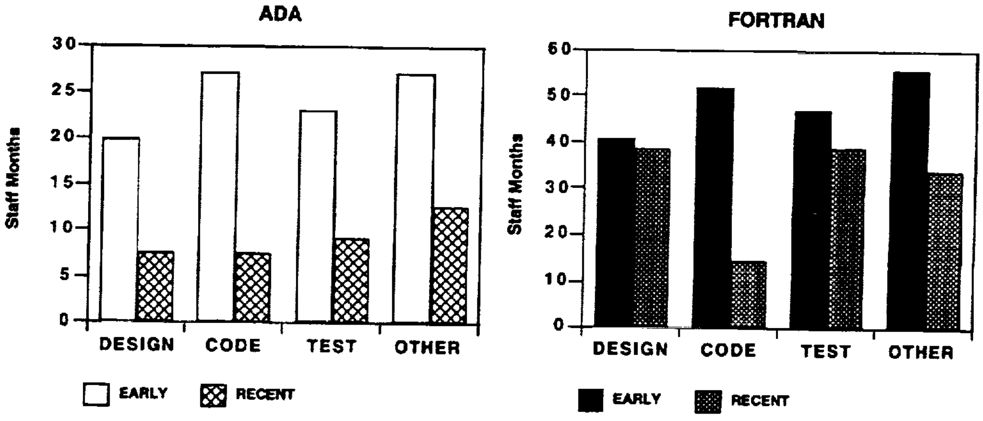
FDD Ada Experience



The Excitement: Data Show Promise

- Process Improvement
- Product Improvement
  - Reuse
  - Cost
  - Project duration
  - Reliability

Evidence of Process Change Over Time

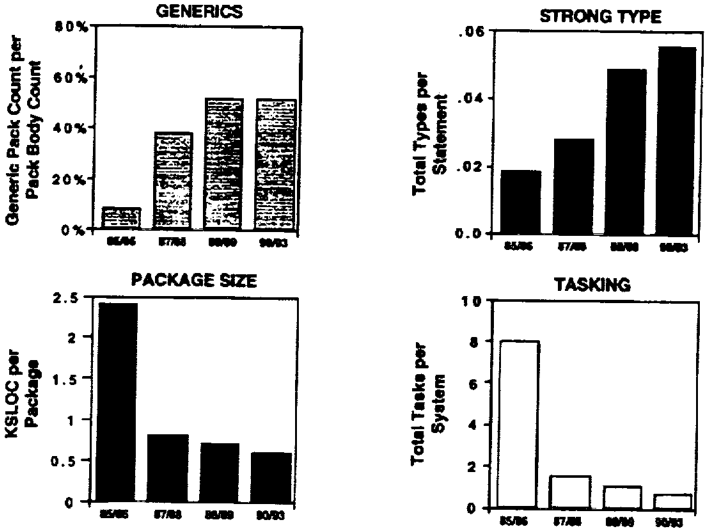


Activity distribution shows process differences

SMI

10015848-prese 7

Maturing Use of Ada

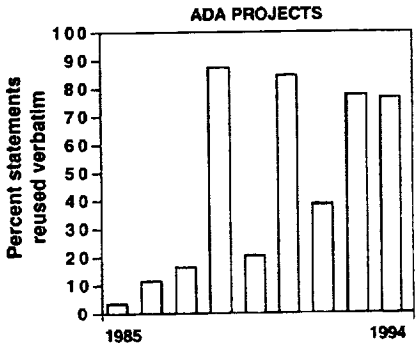


Use of new features stabilized after experience with domain

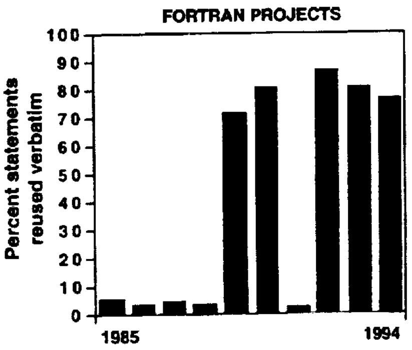
SMI

10015848-prese 8

High Verbatim Reuse Achieved in Both Languages

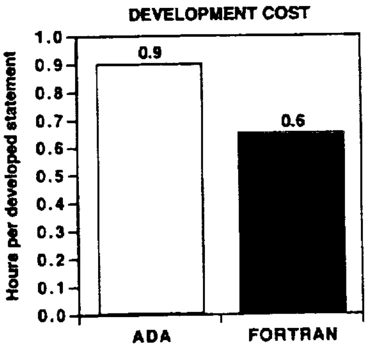


Ada approach uses generics parameterized for mission-specific functionality

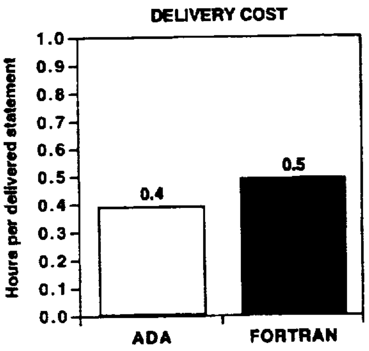


FORTRAN approach uses separately maintained utilities, with mission-specific functionality added as needed

Cost To Develop and Deliver

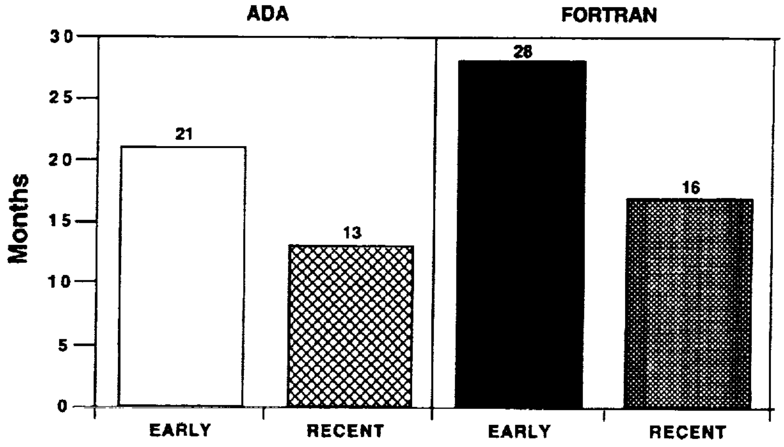


Cost to develop a statement of Ada is higher



Cost to deliver a statement of Ada is lower

Reuse Shortens Project Duration

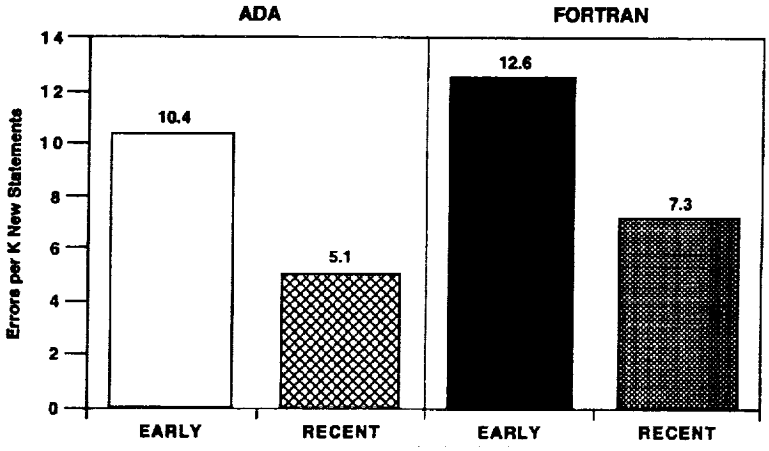


Both Ada and FORTRAN show 38% reduction due to reuse

SMI

10015848-prese 11

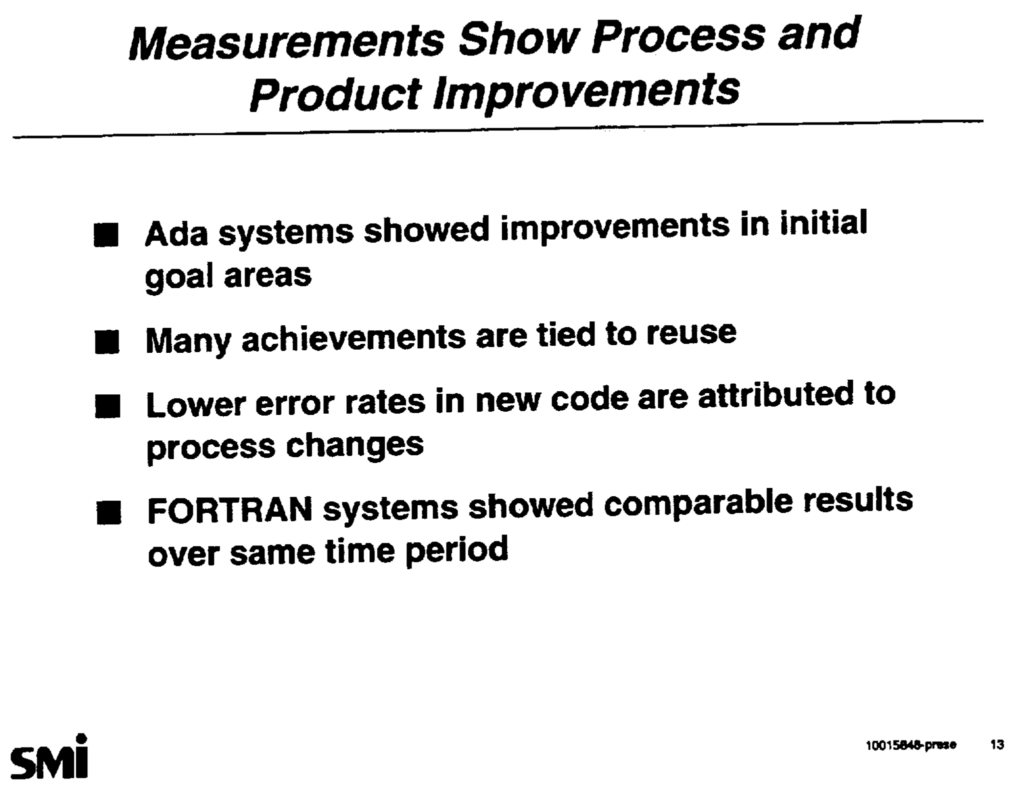
Reliability of Systems Increased



Lower error rates are observed for both languages

SMI

10015848-prese 12

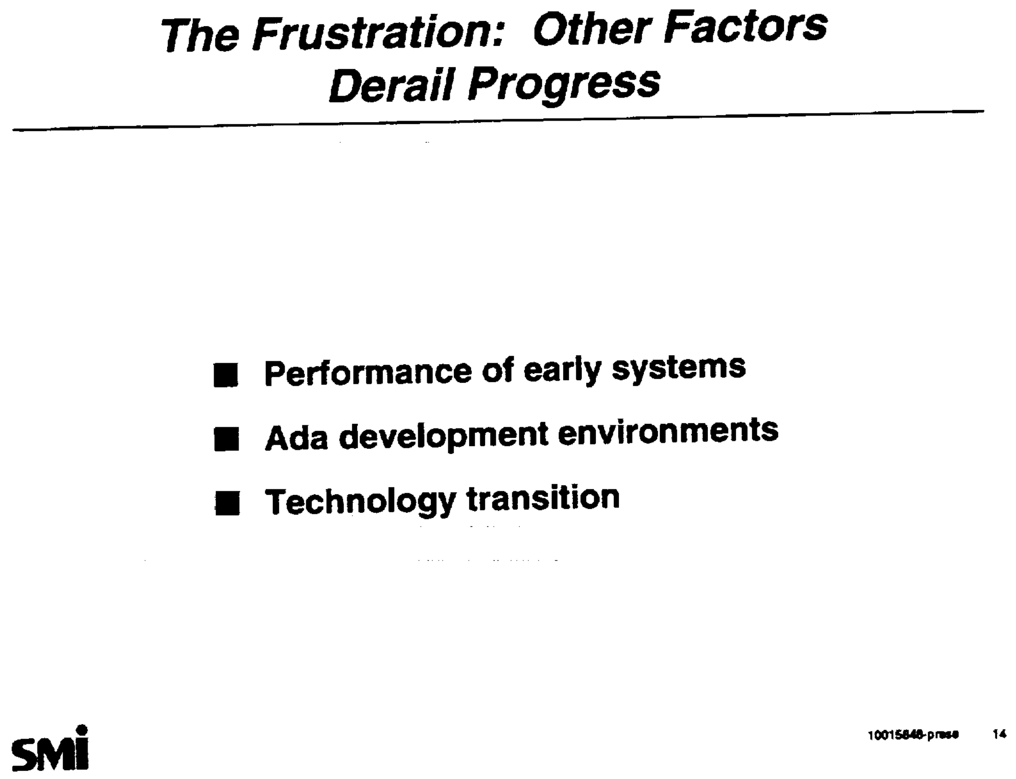


Measurements Show Process and Product Improvements

- Ada systems showed improvements in initial goal areas
- Many achievements are tied to reuse
- Lower error rates in new code are attributed to process changes
- FORTRAN systems showed comparable results over same time period

SMI

10015848-prese 13



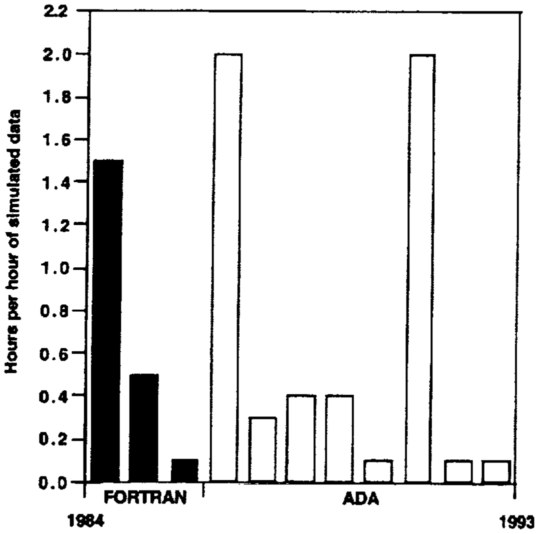
The Frustration: Other Factors Derail Progress

- Performance of early systems
- Ada development environments
- Technology transition

SMI

10015848-prese 14

Performance: First Impressions Count



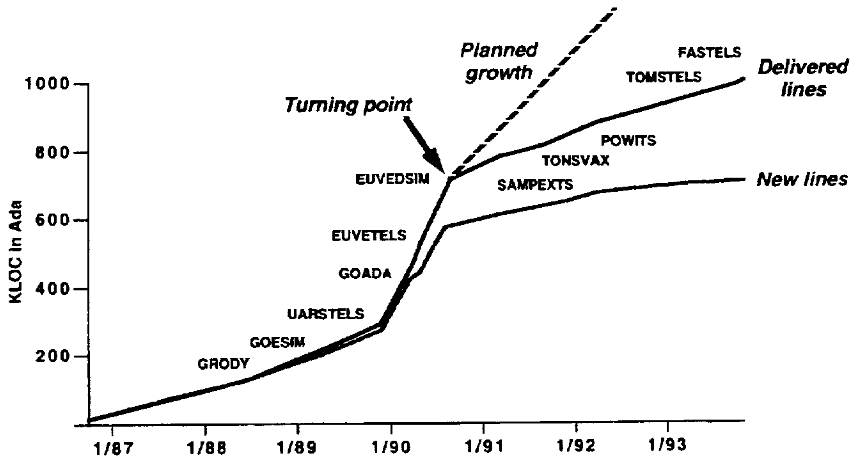
Weak performance of early simulators gave Ada a bad reputation

Ada Development Environments

- VAX Ada
  - Adequate performance
  - Good tools
- IBM mainframe
  - Very poor usability
  - Limited set of immature tools
- Tartan/1750A
  - Separate vendors for hardware and software
  - Major hardware/software interface problems
  - Required assembler-level debugging

Limited vendor support for Ada environments hampered efforts

Mainframe Environment Problems Deterred Growth

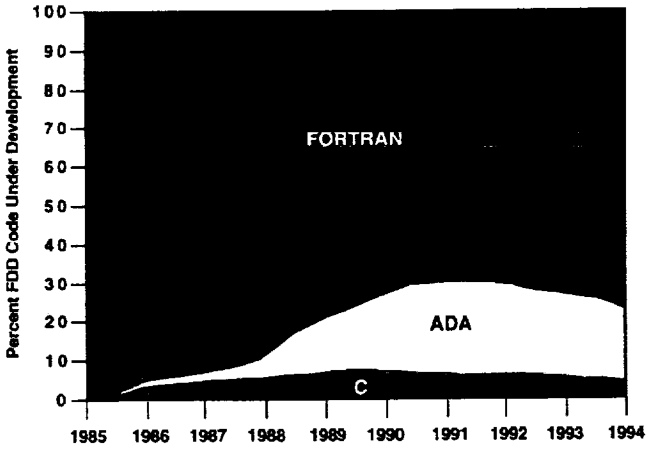


Even today, Ada cannot be used on the FDD (mainframe) operational environment

SMI

10015848-prese 17

Technology Transition to Ada Has Slowed

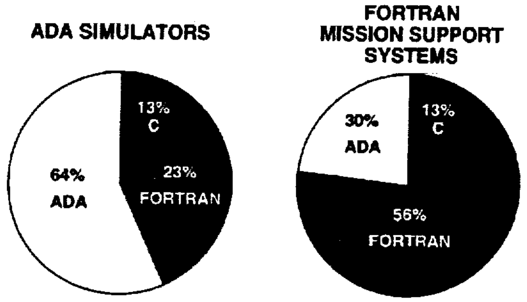


Use of Ada drops as a result of deterrents

SMI

10015848-prese 18

Developers' Preferences



- Interviewed 35 developers with Ada exposure
- Reuse was main driver for language preference
- Ada has advantages but requires more tool support
- "Ada is just another language"

SMI

Process or Language?

- Data show few quantitative differences between Ada and FORTRAN
- Process is more significant factor than language
- Ada concepts (generalization, OOD, domain analysis, information hiding) lay foundation for broad process improvements
- Structured environment and strong process management institutionalize improvement

SMI

Future of Ada at FDD

omit

- Ada should not be mandated at the FDD
  - No pressing need for common language
- Ada should be used as any other method or tool

SMI

10015848-prese 21

N94- 36502

Software Engineering Technology Transfer: Understanding the Process

518-61
12700
p- 19

Marvin V. Zelkowitz
Institute for Advanced Computer Studies
and Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

Technology transfer is of crucial concern to both government and industry today. In this report, the mechanisms developed by NASA to transfer technology are explored and the actual mechanisms used to transfer software development technologies are investigated. Time, cost, and effectiveness of software engineering technology transfer is reported.

1 Introduction

The transfer of technology from the developer to the consumer of that technology is of crucial concern to U. S. industry today as the need to remain economically competitive in a global marketplace forces all organizations to constantly improve their mechanisms for doing business. Government is not immune from these forces and needs to understand and participate in such activities at all levels.

The National Aeronautics and Space Administration (NASA), as a large government agency, plays a role as both a producer and consumer of such new technologies:

As producer. As the premier space agency of the United States, NASA has a mission to develop space technologies. Transferring these technologies to private industry and aiding in the commercialization of those technologies allows for government help in promoting U.S. industry internationally.

As consumer. However, with an annual budget of over $15 billion, NASA is involved in a great many activities, and using the best techniques – whether developed internally or developed by those outside of NASA – enables NASA to wisely use its appropriated funds in order to work on complex tasks as economically as is practical.

NASA understands its role in technology transfer:

> "Technology transfer is a fundamental mission [of NASA]. It is as important as any NASA mission and it must be pursued."[1]

Accordingly, NASA has set up several organizations within NASA, or affiliated with NASA, to deal with technology transfer. NASA has a perceived model of how technology transfer should operate. However, how well do these mechanisms actually work? What is the actual process used to transfer technology? What are the characteristics of technology transfer?

While NASA's main function is to develop space technology by building and launching satellites and manned missions, as a large technological organization, NASA must increasingly rely on computer technology to play an increasingly important role in all of its operations. Therefore, technology transfer of computer technology is also a major component of its technology transfer mission.

Therefore, given the existing model of technology transfer within NASA, how well does it address software technology? More specifically, given that:

1. Industry must transfer technology from developers to users,
2. Technology transfer is an integral part of NASA's mandate,
3. Software technology is an important component of many NASA activities,
4. Mechanisms have already been established by NASA to affect that transfer, and
5. NASA has a perceived model of this transfer.

we wish to learn:

[1] Daniel S. Goldin, NASA Administrator, December, 1992.

1. What is the real model used to affect technology transfer?

2. How well does this model work with software development technologies?

3. What software development technologies have actually been transferred successfully? and

4. What characteristics can we learn about technologies that have been transferred?

This report is organized as follows: In Section 2 some existing notions about technology transfer are presented and in Section 3 the current NASA model on technology transfer is given. Section 4 describes one general survey of industry that provides a rough baseline of technologies that have been transferred within the last 15 years, while in Section 5 NASA's role in both importing and exporting software engineering technologies are discussed. Conclusions from this work are given in Section 6.

2 Background

2.1 Process Improvement

Of great concern to all industry is the need to improve productivity. Within the computer science community, the ability to improve the process of developing software has been found to be a major impetus towards improving productivity and reliability of the resulting systems systems. Concepts like the Software Engineering Institute's Capability Maturity Model [2] have grown in importance as a means for modifying the software development process. The Experience Factory concept of the NASA/GSFC Software Engineering Laboratory (SEL) [1] has shown the value of process improvement.

However, all process improvement involves changes. Some of these may be relatively minor alterations to the current way of doing business (e.g. replacing one compiler or editor by another). However, some may require major changes that affect the entire development process (e.g., using cleanroom software development).

In order for an organization to continually improve its process, it must be aware of how it operates and what other technologies are available that may be of use. Understanding this process of technology transfer should enable NASA to better use its existing resources and to better plan for the future.

2.2 Technology transfer

When we discuss *technology transfer*we will mean the insertion of one technology into a new organization that previously did not use that technology. The insertion must be such that the new organization regularly uses that technology if the appropriate conditions on its use should arise in the future.

We will call the original creator of that technology the *producer* of the technology and the organization that accepts and uses the new technology the *consumer* of that technology. The process of moving the technology out of the producing organization will be called *exporting* the technology while the process of installing the technology in the new organization will be called *infusing* the technology.

Implied by the above definitions is the notion that a successfully transferred technology becomes part of the state-of-the-practice, or normal operating procedures, of the infusing organization. For example, an organization that experiments with Ada as a programming language and then decides to use it for all applications in a specific domain (e.g., for all flight simulators) can be said to have successfully transferred that technology. On the other hand, if a technology is tried once or twice (e.g., the ML programming language for expert system development) and is found wanting and will not be used again, then that technology will not be considered to be transferred.

Not transferring a technology does not imply that the technology is not effective; only that it does not apply to the particular consumer domain. For example, there is still a demand for buggy whips among horse enthusiasts and certain theme park operators, but they have few applications among most urban automobile repair shops.

What technology are we interested in?

"Technology" is a very imprecise concept. For this report we are mainly concerned with tools, procedures and mechanisms that aid in the development of software products. We can divide this domain into two categories:

Software development technology. This includes the tools and procedures used by the software engineering profession to build software. It includes, in addition to the usual computer-based items like machines, editors, compilers, testing tools and configuration management systems, items like electronic mail, desktop publishing, spreadsheets and any other tool or device useful for software production. This can even include the telephone or fax machine if either provides aid in the development of software.

	NASA Consumers	External Consumers
NASA Producers	Transferred within NASA	Exported from NASA
External Producers	Infused into NASA	*Not of interest*

Table 1: Participants in technology transfer

Software engineering technology. This includes those software development technology items created specifically for software development. Thus, while it will include compilers and testing tools, it will not include items like electronic mail or the fax machine which also have uses in other domains.

2.3 Technology transfer participants

In describing the transfer of technology into and out of NASA, we have four potential groups of producers and consumers to consider (Table 1). NASA may be either a producer of a consumer of some technology. Similarly, some other organization may be either the producer or consumer of that technology. Of the four potential cases, only three are considered in this report – those involving NASA as either a producer or consumer. The case where an external producer transfers a technology to an external consumer is certainly of interest, but is outside of the scope of this work on NASA's role in technology transfer.

2.4 Technology maturation

In 1985, Redwine and Riddle [3] published the first comprehensive study of software engineering technology maturation. Their goal was to understand the nature of technology maturation – what was the length of time required for a new concept to move from being a laboratory curiosity to general acceptance by industry. They defined maturation of a technology as a 70% usage level across the industry.

Technology maturation involves 5 stages – two by the producer of the technology and three by consumers of that technology (See Figure 1):

1. The original *concept* for the technology appears as a published paper or initial prototype implementation. The initial time period is the development of the concept by the originator of the technology.
2. The *implementation* of the technology involves the further development of the concept by the originator or successor organization until a stable useful version is created.
3. In the initial *experimental* (or understanding) stage, other organizations experiment, tailor, expand, modify and try to use the technology.
4. In the later *exploration* (or transition) stage, use of the technology is further modified and expands penetration across the industry.
5. The final *maturation* stage is reached when 70% of the industry uses the technology.

In their study, they looked at 17 software development technologies that were developed from the 1960s through the early 1980s (e.g., UNIX, spreadsheets, object oriented design, etc.). Their results, most related to this current project are:

- They were unable to clearly define "maturation" for most technologies, but were able to make reasonable estimates as to the length of time needed for new technologies to be widely available.
- Technologies required an average of 17 years to pass from an initial concept to a mature product.
- Technologies, once developed, required an average of 7.5 years to become widely available.

In this current study, we are not interested in the general issue of technology maturation, but instead the infusion (or exporting) of a technology into or out of a single organization (NASA). Therefore, we would expect this 7.5 year average exploration stage to be an upper bound. What would be a reasonable value for NASA-infused or developed software technology?

3 NASA Model of Technology Transfer

Since technology transfer is part of NASA's mandate, a model of technology transfer has grown within the agency. Several offices and related organizations have been created for dealing with technology transfer. These include the following organizations:

- Technology transfer organizations at each NASA center:

Technology Utilization Office. The Technology Utilization Office (TUO – or Technology Transfer Office (TTO) as part of Code 700 (Engineering) at GSFC) is the major proponent of technology

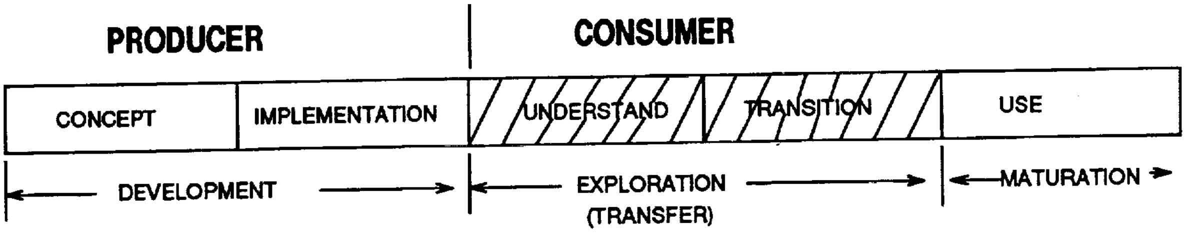


Figure 1: Technology Maturation Life Cycle

transfer between the engineer and industry. Its focus is to aid the NASA engineer in moving an idea into industry.

Office of Commercial Programs. The Office of Commercial Programs (OCP) is the major interface between each NASA center and industry as an intermediary in the commercialization of concepts that arose in NASA research.

- National and regional technology transfer organizations:

National Technology Transfer Network. The National Technology Transfer Network and the various regional technology transfer field centers act as intermediaries between the individual TUOs at each center and industry. Six Regional Technology Transfer Centers (RTTC) work directly with industry to aid in the commercialization of NASA products.

Technology Application Team. The Technology Application Team (TAT) is located at Research Triangle Institute and works with each TUO in developing technology transfer projects.

COSMIC. The Software Technology Transfer Center is a repository of software developed by NASA personnel. It is located at the University of Georgia. Over 5,000 programs have been submitted to COSMIC for distribution since 1966.

- Technology transfer publications and agreements:

Space Act Agreements. Space Act Agreements (SAA) are like memoranda of Understandings (MOUs) or CRADAs (Cooperative Research and Development Agreements in other federal agencies) for joint industry–NASA cooperation on specific projects.

NASA Tech Briefs. "NASA Tech Briefs" is a monthly publication for announcing new inventions and innovations.

	NASA Consumers	External Consumers
NASA Producers	COSMIC Tech Briefs Conferences Papers	COSMIC Tech Briefs Conferences Papers TAT NTTN TUO OCP SAA Spinoff
External Producers	Conferences Papers	*Not of Interest*

Table 2: NASA Tech Transfer Model

Spinoff. "Spinoff" is an annual publication that summarizes those technologies that have been successfully commercialized during the previous year.

Conferences and publications. Conferences and publications (both NASA sponsored and non-NASA sponsored) are a major source of information on technology that has been produced both within and outside of NASA.

Merging this list of technology transfer mechanisms with our previous model of technology transfer participants (Table 1), we get a clearer picture of how NASA addresses technology transfer (Table 2).

Two results become immediately apparent from this table:

1. There is no infusion mechanism for bringing new technology into the agency.

2. The major goal is the transfer of products.

The former result, under today's economic climate, needs to be reevaluated. Previously, there was a desire on the part of Congress and those in charge of NASA to show that such large sums of money spent on space applications had practical benefits for U.S. industry. Therefore, technology transfer to industry gave concrete indications of the value of space exploration.

However, the situation today is an era of static or shrinking budgets. The concept of "Faster, Better, Cheaper" is heard more and more. NASA needs to "Work Smarter." One way to do that is to use better technology and infuse better techniques into the agency. However, the current model assumes that engineers can simply learn about such technologies from reading papers and going to conferences. There is no explicit aid to help in this search for better ways to do NASA's job.

The second result, transfer of products, also needs to be reevaluated in the light of software engineering technologies. In most engineering disciplines, processes are centered in various products that implement that technology. Thus transferring a technology is generally equivalent to transferring a product.

The same cannot be said of many software engineering processes. For example, within the GSFC Software Engineering Laboratory (SEL), the following list of processes have been studied over the past few years:

- Object Oriented Technology,
- Goals/Questions/Metrics paradigm of software development,
- The Experience Factory model of development, and
- Cleanroom software development.

None of these processes is embodied in a product. One cannot buy a "Cleanroom" program. Instead one buys some books, a training course and some guidance on using the technique. Although NASA does not explicitly address the packaging of such processes as assets to be transferred, NASA in not unique in this regard. It is not clear that much of industry understands the unique role that processes play in software development compared to most engineering processes. It is imperative for NASA to understand this distinction and to address the transfer of processes as well as products.

Total Replies (44)		NASA Replies (12)	
Item	No.	Item	No.
Workstations & PCs	27	*Object oriented	12
*Object oriented	21	Networks	10
GUIs	17	Workstations & PCs	8
*Process models	16	*Process models	7
Networks	16	*Measurement	5
*C and C++	8	GUIs	4
*CASE tools	8	*Structured design	3
Database systems	8	Database systems	2
Desktop publishing	8	Desktop publishing	2
*Inspections	7	*Dev. methods	2
Electronic mail	7	*Reuse	2
		*Cost estimation	2
		Comm. Software	2

* – Software engineering technologies

Table 3: Top 10 transferred technologies

4 Software Development Technology

In order to understand software engineering technology transfer within NASA, it was first necessary to understand if there was any consensus about how software development technology has evolved over the past decade. Papers are often written about the so-called "software crisis" with comments that software development has not changed at all in 30 years. If so, then obviously no technology has been transferred lately.

In order to address this, a brief survey form was prepared and widely distributed. The form asked for the top five technologies that have changed software development practices since 1980. A list of approximately 100 items was included, and the respondent could pick from that list or add any other items that seemed relevant.

A total of 44 forms were returned. Of these, 12 were from NASA/GSFC personnel or NASA contractors. The top 10 (including ties) from the total list and the NASA list are given in Table 3.

The level of consensus among the 44 returned forms was quite surprising. The top 5 in the list clearly dominated the others. Of the top 5, only two of them (objected oriented technology and process models) were clearly software oriented technology. Two were mostly hardware (workstations and networks) and the other (graphical user interfaces) was partially a software engineering issue, but does not strictly fit into our definition of software engineering technology given its use in many application programs.

Among the NASA replies, there was likewise strong

	NASA Consumers	External Consumers

NASA Producers	Reuse(Kaptur, CLIPS)
	Environments (SSE)
	GUI(TAE)
	Measurement(SME, GQM)
	AI tools
	CASE Tools
		Reuse(Kaptur, CLIPS)
		Environments (SSE)
		GUI(TAE)
		Measurement(SME, GQM)

External Producers	Cost models * Formal Inspections * Object oriented technology * Ada, C, C++ * Cleanroom Rate monotonic scheduling CASE tools	*Not of Interest*

* – technologies discussed in more detail

Table 4: Transferred technologies at NASA

agreement with the total list. The top 5 on the total list were 5 out of 6 among the NASA entries. Only measurement, a strong component of the SEL, raised the recognition level among NASA personnel of its importance.

Other technologies that have been claimed as effective that were mentioned in the survey include: Measurement (in 12th place), Ada (in 14th), Formal methods (in 17th) and UNIX (in 17th).

Clearly some software development technologies have been transferred. The question was now: What effect did NASA have on this transfer?

5 Software Engineering Technology

In order to understand technology transfer in NASA, three or four software engineering experts at each NASA center were interviewed in order to determine what software engineering techniques were being used effectively in the agency. In order to keep the scope of this problem reasonable, the following two restrictions were imposed:

1. The technology had to clearly be software engineering. For example, successfully transferred programs, such as the modelling system NASTRAN available through COSMIC, were not included.

2. The technology had to have a major impact on several groups within NASA. With more than 4,000 software professionals affiliated with GSFC alone (including government and contractors), almost every software product has probably been used somewhere in the agency. While this was somewhat subjective, a list of transferred technologies was developed (Table 4).

Those technologies used (i.e., consumed) by NASA and those technologies produced by NASA will be considered separately.

5.1 Technology Infusion

Within the SEL

Three technologies that were successfully transferred by the SEL (Ada, Object oriented technology, and Cleanroom) will be discussed in greater detail. The details of transferring those technologies are summarized by Figure 2 and are explained in greater detail in the following subsections.

SEL use of Ada

Understanding phase of Technology Transfer.

Use of Ada on SEL projects was first considered in 1985. A training and sample program was the first Ada activity. However. in order to truly evaluate the appropriateness of Ada within the SEL environment, a parallel development of an Ada and FORTRAN simulator (GRODY) was undertaken. The results was an operational, but slow, product. Although the development of this simulator continued until early 1988, by early 1987 it was decided that the initial project was sufficently successful to continue the investigation of

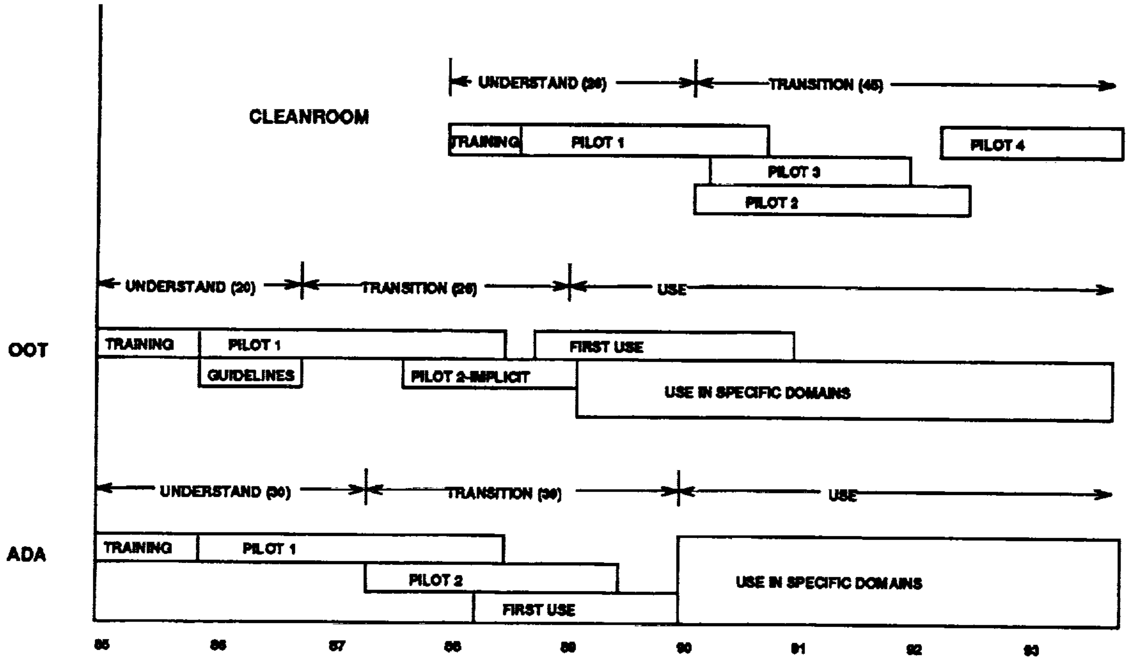


Figure 2: SEL technology transfer experience

Ada. Elapsed time since start of Ada activity was 30 months.

Transition phase of technology Transfer.

Because of the poor performance on the GRODY simulator, a second Ada project (GOADA) was undertaken where performance was of greater concern. In this case, the resulting product was comparable to performance of previous FORTAN simulators. In 1988 a third simulator was undertaken and developed successfully. By the end of 1989, Ada became the language of choice for simulators in the flight dynamics branch. Transition time was another 30 months.

Comments on Technology Transfer.

Total transfer time for Ada was approximately 60 months. Ada is the language of choice for simulator projects. Although Ada code costs more, line by line, than FORTRAN code, the higher levels of reuse result in lower overall delivery costs for such projects.

Ada was also used as the implementation language on larger mission ground support software. This was not as successful. However, the inhibitors in this case were outside of the scope of the Ada language, itself. The operational systems at GSFC are IBM mainframe compatible, and no effective Ada compiler existed for this environment during the 3 times Ada was evaluated. All of the successful simulator projects were implemented on DEC VAX computers, which had an effective Ada system.

Presently, Ada is used on approximately 15% of the SEL's software. Eleven Ada projects have been completed to date.

SEL use of object oriented technology

Understanding phase of Technology Transfer.

Use of object oriented technology (OOT) in the SEL was seriously investigated at the same time as Ada was considered. In developing the requirements for GRODY, it became apparent that the standard GSFC requirements document was oriented towards a FORTRAN functional decomposition and the use of these requirements on an Ada project would be very inefficient.

Therefore the requirements were redone to use a more object oriented approach. Following this, an OOT guidebook for GSFC was developed (GOOD - General Object Oriented Software Development [4]) for use on future projects.

Elapsed time for these activities took from early 1985 until August, 1986, or a total of 20 months. Expenses for understanding this technology were high since this activity was wrapped up in the Ada evaluation which required parallel system development.

Transition phase of technology Transfer.

On a second project (UARSAGSS), object oriented design was used implicitly. This was a FORTRAN ground support system, and experiences gained from the earlier GRODY effort allowed the programmers to better understand the design and use OOT. By the end of this project, it was sufficiently clear that OOT was an effective technique in some domains. Transition time was on the order to 26 months.

Comments on Technology Transfer.

Total transfer time in this case was only 46 months. Although almost 4 years, this was relatively short since it did not require major changes in system development. OOT could be used with Ada, FORTRAN, or any other programming language. Since it fit within the usual development paradigm, tailoring the method and inserting it into the usual NASA development process was relatively easy.

SEL use of cleanroom

Understanding phase of Technology Transfer.

In order to understand cleanroom, a series of training courses were given in 1988 by Dr. Harlan Mills, original developer of the method. A pilot project was undertaken and proved to be very successful. All participants were converts to the method, even though several had reservations about it before they began. Time to understand the method (training until the start of the second cleanroom project) was 26 months.

Transition phase of technology Transfer.

Two follow-on cleanroom projects were undertaken. A smaller in-house development was very successful, but a larger contracted project was not so successful. It was not as apparent that problems on the larger project were due to scaling up of cleanroom to larger tasks or to a lack of training and motivation of the development team on this project.

Because of this, a fourth cleanroom project was undertaken. This project is still under development, but preliminary results look very promising.

Comments on Technology Transfer.

Cleanroom technology appears to be an effective technology. Understand time was 26 months and transition time is at least 46 months, with transition still underway. Cleanroom cannot be said to be a mature technology yet, although results look very good.

Technology infusion at other NASA centers

Use of Formal Inspections

Transfer of technology.

Work began by John Kelly on formal inspections at JPL. The elapsed time for developing the method was 30 months and involved about 6 staff months of effort. Contacts with Mike Fagan of IBM, developer of the technique, aided the transition process. It has been successfully transferred to JPL and between its initial use in February, 1987 through 1990, over 200 inspections were carried out.

Based upon experience at JPL, formal inspections were moved to Langley. This took only 16 months, because of the previous experiences at JPL. About 12 staff months of effort were required, but most of this effort was in "unpaid overtime." No NASA support was available for developing the technology.

Once installed at Langley, it has been transferred to several contractors working at Langley.

Comments on Technology Transfer.

Formal inspections were successfully transferred at Langley. Total time for transferring at both centers were 30 months and 16 months. These were relatively short since formal inspections cover only a relatively narrow and precise process in software development and can be inserted relatively easily into almost any mature development process.

5.2 Exporting Technology

This present version of this report is concerned mainly with the infusion of technology from outside of the agency. A later version will address technology exporting in more detail.

6 Conclusions

From this initial study, we can make several conclusions and observations:

1. **NASA mechanisms do not address software engineering technologies well.** Technology infusion is generally ignored and left up to the individual engineer to discover on his own what is needed and available. With today's shrinking budgets and the need to work "Better, Faster, Cheaper," NASA needs to address this issue and help in the search for new effective technology to use.

   In addition, software engineering processes are not addressed. These are not product centered. How to package and transfer processes as a corporate asset needs to be handled better.

2. **Technology transfer is more than simply understanding the new technology.** Technology transfer takes time. Understanding the technology has been shown to take upwards of 2.5

years. The transition time when the organization is exploring, tailoring and modifying the technology for its own use often takes more than the understanding time, with a total transfer time on the order of five years not being unusual.

3. **Technology transfer is part of the total environment of the consumer organization.** Technology transfer does not occur in a vacuum. The SEL experience with Ada demonstrates this concept. Ada has proven to be successful with flight simulators, and the effective Ada compiler on the DEC VAX computer helped in this transfer. However, the operational systems for flight dynamic software was the IBM mainframe, and no effective Ada compiler was available during the 5 years (from 1985 to 1990) when Ada was under evaluation. Because of this, FORTRAN is still the language of choice for such applications. Had an effective mainframe Ada compiler been available, then the result of evaluating Ada for large AGSS (Attitude Ground Support Software) systems might have been different.

4. **People contact is the main transfer agent of change.** As many others have observed, technology transfer occurs best when the developers of a technology are involved in the technology transfer process. In this report, that happened in order for cleanroom to be effectively used at GSFC and for inspections to be brought to JPL and then to Langley.

7 Acknowledgment

This research supported in part by grant NSG-5123 from NASA Goddard Space Flight Center to the University of Maryland, College Park, Maryland. Frank McGarry of NASA/GSFC and Kellyann Jeletic of Computer Sciences Corporation helped in collecting the data described in this report.

References

[1]Basili V. R., The experience factory: Can it make you a 5?, 17th NASA/GSFC Software Engineering Workshop, Greenbelt, MD (December, 1992) 55-64.

[2]Paulk M. C, B. Curtis, M. B. Chrissis, and C. V. Weber, Capability Maturity Model for Software, Version 1.1, Technical Report SEI-93-TR-24, Software Engineering Institute, Pittsburgh, PA (1993).

[3]Redwine S. and W. Riddle, Software technology maturation, 8th IEEE/ACM International Conference on Software Engineering, London, UK, August, 1985, 189-200.

[4]Seidewitz E. and M. Stark, General Object Oriented Software Development, SEL-86-002, NASA/GSFC, August, 1986.

1993 Software Engineering Workshop

Software Engineering Technology Transfer: Understanding the Process

Marvin V. Zelkowitz

Institute for Advanced Computer Studies and

Department of Computer Science

University of Maryland

College Park, Maryland

Technology Transfer Overview

- "Technology transfer is a fundamental mission. It is as important as any NASA mission and it must be pursued." – Daniel S. Goldin, NASA Administrator, December, 1992
- It is critical to understand technology transfer as part of any process improvement program.
- Technology maturation takes time: From Redwine – Riddle study (1985) on software engineering technologies:
  - Studied 17 software engineering technologies of the 1960s and 1970s.
  - Required an average of 17 years from concept to maturation.
  - Required an average of 7.5 years after initial development to widespread availability in industry.
- Fundamental issues:

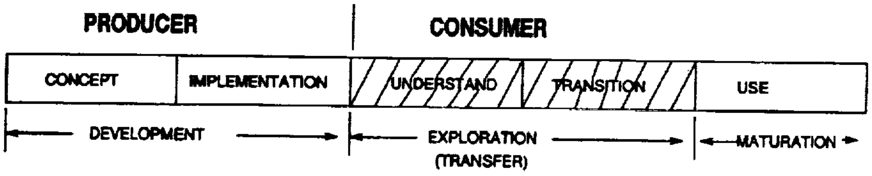
  How does NASA think technology transfer takes place?

  How does technology transfer really take place?

Technology Transfer Goals

- Issues:
  - Must transfer technology from developers to users
  - Technology transfer is an integral part of NASA's mandate
  - Mechanisms have been established by NASA for transfer
  - NASA has a perceived model of this transfer
- But:
  - What is the real model of technology transfer?
  - What processes were used to affect those transfers?
  - What software development technologies have been transferred?
  - What were the costs and effort for those transfers?

Technology Transfer Stages



PRODUCER

CONSUMER

CONCEPT

IMPLEMENTATION

UNDERSTAND

TRANSITION

USE

DEVELOPMENT

EXPLORATION (TRANSFER)

MATURATION

This initial study covers technology infusion only (e.g., Exploration stage within NASA.)

Domain of Interest

- What technologies are of interest?
  - Software development technology – Tools and procedures used by software engineering profession to build software
  - Software engineering technology – Tools and procedures developed specifically for software development
- What technologies do software engineers use?
  - Software development technologies in use – Present broad-based survey of software engineering professionals on what software development technologies have been successful since 1980.
  - Software engineering technologies transferred by NASA – Present results of interviews and surveys with selected NASA personnel and reading selected NASA documentation and reports.

Technology Transfer Participants

	NASA Consumers	External Consumers
NASA Producers	Transferred within NASA	Exported from NASA
External Producers	Infused into NASA	*Not of interest*

Purpose of this analysis is to understand both the exporting and importing of software engineering technology for NASA

Technology Transfer Parameters

- From Producers:
  - Motivation (need) for technology
  - Cost of technology
  - Time to develop technology
  - Commercialization potential of technology
  - Cost and time to transfer technology
- From Consumers:
  - Motivation (need) of technology
  - Methods to investigate technology
  - Cost required to infuse technology
  - Time required to infuse technology

Top 10 Recently Transferred Technologies

Total Replies (44)		NASA Replies (12)	
Item	No.	Item	No.
Workstations and PCs	27	*Object oriented	12
*Object oriented	21	Networks	10
GUIs	17	Workstations and PCs	8
*Process models	16	*Process models	7
Networks	16	*Measurement	5
*C and C++	8	GUIs	4
*CASE tools	8	*Structured design	3
Database systems	8	Database systems	2
Desktop publishing	8	Desktop publishing	2
*Inspections	7	*Development methods	2
Electronic mail	7	*Reuse	2
		*Cost estimation	2
		Comm. Software	2
. . .		. . .	
*Measurement (12)	6	*C (14)	1
*Ada (14)	4	*Ada (14)	1
*Formal methods (17)	3	*Inspections (14)	1
UNIX (17)	3		

* – Software engineering technologies

NASA Transfer Mechanisms

- NTTN – National Technology Transfer Network. Joint NASA and other Federal agency transfer centers. NASA field centers and regional technology transfer centers for interacting with industry.
- COSMIC – NASA Software Technology Transfer Center. At University of Georgia to make NASA software available.
- TAT – Technology Application Team. At Research Triangle Institute, works with each Technology Utilization Office at each NASA center for in developing technology transfer projects.
- Space Act Agreement – Joint NASA/industry project. (Similar to MOUs, CRADAs)
- TUO – Technology Utilization Office – Office at each center for interacting with outside agencies
- NASA Tech Briefs – Monthly publication for announcing new inventions and innovations.
- Spinoff – Annual NASA publication describing successfully transferred technologies.

NASA Transfer Model

	NASA Consumers	External Consumers
NASA Producers	COSMIC Tech Briefs Conferences Papers	COSMIC Tech Briefs Conferences Papers TAT NTTN TUO Space Act Agreements Spinoff
External Producers	Conferences Papers	*Not of Interest*

Goal is transfer of products.

No infusion mechanism.

Software Engineering Processes

- Technology Transfer is generally product oriented – In most engineering disciplines, the process is centered in the product.
- Software engineering does not yet fulfill that model – Processes describing actions to take are as important as the tools that are used.
- For example, many of the technologies explored by the GSFC Software Engineering Laboratory are procedures only and not tools:
  - Object oriented technology
  - Goals/Question/Metrics model
  - Measurement
  - Cleanroom
  - Inspections

Processes as opposed to products are dominant.

NASA Emphasis on Technology Transfer

- Summary of NASA Technology Transfer Model:
  - Agents of technology transfer are people.
  - Description of technology transfer are published papers.
  - Objects of technology transfer are products.
- But:
  - No mechanisms for transfer of processes.
  - Seems to be true throughout industry, not just NASA.
  - No mechanism for technology infusion.

What Has Been Transferred?

- Domain of interest – Software engineering technologies (e.g., Most programs in COSMIC are application programs and not of interest for this talk.)
- But NASA is big ...
  - Thousands of programmers nationwide. Probably every tool sold has been used somewhere within NASA.
  - Need to identify only those technologies that have made major impact on development practices
- Preliminary results of directed survey of software engineering professionals within NASA.

Transferred Software Development Technology

	NASA Consumers	External Consumers

NASA Producers	Reuse(Kaptur, CLIPS)
	Environments (SSE)
	GUI(TAE)
	Measurement(SME, GQM)
	AI tools
	CASE Tools
		Reuse(Kaptur, CLIPS)
		Environments (SSE)
		GUI(TAE)
		Measurement(SME, GQM)

External Producers	Cost models * Formal Inspections * Object oriented technology * Ada, C, C++ * Cleanroom Rate monotonic scheduling CASE tools	*Not of Interest*

* – Technologies to be discussed

Representative list based upon survey and interviews

Case study: SEL use of Ada

Understand time	30 mo
Transition time	30 mo
Cost	High, Parallel development
Replaces	FORTRAN
Infusion method	Courses, 2 pilot projects
Status	Mature use in specific domain
Tech Transfer	Used on some projects
Success	Generally positive

Comments

- Increased costs for new projects
- 10%–25% savings on later projects due to 25% – 30% reuse

Case study: SEL use of OOT

Understand time	20 mo
Transition time	26 mo
Cost	High (Part of Ada evaluation)
Replaces	Functional decomposition
Infusion method	Courses, Training guide, 2 pilot projects
Tech Transfer	Used on most projects
Status	Mature use in specific domains
Success	Very positive

Comments

- Initial results – Decreased time and effort and improved error rates
- Needs training – Replaces design method that already worked well and generates few errors

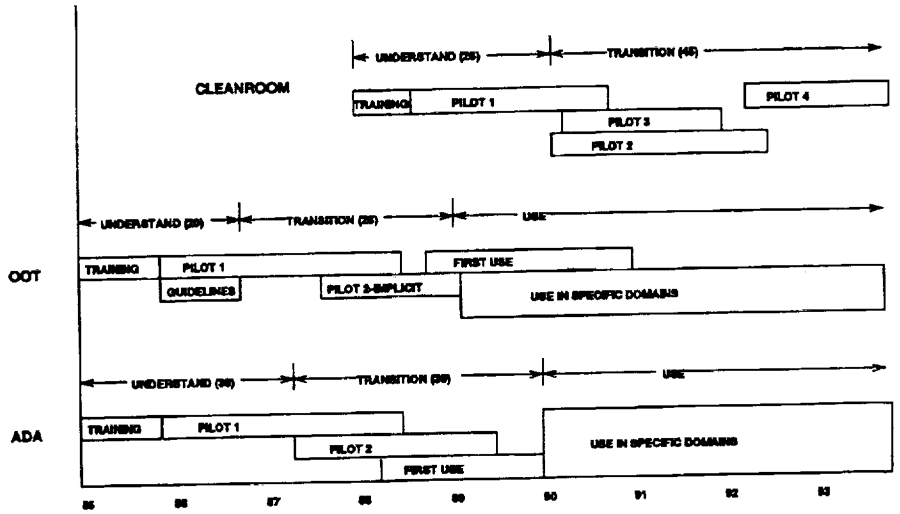
Case study: SEL use of Cleanroom

Understand time	26 mo
Transition time	45+ mo
Cost	High
Replaces	Traditional testing
Infusion method	External developers, training, pilot studies
Status	Still in transition
Tech Transfer	Unclear
Success	Appears very positive

Comments

- Contact with developer important for early success
- Large project not as successful – less training and motivation
- Productivity and error rates improved on all projects
- Still evaluating, training and undergoing transition

Summary of SEL Experiences



Case study: Formal Inspections

	Site 1	Site 2
Transfer time	30 mo	16 mo
Cost	.5 FTE	1 FTE, unpaid overtime
Replaces	Walkthroughs	New activity
Infusion method	External developer	External developer, site 1
Status	In use	In use
Tech Transfer	Used within NASA, site 2	NASA government contractors
Success	High	High

Comments

- Technology transfer not well supported
- People contact main agent of change

Conclusions

NASA mechanisms do not address software engineering technologies well.

- Technology infusion is generally not addressed.
- Process technology is similarly not addressed.

Technology transfer is more than simply understanding the new technology.

- Time to understand technology is generally on the order of 2.5 years.
- Transition time at least as long as understanding time.

People contact seems to be the main transfer agent of change.

Disclaimer: Presentation based upon preliminary analysis of available information. Will be refined over next few months.

omit to END

Appendix A: Workshop Attendees

- Abd-El-Hafiz, Salwa K., University of Maryland
- Agresti, Bill W., MITRE Corp.
- Allen, Russell G., IRS
- Anderson, Barbara, Jet Propulsion Lab
- Angier, Bruce, Institute for Defense Analyses
- Astill, Pat, GSC/SAIC
- Aucoin, Ed, GenRad, Inc.
- Ayers, Everett, Ayers Associates

- Bachman, Scott, DoD
- Bacon, Beverly, Computer Sciences Corp.
- Bailey, John, Software Metrics, Inc.
- Baker, David, Computer Sciences Corp.
- Barnard, Julie, IBM Federal Systems Company
- Barski, Renata M., AlliedSignal Technical Services Corp.
- Basili, Vic, University of Maryland
- Bassman, Mitchell J., Computer Sciences Corp.
- Beifeld, David, UniSys Corp.
- Belcher, Melody S., SYSCON
- Bickford, Paul, USDA FNS
- Bisignani, Margaret, MITRE Corp.
- Bissonette, Michele, Computer Sciences Corp.
- Blackwelder, Jim, Naval Surface Warfare Center
- Blagmon, Lowell E., Naval Center For Cost Analysis
- Blum, Bruce I., Applied Physics Lab
- Blumberg, Maurice H., IBM/FSC
- Bohner, Shawn, MITRE Corp.
- Boland, Dillard, Computer Sciences Corp.
- Bond, Jack, DoD
- Booth, Eric, Computer Sciences Corp.
- Bowen, Gregory M., Computer Sciences Corp.
- Bowers, Allan, Loral AeroSys
- Bowser, Jeff, Hughes/STX
- Boycan, Steven T., HQ AFC4A
- Brackett, Edwin H., Computer Sciences Corp.
- Bradley, Stephen, MMS Systems
- Bredeson, Mimi, Space Telescope Science Institute
- Brejcha, Albert G., Jet Propulsion Lab
- Briand, Lionel, University of Maryland
- Brill, Gary B., IRS
- Britt, Joan J., MITRE Corp.
- Brown, Kenneth L., Comptek Federal Systems, Inc.
- Bullock, Steve, IBM
- Burd, Reg, Computer Data Systems, Inc.
- Burks, Catherine, FMS/QAD
- Busby, Mary, IBM/FSC
- Button, Janice, DoD

- Calavaro, Giuseppe F., University of Maryland
- Caldiera, Gianluigi, University of Maryland
- Cameron, Charlie, Computer Data Systems, Inc.
- Card, Dave, Computer Sciences Corp.
- Carlisle, Candace, NASA/GSFC
- Carlson, Randall, NSWCDDCarson, Eric T, Logicon, Inc.
- Chen, Lily Y., AlliedSignal Technical Services Corp.
- Cheng, Betty, Michigan State University
- Chiverella, Ron, PA Blue Shield
- Christenson, Irene, FMS/QAD
- Chu, Richard, Loral AeroSys
- Cicslak, Don, Hughes Aircraft Co.
- Clark, James D., Naval Surface Warfare Center
- Clark, John, COMPTEK Federal Systems
- Cochrane, Jr., J. T., Planning Systems, Inc.
- Coger, George, UniSys Corp.
- Cohen, Joel, GTE Government Systems
- Condon, Steven E., Computer Sciences Corp.
- Conley, Ronald B., NCCOSC RDTE DIV DET
- Cook, James E., Price Waterhouse
- Cook, John F., NASA/GSFC
- Còmett, Lisa K., U.S. Air Force
- Cover, Donna, Computer Sciences Corp.
- Cowan, Marcia, Loral AeroSys
- Cuesta, Ernesto, Computer Sciences Corp.

- D'Agostino, Jeff, The Hammers Co.
- D'Elia, Barbara, IRS
- Daku, Walter, UniSys
- Day, Nancy A., Naval Surface Warfare Center
- Decker, William, Computer Sciences Corp.

Denney, Valerie P., Martin Marietta
Deutsch, Michael S., Hughes Applied Information Systems, Inc.
DiIorio, Bob, IBM
DiLeo, Mike, Innovative Technology & Engineering
DiNunno, Donn, Computer Sciences Corp.
Dikel, David, Applied Expertise, Inc.
Diskin, David H., Defense Information Systems Agency
Do, Thang T., IBM
Doland, Jerry T., Computer Sciences Corp.
Donnelly, Laurie M., AlliedSignal Technical Services Corp.
Dorfman, Audrey, Vitro Corp.
Dortenzo, Donald V., Fairchild Space & Defense
Douglas, Frank J., Softran, Inc.
Drake, Anthony M., Computer Sciences Corp.
Dumas, Michel R., Fairchild Space & Defense
Dunn, Joseph, Computer Sciences Corp.
Dunn, Nepolia, Computer Sciences Corp.
Duva, Larry, IBM
Duvall, Lorraine, Syracuse University
Dyer, Michael, IBM Federal Systems Company

El-Hajj, Terry, Comptek Federal Systems, Inc.
Ellis, Walter J., Software Process & Metrics
Emery, Richard D., Vitro Corp.
Engelmeyer, Bill, North Arundle Hospital
Evans, Lawrence, IBM/FSC

Fakhre-Zakeri, Issa, .University of Maryland
Farr, William H., Naval Surface Warefare Center
Farrell, William T., DSD Laboratories, Inc.
Feerrar, Wallace, MITRE Corp.
Felber, Henry D., Software Productivity Consortium
Ferguson, Jay, DoD
Fernandes, Vernon, Computer Sciences Corp.
Fike, Sherri, Ball Aerospace
Finley, Doug, UniSys Corp.
Firer, Jacob, Hughes/STX
Flens, Leonard, Air Force
Flowers, Ed, U.S. Air Force
Forsythe, Ron, NASA/Wallops Flight Facility
Fuentes, Wilfredo, Logicon, Inc.
Futcher, Joseph M., Naval Surface Warfare Center

Gaeta, Richard A., GDE Systems, Inc.
Gaffney, John E., Software Productivity Consortium
Gallagher, Barbara, DoD
George, Leesa M., SYSCON
Glazer, Joel, Westinghouse
Godfrey, Sally, NASA/GSFC
Goel, Amrit L., Syracuse University
Goldberg, Nancy, Computer Sciences Corp.
Golden, John R., RLS, Inc.
Goldstein, Aaron G., TRW
Goodman, Nancy, NASA/GSFC
Gosnell, Arthur B., U.S. Army Missile Command
Gotterbarn, Donald, East Tennessee State University
Gover, Gary, Dept. of Veterans Affairs
Green, Scott, NASA/GSFC
Greenblatt, Alan, USDA
Griesel, Ann, Jet Propulsion Lab
Guido, Tony, Naval Air Systems Command

Hall, Dana L., SAIC
Hall, Susan M., SofTech, Inc.
Halterman, Karen, NASA/GSFC
Handrick, Bill, Systems Research & Applications Corp.
Haneef, N., IBM
Hannan, Thomas L., BTG, Inc.
Harris, Barbara A., USDA
Hauppa, Gary A., DoD
Heasty, Richard, Computer Sciences Corp.
Heller, Gerard H., Computer Sciences Corp.
Hendrick, Robert B., Computer Sciences Corp.
Hendrzak, Gary, Booz, Allen & Hamilton, Inc.
Henry, Joel, East Tennessee State Univ.
Hermosilla, Manuel N., DISA/JIEO/CIM
Higgins, Herman A., DoD
Hihn, Jairus M., Jet Propulsion Lab/Caltech
Hill, Ken, UniSys Corp.
Hill, Paul E., Computer Sciences Corp.
Hoffman, Gail, Dept. of Veterans Affairs
Hogan, Edward L., UniSys Corp.
Hollingsworth, Susan W., Lockheed Aeronautical Systems Co.
Holmes, Barbara, CRM
Hopkins, Ronald E., Dept. of the Air Force
Hormby, Tom W., Johns Hopkins University
Houston, Rod, IIT Research Institute
Howard, Scarlette, Computer Sciences Corp.
Howard, William H., UniSys Corp.
Howland, John C., GenRad, Inc.

Huffman, Dorothy, Jet Propulsion Lab
Huguley, Alan, Defense Mapping Agency
Hummer, Carroll, Tidewater Consultants, Inc.
Huza, Marilyn, IRS

Ippolito, Laura, NIST

Jackson, Lyn, Logicon, Inc.
Jai, Atal, Hughes STX
James, Jason S., DoD
Jay, Elizabeth M., NASA/GSFC
Jeffery, Ross, The University of New South Wales
Jeletic, Jim, NASA/GSFC
Jeletic, Kellyann, NASA/GSFC
Johannes, James D., University of Alabama at Huntsville
Johnson, Arnold, National Bureau of Standards
Johnson, Temp, Hughes-STX
Jones, Christopher C., IIT Research Institute
Jones, Howard L., USDA
Jones, John L., MITRE Corp.
Jones, Nick, Andrulis Research Corp.
Jordano, Tony J., SAIC

Kalin, Jay, Loral AeroSys
Kapoor, Naveen, Innovative Technology & Engineering
Kassebaum, Rob, MCI
Kelley, Rich, Computer Sciences Corp.
Kelly, John, Jet Propulsion Lab
Kelly, John C., Jet Propulsion Lab
Kemp, Kathryn M., NASA/HQ
Keshavarz-Nia, Hamid, Innovative Technology & Engineering
Keshavarz_Nia, Navid, Innovative Technology & Engineering
Kester, Rush, Computer Sciences Corp.
Kierk, Isabella K., Jet Propulsion Lab
Kleis, Karen, Computer Sciences Corp.
Knoell, Roger, U.S. Air Force
Koeser, Ken, Vitro Corp.
Kohl, Ron, IBM Federal Systems Co.
Kotov, Alexei, Oregon Graduate Institute
Kuo, Jerry H., Compex Corp.
Kurihara, Tom, Logicon, Inc.

LaPorte, Claude Y., Oerlikon Aerospace
LaVallee, David, Loral AeroSys
Landis, Linda, Computer Sciences Corp.
Langston, James H., Computer Sciences Corp.
Larman, Brian, SEI
Laubenthal, Nancy, NASA/GSFC
Leaman, Mark, Systems Research & Applications Corp.
Leavitt, Karen, Computer Sciences Corp.
Lehman, Meir, Imperial College of Science
Lemmon, Doug, University of Maryland
Levine, Leonard F., Defense Information Systems Agency
Levitt, David S., Computer Sciences Corp.
Li, Ned C., IBM
Li, Ningda Rorry, University of Maryland
Liebrecht, Paula, Computer Sciences Corp.
Lien, Chin-Ho, Computer Sciences Corp.
Lin, Chi Y., Jet Propulsion Lab
Lindsay, Scott, GSI
Lindsey, Brad, IITRI/ECAC
Lippens, Gary A., U.S. Air Force
Little, Linda C., Litton Data Systems
Liu, Jean C., Computer Sciences Corp.
Liu, Kuen-san, Computer Sciences Corp.
Loesh, Bob E., Software Engineering Sciences, Inc.
Loy, Patrick H., Loy Consulting, Inc.
Lucas, Janice P., FMS
Luczak, Ray, Computer Sciences Corp.
Luk, William S., Rockwell International

MacDonald, Kathy, IITRI/ECAC
Maccannon, Cecil, FAA
Machtey, Barbara, Computer Sciences Corp.
Mackness, Abby M., Booz, Allen & Hamilton, Inc.
Madachy, Raymond J., Litton Data Systems
Madden, Joseph A., U.S. Air Force
Majane, John A., HSTX
Marciniak, John, CTA, Inc.
Marcoux, Darwin, DoD
Markel, Susan, TRW
Mattingly, Joe, HQ AFC4A/XPSP
Maury, Jesse, Omitron, Inc.
Mazzola, Ray, Loral AeroSys
McConnel, Pat, IRS
McCreary, Julia M., IRS
McGarry, Frank E., NASA/GSFC
McGarry, Mary Ann, IIT Research Institute
McGuire, Eileen, SofTech, Inc.
McLean, David R., AlliedSignal Technical Services Corp.
McNeill, Justin F., Jet Propulsion Lab
McSharry, Maureen, Computer Sciences Corp.

Meade Gallier, Julia, Naval Surface Warfare Center
Mechels, Chris, Los Alamos National Lab
Mendonca, Manoel G., University of Maryland
Mersky, Jerry, Logicon, Inc.

Miller, Ronald W., NASA/GSFC
Moleski, Laura, CRM
Moleski, Walt, NASA/GSFC
Molloy, Michael, Martin Marietta
Monteleone, Richard A., Loral AeroSys
Montgomery, Jane, New Technology, Inc.
Moore, Paula, NOAA/SPOx3
Morgan, Elizabeth, AlliedSignal Technical Services Corp.
Morusiewicz, Linda M., Computer Sciences Corp.
Moulds, Thomas, J.F. Taylor, Inc.
Mrenak, Gary, Top Down Software, Inc.
Murthy, Mohan V., AlliedSignal Technical Services Corp.
Myers, Philip I., Computer Sciences Corp.

Narrow, Bernie, AlliedSignal Technical Services Corp.
Nassau, Dave, Applied Expertise, Inc.
Nellen, Eric, Tandem Computers
Nicholson, Dan, IITRI/ECAC
Noone, Estelle, Computer Sciences Corp.
Norbedo, Robert, Textron Defense Systems
Norcio, Tony F., University of Maryland Baltimore County

O'Brien, Robert L., Paramax Aerospace Systems
O'Donnell, Charlie, ECA, Inc.

O'Hara, Audrey J., IMDT/FAA
O'Neill, Don, Software Engineering Consultant
Oifer, Marilyn, Jet Propulsion Lab.
Ollove, Elizabeth, DoD
Olsen, Doug, Hughes STX

Padgett, Kathy G., Dept. of Commerce Census Bureau
Page, Gerald, Computer Sciences Corp.
Pajerski, Rose, NASA/GSFC
Palmer, Regina, Martin Marietta
Panlilio-Yap, Nikki M., IBM
Parker, Fran, Dept. of Vererans Affairs
Parks, Mark, Computer Sciences Corp.
Patton, Kay, Computer Sciences Corp.
Pavnica, Paul, Treasury/Fincen
Payne, Jeffery E., Reliable Software Technologies Corp.
Pecore, Joseph N., Vitro Corp.
Peeples, Ron L., IBM
Pendley, Rex, Computer Sciences Corp.
Perez, Frank, UniSys Corp.
Perlberg, Cindy, Alta Systems
Peterman, David, Texas Instruments
Pettengill, Nathan, Martin Marietta
Plett, Michael E., Computer Sciences Corp.
Ponder, Melynda, Boeing
Porter, Adam A., University of Maryland
Potter, Marshall R., Dept. of the Navy
Powell, Jeffrey, CBSI
Provenza, Clint, Booz, Allen & Hamilton, Inc.
Putney, Barbara, NASA/GSFC

Quann, Eileen S., Fastrak Training, Inc.
Quinn, David, NSA

Rasmussen, Earl, U.S. Army
Rathbun, Bob, DISA
Ray, Julie, New Technology, Inc.
Regardie, Myrna L., Computer Sciences Corp.
Reifer, Donald J., DISA/CIM/TXE
Reis, Richard, Computer Sciences Corp.
Reitzel, Morris, DoD
Rhodes, Tom, NIST
Richard, Dan, Federal Systems Company
Rinearson, Linda, GTE - Government Systems
Risser, Gary E., Dept. of Veterans Affairs
Ritter, George, New Technology, Inc.
Roberts, Becky L., CBIS Federal, Inc.
Robertson, Laurie, Computer Sciences Corp.
Robinson, Betty J., USDA
Rockwell, Frank, Intermetrics, Inc.
Rohr, John A., Jet Propulsion Lab
Rohrer, Amos M., SYSCON
Rosenberg, Linda H., UniSys Corp.
Rosenfeld, Scott R., MITRE Corp.
Roulette, Gary, Fairchild Space & Defense
Russ, Kevin M., USDA
Russell, Wayne M., GTE
Russo Waters, Olga, Logicon, Inc.
Ryder, Regina, USDA
Rymer, John, IBM/FSC

Saisi, Robert O., DSD Laboratories, Inc.
Salmon, Seth, Systems Research & Applications Corp.

Samii-Mooney, Sousan, DISA/JEIO/TFC
Samson, Don, IIT Research Institute
Santiago, Richard, Jet Propulsion Lab
Santo, Dave, TRW
Satyen, Uma D., MITRE Corp.
Sauble, George, Omitron, Inc.
Schappelle, Sam, IBM
Schilling, Mark, Project Engineering, Inc.
Schott, Gary, Hughes Missile Systems Co.
Schulmeyer, Gordon, Westinghouse
Schwartz, Michael, IITRI/ECAC
Schwarz, Henry, NASA/KSC
Schweiss, Robert, NASA/GSFC
Scott, Donna, Planning Research Corp.
Scott, Rhonda M., IBM
Seaman, Carolyn B., University of Maryland
Seaton, Bonita, NASA/GSFC
Seaver, David P., Project Engineering, Inc.
Seidewitz, Ed, NASA/GSFC
Senger, Paul J., Vitro Corp.
Shaffer, Edward S., AlliedSignal Technical Services Corp.
Shavell, Zalman A., Vitro Corp.
Sheckler, John D., AlliedSignal Technical Services Corp.
Shim, Errol D., IFPUG
Shockey, Donna, IRS
Shugerman, Marvin, Systems Research & Applications Corp.
Siddalingaiah, Vimala, Computer Sciences Corp.
Siegel, Karla, MITRE Corp.
Silberberg, David, DoD
Simmons, Barbara, DoD
Sinclair, Craig, SAIC
Singer, Carl A., BELLCORE
Singleton, Frank L., Jet Propulsion Lab
Slade, Lucius, USDA
Slonim, Jacob, IBM Canada Ltd.
Slud, Eric, University of Maryland
Smidts, Carol, University of Maryland
Smith, Arnold W., Martin Marietta
Smith, Donald, NASA/GSFC
Smith, Donna, NAWC-AD
Smoller, Gary, Hughes Information Technology Co.
Smolyak, Mikhail, Computer Sciences Corp.
Sohmer, Robert, RAM Engineering Associates
Solomon, Carl A., Hughes STX
Somerstein, Evan, IBM/FSC
Sova, Don, NASA/HQ
Spangler, Alan, IBM
Spence, Bailey, Computer Sciences Corp,
Splain, John M., Computer Sciences Corp.
Spool, Peter R., Siemens Corporate Research, Inc.
Sporn, Patricia A., NASA/HQ
Squires, Burton E., Orion Scientific, Inc.
Srivastava, Alok., Computer Sciences Corp.
Stark, Michael, NASA/GSFC
Starr, Ted, Booz, Allen & Hamilton
Stevens, K. Todd, Virginia Tech.
Sudman, William R., NAWCAD - Pax River
Sukri, Judin, Computer Sciences Corp.
Suryani Jamin Tung, Angela, AlliedSignal Technical Services Corp.
Swain, Barbara, University of Maryland
Szulewski, Paul A., C.S. Draper Labs, Inc.
Taneja, Rajiv, Tandem Computers Inc.
Tasaki, Keiji, NASA/GSFC
Theeke, Patrick A., ASSET
Thomas, George L., USDA
Thomas, William, MITRE Corp.
Thompson, John T., Loral AeroSys
Thornton, Tammye, Naval Surface Warfare Center
Thurston, Robert, NASA Center for Aerospace Information
Tiplitz, Lillian, Martin Marietta
Truong, Son, NASA/GSFC
Turek, Margaret, BTG, Inc.
Turney, Brad, Systems Research & Applications Corp.

Ulery, B. T., Mitre Corp.

Valett, Jon, NASA/GSFC
Van Verth, Patricia B., Canisius College
Vaughan, Frank R., RAM Engineering Associates
Venkaresh, Via, .CTA, Inc.
Viehman, Ralph, NASA/GSFC
Voas, Jeffrey, Reliable Software Technologies Corp.
Von Mayrhauser, Anneliese, Colorado State University

Waligora, Sharon R., Computer Sciences Corp.
Walker, Derek, IBM
Walker, John, Fairchild Space & Defense
Wallace, Dolores, NIST
Walsh, Bob, IRS
Walsh, Chuck, NASA Center for Aerospace Information
Ward, Christopher B., .DoD
Waters, Olga, Logicon, Inc.
Weber, Paul A., Technology Planning, Inc.

Weiss, Sandy L., IIT Research Institute
Weissler, Harold, Newport News Shipbuilding
Wenneson, Gregory J., Sterling Software, Inc.
Wentz, Brian, IBM
Weszka, Joan, IBM Federal Systems Company
Wetzel, Paul E., Vitro Corp.
Wheeler, David A., Institute for Defense Analyses
Wheeler, J. L., Computer Sciences Corp.
White, Cora P., New Technology, Inc.
Whitfield, Josette, IIT Research Institute
Wilson, Jim, Applied Expertise, Inc.
Wilson, Randy D., Naval Center For Cost Analysis
Wilson, Robert K., Jet Propulsion Lab
Winston, Audrey B., Hughes Applied Information Systems
Withers, Tim, Tidewater Consultants, Inc.
Wohlwend, Harvey, Schlumberger Lab. for Computer Science
Wolf, Bryan, McDonnell Douglas Aerospace
Wood, Dick J., Computer Sciences Corp.
Wood, John W., Computer Sciences Corp.
Wood, Terri, NASA/GSFC
Woodard, Fred, TRW
Woodyard, Charles E., NASA/GSFC
Wu, Sabina L., IITRI/ECAC
Wyskida, Dick M., University of Alabama at Huntsville

Youman, Charles, SETA Corp.
Young, Andy, Young Engineering Services, Inc.
Young, James M., Fairchild Space & Defense
Youngblood, Jim, Lockheed (LESC)

Zaveler, Saul, U.S. Air Force

Zelkowitz, Marv, University of Maryland
Zimet, Beth, Computer Sciences Corp.
Zucconi, Lin, Lawrence Livermore National Laboratory

Appendix B: Standard Bibliography of SEL Literature

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, *Proceedings From the First Summer Software Engineering Workshop*, August 1976

SEL-77-002, *Proceedings From the Second Summer Software Engineering Workshop*, September 1977

SEL-78-005, *Proceedings From the Third Summer Software Engineering Workshop*, September 1978

SEL-78-006, *GSFC Software Engineering Research Requirements Analysis Study*, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, *Applicability of the Rayleigh Curve to the SEL Environment*, T. E. Mapp, December 1978

SEL-78-302, *FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3)*, W. J. Decker, W. A. Taylor, et al., July 1986

SEL-79-002, *The Software Engineering Laboratory: Relationship Equations*, K. Freburger and V. R. Basili, May 1979

SEL-79-004, *Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment*, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, *Proceedings From the Fourth Summer Software Engineering Workshop*, November 1979

SEL-80-002, *Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation*, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-005, *A Study of the Musa Reliability Model*, A. M. Miller, November 1980

SEL-80-006, *Proceedings From the Fifth Annual Software Engineering Workshop*, November 1980

SEL-80-007, *An Appraisal of Selected Cost/Resource Estimation Models for Software Systems*, J. F. Cook and F. E. McGarry, December 1980

SEL-80-008, *Tutorial on Models and Metrics for Software Management and Engineering*, V. R. Basili, 1980

SEL-81-011, *Evaluating Software Development by Analysis of Change Data*, D. M. Weiss, November 1981

SEL-81-012, *The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems*, G. O. Picasso, December 1981

SEL-81-013, *Proceedings of the Sixth Annual Software Engineering Workshop*, December 1981

SEL-81-014, *Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL)*, A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, *Guide to Data Collection*, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-104, *The Software Engineering Laboratory*, D. N. Card, F. E. McGarry, G. Page, et al., 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-81-305, *Recommended Approach to Software Development*, L. Landis, S. Waligora, F. E. McGarry, et al., June 1992

SEL-81-305SP1, *Ada Developers' Supplement to the Recommended Approach*, R. Kester and L. Landis, November 1993

SEL-82-001, *Evaluation of Management Measures of Software Development*, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, *Collected Software Engineering Papers: Volume 1*, July 1982

SEL-82-007, *Proceedings of the Seventh Annual Software Engineering Workshop*, December 1982

SEL-82-008, *Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory*, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, *FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1)*, W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, *Glossary of Software Engineering Laboratory Terms*, T. A. Babst, M. G. Rohleder, and F. E. McGarry, October 1983

SEL-82-1106, *Annotated Bibliography of Software Engineering Laboratory Literature*, L. Morusiewicz and J. Valett, November 1992

SEL-82-1206, *Annotated Bibliography of Software Engineering Laboratory Literature*, L. Morusiewicz and J. Valett, November 1993

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 of the Eighth Annual Software Engineering Workshop*, November 1983

SEL-83-106, *Monitoring Software Development Through Dynamic Variables (Revision 1)*, C. W. Doerflinger, November 1989

SEL-84-003, *Investigation of Specification Measures for the Software Engineering Laboratory (SEL)*, W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, *Proceedings of the Ninth Annual Software Engineering Workshop*, November 1984

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

SEL-85-001, *A Comparison of Software Verification Techniques*, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, *Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team*, R. Murphy and M. Stark, October 1985

SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985

SEL-85-004, *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics*, R. W. Selby, Jr., and V. R. Basili, May 1985

SEL-85-005, *Software Verification and Testing*, D. N. Card, E. Edwards, F. McGarry, and C. Antle, December 1985

SEL-85-006, *Proceedings of the Tenth Annual Software Engineering Workshop*, December 1985

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

SEL-86-002, *General Object-Oriented Software Development*, E. Seidewitz and M. Stark, August 1986

SEL-86-003, *Flight Dynamics System Software Development Environment (FDS/SDE) Tutorial*, J. Buell and P. Myers, July 1986

SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986

SEL-86-005, *Measuring Software Design*, D. N. Card et al., November 1986

SEL-86-006, *Proceedings of the Eleventh Annual Software Engineering Workshop*, December 1986

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

SEL-87-002, *Ada® Style Guide (Version 1.1)*, E. Seidewitz et al., May 1987

SEL-87-003, *Guidelines for Applying the Composite Specification Model (CSM)*, W. W. Agresti, June 1987

SEL-87-004, *Assessing the Ada® Design Process and Its Implications: A Case Study*, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987

SEL-87-010, *Proceedings of the Twelfth Annual Software Engineering Workshop*, December 1987

SEL-88-001, *System Testing of a Production Ada Project: The GRODY Study*, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988

SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby and L. Esker, December 1988

SEL-88-004, *Proceedings of the Thirteenth Annual Software Engineering Workshop*, November 1988

SEL-88-005, *Proceedings of the First NASA Ada User's Symposium*, December 1988

SEL-89-002, *Implementation of a Production Ada Project: The GRODY Study*, S. Godfrey and C. Brophy, September 1989

SEL-89-004, *Evolution of Ada Technology in the Flight Dynamics Area: Implementation/Testing Phase Analysis*, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, *Lessons Learned in the Transition to Ada From FORTRAN at NASA/Goddard*, C. Brophy, November 1989

SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989

SEL-89-007, *Proceedings of the Fourteenth Annual Software Engineering Workshop*, November 1989

SEL-89-008, *Proceedings of the Second NASA Ada Users' Symposium*, November 1989

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

SEL-89-301, *Software Engineering Laborary (SEL) Database Organization and User's Guide (Revision 3)*, L. Morusiewicz, December 1993

SEL-90-001, *Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide*, M. Buhler, K. Pumphrey, and D. Spiegel, March 1990

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

SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL)*, L. O. Jun and S. R. Valett, June 1990

SEL-90-004, *Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary*, T. McDermott and M. Stark, September 1990

SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990

SEL-90-006, *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November 1990

SEL-91-001, *Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules*, W. Decker, R. Hendrick, and J. Valett, February 1991

SEL-91-003, *Software Engineering Laboratory (SEL) Ada Performance Study Report*, E. W. Booth and M. E. Stark, July 1991

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

SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991

SEL-91-006, *Proceedings of the Sixteenth Annual Software Engineering Workshop*, December 1991

SEL-91-102, *Software Engineering Laboratory (SEL) Data and Information Policy (Revision 1)*, F. McGarry, August 1991

SEL-92-001, *Software Management Environment (SME) Installation Guide*, D. Kistler and K. Jeletic, January 1992

SEL-92-002, *Data Collection Procedures for the Software Engineering Laboratory (SEL) Database*, G. Heller, J. Valett, and M. Wild, March 1992

SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992

SEL-92-004, *Proceedings of the Seventeenth Annual Software Engineering Workshop*, December 1992

SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993

SEL-93-002, *Cost and Schedule Estimation Study Report*, S. Condon, M. Regardie, M. Stark, et al., November 1993

SEL-93-003, *Proceedings of the Eighteenth Annual Software Engineering Workshop*, December 1993

SEL-94-001, *Software Management Environment (SME) Components and Algorithms*, R. Hendrick, D. Kistler, and J. Valett, February 1994

SEL-RELATED LITERATURE

[10]Abd-El-Hafiz, S. K., V. R. Basili, and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components," *Proceedings of the IEEE Conference on Software Maintenance-1991 (CSM 91)*, October 1991

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," *Program Transformation and Programming Environments*. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[8]Bailey, J. W., and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability," *Proceedings of the Eighth Annual National Conference on Ada Technology*, March 1990

[10]Bailey, J. W., and V. R. Basili, "The Software-Cycle Model for Re-Engineering and Reuse," *Proceedings of the ACM Tri-Ada 91 Conference*, October 1991

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," *ASME Advances in Computer Technology*, January 1980, vol. 1

Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering*. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," *Proceedings of the First Pan-Pacific Computer Conference*, September 1985

[7]Basili, V. R., *Maintenance = Reuse-Oriented Software Development*, University of Maryland, Technical Report TR-2244, May 1989

[7]Basili, V. R., *Software Development: A Paradigm for the Future*, University of Maryland, Technical Report TR-2263, June 1989

[8]Basili, V. R., "Viewing Maintenance of Reuse-Oriented Software Development," *IEEE Software*, January 1990

[1]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[9]Basili, V. R., G. Caldiera, and G. Cantone, "A Reference Architecture for the Component Factory,"*ACM Transactions on Software Engineering and Methodology*, January 1992

[10]Basili, V., G. Caldiera, F. McGarry, et al., "The Software Engineering Laboratory—An Operational Software Experience Factory," *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE 92)*, May 1992

[1]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[3]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," *Proceedings of the International Computer Software and Applications Conference*, October 1985

[4]Basili, V. R., and D. Patnaik, *A Study on Fault Prediction and Reliability Assessment in the SEL Environment*, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," *Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March 1981

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P—A Prototype Expert System for Software Engineering Management," *Proceedings of the IEEE/MITRE Expert Systems in Government Symposium*, October 1985

Basili, V. R., and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*. New York: IEEE Computer Society Press, 1979

[5]Basili, V. R., and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proceedings of the 9th International Conference on Software Engineering*, March 1987

[5]Basili, V. R., and H. D. Rombach, "T A M E: Tailoring an Ada Measurement Environment," *Proceedings of the Joint Ada Conference*, March 1987

[5]Basili, V. R., 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

[9]Basili, V. R., and H. D. Rombach, "Support for Comprehensive Reuse," *Software Engineering Journal*, September 1991

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[3]Basili, V. R., and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology," *Proceedings of the NATO Advanced Study Institute*, August 1985

[5]Basili, V. R., and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[9]Basili, V. R., and R. W. Selby, "Paradigms for Experimentation and Empirical Studies in Software Engineering," *Reliability Engineering and System Safety*, January 1991

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, July 1986

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983

[2]Basili, V. R., and D. M. Weiss,*A Methodology for Collecting Valid Software Engineering Data*, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," *Proceedings of the Second Software Life Cycle Management Workshop*, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," *Computers and Structures*, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," *Proceedings of the Third International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1978

[9]Booth, E. W., and M. E. Stark, "Designing Configurable Software: COMPASS Implementation Concepts," *Proceedings of Tri-Ada 1991*, October 1991

[10]Booth, E. W., and M. E. Stark, "Software Engineering Laboratory Ada Performance Study—Results and Implications," *Proceedings of the Fourth Annual NASA Ada User's Symposium*, April 1992

[10]Briand, L. C., and V. R. Basili, "A Classification Procedure for the Effective Management of Changes During the Maintenance Process," *Proceedings of the 1992 IEEE Conference on Software Maintenance (CSM 92)*, November 1992

[10]Briand, L. C., V. R. Basili, and C. J. Hetmanski, "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

[11]Briand, L. C., V. R. Basili, and C. J. Hetmanski, *Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components*, TR-3048, University of Maryland, Technical Report, March 1993

[9]Briand, L. C., V. R. Basili, and W. M. Thomas,*A Pattern Recognition Approach for Software Engineering Data Analysis*, University of Maryland, Technical Report TR-2672, May 1991

[11]Briand, L. C., S. Morasca, and V. R. Basili, "Measuring and Assessing Maintainability at the End of High Level Design," *Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM 93)*, November 1993

[11]Briand, L. C., W. M. Thomas, and C. J. Hetmanski, "Modeling and Managing Risk Early in Software Development," *Proceedings of the Fifteenth International Conference on Software Engineering (ICSE 93)*, May 1993

[5]Brophy, C. E., W. W. Agresti, and V. R. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," *Proceedings of the Joint Ada Conference*, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," *Proceedings of the Washington Ada Technical Conference*, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

[3]Card, D. N., "A Software Technology Evaluation Program," *Annais do XVIII Congresso Nacional de Informatica*, October 1985

[5]Card, D. N., and W. W. Agresti, "Resolving the Software Science Anomaly," *Journal of Systems and Software*, 1987

[6]Card, D. N., and W. W. Agresti, "Measuring Software Design Complexity," *Journal of Systems and Software*, June 1988

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, February 1986

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D. N., F. E. McGarry, and G. T. Page, "Evaluating Software Engineering Technologies," *IEEE Transactions on Software Engineering*, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," *ACM Software Engineering Notes*, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," *Proceedings of the Seventh International Computer Software and Applications Conference*. New York: IEEE Computer Society Press, 1983

Doubleday, D., *ASAP: An Ada Static Source Code Analyzer Program*, University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S., and C. Brophy, "Experiences in the Implementation of a Large Ada Project," *Proceedings of the 1988 Washington Ada Symposium*, June 1988

[5]Jeffery, D. R., and V. Basili, *Characterizing Resource Data: A Model for Logical Association of Software Data*, University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," *Proceedings of the Tenth International Conference on Software Engineering*, April 1988

[11]Li, N. R., and M. V. Zelkowitz, "An Information Model for Use in Software Management Estimation and Prediction," *Proceedings of the Second International Conference on Information Knowledge Management*, November 1993

[5]Mark, L., and H. D. Rombach, *A Meta Information Base for Software Engineering*, University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L., and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[5]McGarry, F. E., and W. W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," *Proceedings of the Sixth Washington Ada Symposium (WADAS)*, June 1989

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," *Proceedings of the Hawaiian International Conference on System Sciences*, January 1985

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," *Proceedings of the Eighth International Computer Software and Applications Conference*, November 1984

[5]Ramsey, C. L., and V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management," *IEEE Transactions on Software Engineering*, June 1989

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, March 1987

[8]Rombach, H. D., "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990

[9]Rombach, H. D., "Software Reuse: A Key to the Maintenance Problem," *Butterworth Journal of Information and Software Technology*, January/February 1991

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," *Proceedings From the Conference on Software Maintenance*, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[7]Rombach, H. D., and B. T. Ulery, *Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL*, University of Maryland, Technical Report TR-2252, May 1989

[10]Rombach, H. D., B. T. Ulery, and J. D. Valett, "Toward Full Life Cycle Control: Adding Maintenance Measurement to the SEL," *Journal of Systems and Software*, May 1992

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," *Proceedings of the CASE Technology Conference*, April 1988

[9]Seidewitz, E., "Object-Oriented Programming Through Type Extension in Ada 9X," *Ada Letters*, March/April 1991

[10]Seidewitz, E., "Object-Oriented Programming With Mixins in Ada," *Ada Letters*, March/April 1992

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[9]Seidewitz, E., and M. Stark, "An Object-Oriented Approach to Parameterized Software in Ada," *Proceedings of the Eighth Washington Ada Symposium*, June 1991

[8]Stark, M., "On Designing Parametrized Systems Using Ada," *Proceedings of the Seventh Washington Ada Symposium*, June 1990

[11]Stark, M., "Impacts of Object-Oriented Technologies: Seven Years of SEL Studies," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, September 1993*

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," *Proceedings of TRI-Ada 1989*, October 1989

[5]Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," *Proceedings of the Joint Ada Conference*, March 1987

[10]Straub, P. A., and M. V. Zelkowitz, "On the Nature of Bias and Defects in the Software Specification Process," *Proceedings of the Sixteenth International Computer Software and Applications Conference (COMPSAC 92)*, September 1992

[8]Straub, P. A., and M. V. Zelkowitz, "PUC: A Functional Specification Language for Ada," *Proceedings of the Tenth International Conference of the Chilean Computer Science Society*, July 1990

[7]Sunazuka, T., and V. R. Basili, *Integrating Automated Support for a Software Management Cycle Into the TAME System*, University of Maryland, Technical Report TR-2289, July 1989

[10]Tian, J., A. Porter, and M. V. Zelkowitz, "An Improved Classification Tree Analysis of High Cost Modules Based Upon an Axiomatic Definition of Complexity," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

Turner, C., and G. Caron, *A Comparison of RADC and NASA/SEL Software Development Data*, Data and Analysis Center for Software, Special Publication, May 1981

[10]Valett, J. D., "Automated Support for Experience-Based Software Management," *Proceedings of the Second Irvine Software Symposium (ISS '92)*, March 1992

[5]Valett, J. D., and F. E. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," *IEEE Transactions on Software Engineering*, February 1985

[5]Wu, L., V. R. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," *Proceedings of the Joint Ada Conference*, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium-Scale Software Projects," *Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science*. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," *Empirical Foundations for Computer and Information Science* (Proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," *Proceedings of the 26th Annual Technical Symposium of the Washington, D. C., Chapter of the ACM*, June 1987

[6]Zelkowitz, M. V., "Resource Utilization During Software Development," *Journal of Systems and Software*, 1988

[8]Zelkowitz, M. V., "Evolution Towards Specifications Environment: Experiences With Syntax Editors," *Information and Software Technology*, April 1990

NOTES:

[1] This article also appears in SEL-82-004, *Collected Software Engineering Papers: Volume I*, July 1982.

[2] This article also appears in SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983.

[3] This article also appears in SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985.

[4] This article also appears in SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986.

[5] This article also appears in SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987.

[6] This article also appears in SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988.

[7] This article also appears in SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989.

[8] This article also appears in SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990.

[9] This article also appears in SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991.

[10] This article also appears in SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992.

[11] This article also appears in SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993.

PAGE 492 INTENTIONALLY BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
November 1993

3. REPORT TYPE AND DATES COVERED
Contractor Report

4. TITLE AND SUBTITLE
Proceedings of the Eighteenth Annual Software Engineering Workshop

5. FUNDING NUMBERS
552

6. AUTHOR(S)
Software Engineering Laboratory

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland

8. PERFORMING ORGANIZATION REPORT NUMBER
SEL-93-003

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, D.C. 20546–0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
CR-189348

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Eighteenth Annual Software Engineering Workshop, sponsored by the Software Engineering Laboratory (SEL), was held on 1 and 2 December 1993 at the National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center in Greenbelt, Maryland.

The Workshop provides a forum for software practitioners from around the world to exchange information on th measurement, use, and evaluation of software methods, models, and tools. This year, approximately 450 peopl attended the Workshop, which consisted of six sessions on the following topics: The Software Engineering Lal tory, Measurement, Technology Assessment, Advanced Concepts, Process, and Software Engineering Issues Three presentations were given in each of the topics areas. The content of those presentations and the resea detailing the work reported are included in these *Proceedings*. The Workshop concluded with a tutorial ses to start an Experience Factory.

14. SUBJECT TERMS

15

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

Sta.

PAGE BLANK NOT FILMED