PROCEEDINGS OF THE TENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

DECEMBER 1985

National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

PROCEEDINGS

OF

TENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

Organized by:

Software Engineering Laboratory
GSFC

December 4, 1985

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

FOREWARD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC (Systems Development and Analysis Branch)

The University of Maryland (Computer Sciences Department)

Computer Sciences Corporation (Flight Systems Operation)

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

Single copies of this document can be obtained from

Ms. Tillery
NASA Scientific And Technical Installation Facility
P.O. Box 8757
B.W.I. Airport, Md 21240

AGENDA

TENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP
NASA/GODDARD SPACE FLIGHT CENTER
BUILDING 8 AUDITORIUM
DECEMBER 4, 1985

8:00 a.m.	Registration - 'Sign-In' Coffee, Donuts	
8:45 a.m.	INTRODUCTORY REMARKS	J. J. Quann, Deputy Director (NASA/GSFC)
9:00 a.m.	Session No. 1	Topic: Research in the Software Engineering Laboratory (SEL)
		Discussant: J. Page (CSC)
	"Can We Measure Software Technology; Lessons from 8 Years of Trying"	V. Basili (Univ. of Maryland)
	"Recent SEL Studies"	F. E. McGarry (NASA/GSFC)
10:00 a.m.	BREAK	
10:30 a.m.	Session No. 2	Topic: Tools for Software Management
		Discussant: D. Card (CSC)
	"Software Management Tools: Lessons Learned From Use"	D. Reifer (RCI)
	"DEASEL: An Expert System for Software Engineering"	J. Valett (NASA/GSFC) A. Raskin (Yale)
	"An Experimental Evaluation of Error Seeding as a Program Validation Technique"	J. Knight (Univ. of Virginia) P. Ammann
	"Quality Assurance Software Inspections at NASA Ames"	G. Wenneson (Informatics)
12:30 p.m.	LUNCH	

1:30 p.m.	Session No. 3	Topic: Software Environments
		Discussant: E. Katz (Univ. of Maryland)
	"A Knowledge Based Software Engineering Environment Testbed"	C. Gill (BCS)
	"Experience with a Software Engineering Environment Framework"	R. Blumberg (PRC) A. Reedy E. Yodis
	"One Approach for Evaluating the Distributed Computing Design System (DCDS)"	L. Baker (TRW)
3:00 p.m.	BREAK	
3:30 p.m.	Session No. 4	Topic: Experiments with Ada
		Discussant: E. Seidewitz (NASA/GSFC)
	"An Ada Experiment with MSOCC Software"	D. Roy (Century Computing)
	"Observations From a Prototype Implementation of the Common APSE Interface Set (CAIS)"	M. McClimens (Mitre)
	"Measuring Ada as a Software Development Technology in the SEL"	B. Agresti (CSC)
5:00 p.m.	ADJOURN	
	Results of the SEL Workshop Questionnaire will be Found at the End of the Proceedings	

SUMMARY OF THE TENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

Prepared for

GODDARD SPACE FLIGHT CENTER

by

L. Jordan

COMPUTER SCIENCES CORPORATION

The Tenth Annual Software Engineering Workshop was held on December 4, 1985, at the National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center (GSFC) in Greenbelt, Maryland. This annual meeting is held to report and discuss experiences in the measurement, utilization, and evaluation of software methods, models, and tools. The workshop was organized by the Software Engineering Laboratory (SEL), whose members represent NASA/GSFC, the University of Maryland, and Computer Sciences Corporation (CSC). The workshop was conducted in four sessions:

- Research in the SEL
- Tools for Software Management
- Software Environments
- Experiments with Ada

Twelve papers were presented, and the audience actively participated in all discussions through general commentary, questions, and interaction with the speakers. Over 400 persons representing 55 private corporations, 6 universities, and 27 agencies of the Federal Government attended the workshop.

John J. Quann, Deputy Director of NASA/GSFC, noted in his opening remarks that programs such as this workshop are very important for the exchange of ideas to improve software development and products. This is especially due to the increasing interest in software engineering (e.g., the procurement of a Space Station software support environment (SSE) by Johnson Space Center), the growth of the Space Station Program, and the increasing use of Ada. Mr. Quann also noted that in the future, the workshop may need to be expanded to 1-1/2 to 2 days and include representatives of the international community.

Because this workshop represented the tenth anniversary of the SEL, the major theme of the first session, Research in

the SEL, consisted of an overview of the SEL experimentation process and a summary of recent studies completed. In his introduction to the session, Dr. Gerald Page of CSC discussed the background of the SEL, its structure, the development characteristics of SEL software, and the scope of SEL activities. The SEL was formally established in 1976 by NASA/GSFC to improve its software development process and products by measuring the software development process, evaluating existing technologies, and transferring successful technologies into the development environment at NASA/GSFC. The software studied within the SEL environment is primarily scientific, ground-based, interactive, near-real-time software written primarily in FORTRAN (85 percent) on IBM mainframes. The typical project is 65 K source lines of code (SLOC) (2 to 160 KSLOC) in size and takes 16 to 25 months (from start of design to start of operations) with 6 to 18 people to complete. Data have been collected by the SEL for more that 50 projects that represent over 2 million LOC produced by over 200 developers and reported by over 30,000 forms submitted. About 50 state-of-the-art technologies have been studied and many tools, standards, and models for use by developers have been produced.

Dr. William Agresti of CSC presented the results of a questionnaire that was circulated to the meeting attendees. The questionnaire was intended to help mark the tenth anniversary of the workshop and requested information from the respondents concerning their

- Role in software development
- Data collection activity
- Perception of changes in software quality

- Opinions regarding progress (or lack of it) in various areas of software engineering

The results are presented elsewhere in these proceedings.

Dr. Victor Basili of the University of Maryland drew on the 10-year history of the SEL to present SEL experience in the area of measurement (Measuring the Software Process and Product: Lessons Learned by the SEL). He noted that there are many reasons for collecting data that measure the software development process and products. These reasons include the establishment of a corporate memory (e.g., for planning), the determination of strengths and weaknesses of current methodologies and technologies, and the determination of a rationale for adopting new technologies. There are also different aspects to measurement, including software characteristics, development resources, and errors. These aspects thus represent many classes of project data. The most important lessons learned by the SEL in this area revolve around the development of a goal-driven paradigm for data collection. The reasons for collecting data must be clearly defined at the detailed level to avoid collection of too much or inappropriate data. This requires a clear characterization of data in terms of explicit goals (e.g., what phase was the greatest source of error) and metrics (e.g., error distribution by phase). Dr. Basili defined six steps for the data collection process:

- Generate a set of goals
- Derive a set of questions or hypotheses to quantify the goals
- Develop a set of metrics to answer the questions
- Define a mechanism to collect the data as accurately as possible

- Validate the data
- Analyze the data to answer the questions

He then discussed a goal-setting template in terms of purpose (to characterize, evaluate, etc.), perspective, environment, and hierarchy of perspective. A subtemplate included the definition of the process (i.e., quality of use, domain of use, cost, effectiveness), feedback (lessons learned, model validation), the product, and the perspective.

Regarding the successes and failures for the SEL, Dr. Basili noted that the effort data have been good (but can be improved) and have led to the development of good cost models. Error data have been good on occurrence (history of errors and changes can be tracked) but have been poor for specifics (detailed technique information for error detection is not easily available). Project characteristics are accurately recorded, but recording problem characteristics is difficult. Technology data are good for level of use for the overall methodology, but it is difficult to isolate the individual impact. In terms of the cost of data collection for the SEL,

- Direct cost can be less that 3 percent
- Processing cost is 5 percent or greater
- Analysis cost is 15 to 20 percent (includes interpretation, reporting, research support, publication of papers, and technology transfer)

In response to questions, Dr. Basili indicated that some measurement could be automated (this may include some aspects of software quality--productivity, reliability, and maintainability--and overall records) and that the cost of data collection does include corrective action in the areas of documents, standards, and training. Some discussion of the Rome Air Development Center work followed the discussion.

Mr. Frank E. McGarry of GSFC presented an overview of 10 years of SEL research and a more detailed look at specific research projects in the last 2 years (Studies and Experiments in the Software Engineering Laboratory). SEL research in four areas has recently concentrated on the following:

- Tools and environments--Management tools and programming environments
- Development methods--Testing approaches and Ada studies
- Measures and profiles--Design and specification measures
- Models--Relationship equations

In the measurement of environment (in terms of software tools, computer support for batch versus interactive processing, and the number of terminals per programmer), Mr. McGarry described an experiment using 14 projects that showed

- Positive correlation for tool support and productivity, effort to change, and effort to repair; no correlation with reliability
- No correlation between computer environment and any of the factors measured
- Negative correlation between terminals per programmer and productivity and reliability; no correlation with effort to change or effort to repair

He described an experiment to determine the characteristics of functional testing in an acceptance testing environment and compare the test profile with operational usage. The characteristics used were percent of code and modules

executed and the profiles of errors found. A single flight dynamics program with 10 functional test and 60 operational use cases yielded results showing that functional testing during acceptance testing is very representative of operational usage.

Mr. McGarry then described an experiment using 3 FORTRAN programs seeded with faults that were tested by 32 professional programmers using 3 verification techniques (code reading, functional testing, and structural testing). The results showed code reading to be the best technique in terms of faults detected (code reading, 61 percent; functional testing, 51 percent; structural testing, 38 percent) and number of faults detected per hour of effort (code reading, 3.3; functional testing, 1.8; structural testing, 1.8). Another analysis of testing techniques versus size showed that functional testing may be more effective for larger programs.

In the area of software design measures, Mr. McGarry presented study results that showed the effects of module strength (types and numbers of module functions), size, and coupling (parameter, mixed, and COMMON) on costs and errors. Based on 450 FORTRAN modules and about 20 developers, the fault rate was zero for 50 percent of the high-strength modules and 18 percent of the low-strength modules. A high fault rate was found for 20 percent of the high-strength modules and 44 percent of the low-strength modules. The analysis for size showed a slightly higher percentage of fault-prone modules for small modules (36 percent) than for medium (29 percent) or large modules (27 percent). The parameter coupling modules had a higher percentage of fault-prone modules (40 percent) than either the mixed (29 percent) or the COMMON (30 percent) coupling types. Overall, good programmers tend to write high-strength modules with no preference for size. High-strength modules have a lower

fault rate and cost less than low-strength modules, and large modules cost less (per executable statement) than small ones. The fault rate does not appear to be directly related to size.

In the area of computer use and technology over time, Mr. McGarry defined a technology index and applied it to projects that started between 1976 and 1982. Computer use has increased from 130 runs per KLOC to 235 runs per KLOC, and the technology index has increased from 90 to 140. There is no significant correlation between computer use and the technology index. In other specific areas:

- Software reuse is increasing over time and appears to have significant potential as a technology.
- The total technology index has a favorable effect on reliability but no obvious correlation with productivity (productivity is too sensitive to too many other factors).
- Individual techniques are difficult to measure.
- Integrated methodologies have a favorable effect on quality.

Responding to questions, Mr. McGarry clarified several points about the detailed methods used in the experiment that compared the 3 software testing techniques, and he emphasized that code reading could not be substituted for acceptance testing. He also indicated that the 32 programmers participating in the study did not seem to be affected (Hawthorne effect) by the monitoring of the experiment. He stated that these results differed with those of Myers because of a difference in the definition of code reading. On the issue of terminal use versus productivity, he felt that more terminals available resulted in more concurrent tasks so that productivity suffered more when the terminals were

down. This effect may also be caused by the lack of a disciplined approach with respect to terminal use and may be corrected with time and effort.

The topic of the second session was Tools for Software Management. Mr. Donald Reifer of Reifer Consultants, Inc., discussed experiences in inserting software project planning tools into more than 100 projects producing mission-critical software and in using a Project Manager's Workstation (PMW) and a SoftCost-R cost estimation package (Software Management Tools: Lessons Learned From Use). He defined the management process as beginning with planning, organizing, and staffing a team and then in communicating, motivating, integrating, measuring, controlling, and directing the efforts of the team through an iterative process. He listed a number of necessary tools in the contexts of the company's system, project management, functional management, and line management. Over 300 packages exist to support these functions. Managers tend not to use tools because of time pressures (too busy to learn and to use them) and because the tools do not fit into the existing system. A need to overcome this problem is recognized by the STARS program in attempting to develop management tools to eliminate paperwork in such areas as scheduling.

PMW is an experimental system to integrate several tools into a package to do scheduling, graphing (e.g., PERT), and reporting in a variety of areas. Mr. Reifer found that the manager/machine interface must be user-friendly (picture oriented, function key driven, and menu based) and that the package must be easy to learn and have built-in safeguards and help facilities (managers do not read manuals). The problem of initial data entry is severe; managers do not have the time to do it and subordinates do not have the knowledge. In general, Mr. Reifer noted that vendors do not implement all the features in their manuals or make it easy

to interface their packages with other packages. He found that the most useful tools are work-planning oriented, the most used tools are time-management oriented, and the most wanted tools are what-if oriented.

SoftCost-R is a package that generates schedule and resource estimates for about 50 tasks making up a project. Based on about 60 sizing and productivity factors, it computes a confidence factor for delivering on time and within budget, produces a standard work breakdown structure for software development tasks, and provides a capability for what-if analysis and plotting. Mr. Reifer found that organizational preconditioning is necessary. Data are not generally available in most companies for using SoftCost-R to develop calibrations for the models or to validate them. There is no existing framework that can supply these tools with the needed information. Application of cost models has, in some cases, forced changes in business practice that seemed disruptive, but were really not. Calibrating the models to the organization is difficult. Model architectures must expose calibration points and sensitivities, and these must be easily altered, since organizations are dynamic. Users often rely too much on models without understanding their scope or limitations. Also, users often do not believe model results (find it difficult to face or believe unpleasant truths).

In response to a question, Mr. Reifer noted that vendors should add a user-friendly demonstration that shows a manager how to get what he wants. He said that, in some cases, these demonstrations can be obtained by writing and that the cost of the demonstration is subsequently subtracted from the cost of the package. In summary, he noted that vendors should pay as much attention to packaging as to functions and features, should make systems manager-friendly and not programmer-friendly, and should provide what-if capability

and a lot of small useful tools. Users should not assume vendors deliver what is advertised, should worry about bridging between packages and not assume it is easily done, and should realize that tools may act as a catalyst for organizational change.

In response to questions regarding bridging applications, Mr. Reifer suggested two strategies: (1) build a data repository that is usable by different tools and (2) get tools that adhere to standard formats. He also noted some possible advantages of SoftCost-R over the widely used COCOMO: SoftCost-R is suited for mission-critical software, covers reused code, provides cradle-to-the-grave project coverage, provides adequate support for parametric and statistical studies. COCOMO does not.

Mr. Jon Valett of GSFC described a tool that combines the SEL data base and a manager's experience to support project estimation and development progress assessment in the flight dynamics environment (DEASEL: An Expert System for Software Engineering). Managers were interviewed in an effort to capture their experience and combine it with specific SEL data to form the knowledge base. The system is defined in terms of rules (factors and weights) and assertions to assess projects. The rules define relationships and weights between specific parameters and system goals (e.g., change rate and design stability). Assertions provide actual values of parameters for a specific project that are then used to compute an assessment of the project compared to system goals in terms of a rating (good to bad) and a confidence factor. The current system is applicable to the design phase and uses 25 rules. It can provide project assessments, explain the assessment, and provide what-if analysis. Current plans are to add rules for other development phases, to validate the existing rules and the current assessment

process, and to catenate the generation of assertions. In response to questions, he indicated that the development environment was VAX and LISP.

Dr. John C. Knight of the University of Virginia described an experiment that seeded errors into 27 functionally identical programs to assess error seeding as a technique for validating programs (An Experimental Evaluation of Error Seeding as a Program Validation Technique). He noted as background that verification is preferred to testing but that it is usually not feasible and is subject to error. In answering the question of when testing should stop, he indicated that testing typically stops when the money is gone or when the project runs out of time.

The classical error seeding approach relies on a relationship between indigenous error and seeded error discovery that assumes the following:

- Indigenous errors are hard to find.
- Indigenous and seeded errors are independent.
- Seeded errors are as hard to find as indigenous errors.

Dr. Knight noted that the last assumption is obviously false because indigenous errors are subtle, and high-powered artificial intelligence methods are required to generate equally subtle errors for seeding.

For this experiment, simple seeding algorithms were applied to FOR, IF, and Assignment statements. The 27 functionally identical programs consisted of 327 to 1004 lines of Pascal code. Seeding algorithms were applied 4 times to each program to produce a total of 108 seeded programs. The programs were subjected to 1 million test cases. Dr. Knight found that a surprising number of seeded errors were found only after thousands of tests and that they were actually

being successfully executed (in one case, a seeded error corrected a bug). His evaluation of the three assumptions was that they were all questionable. He also stated that the assumption of N-Version Programming, that independently written programs will fail independently, is false. This conclusion is based on his finding that many different types of errors can produce similar patterns of failure.

In response to questions, Dr. Knight noted that the class of seeded errors was very small compared to the class of indigenous errors and that robust testing techniques do not eliminate long-mean-time-to-failure errors. Simple errors may survive 10,000 tests before being located. He said that random test generation was used for his experiment and that scientific testing might have done better.

Mr. Greg Wenneson of Informatics General Corporation described procedures to control software quality (Software Inspections at NASA Ames). Productivity gains of 40 percent have been realized through the use of these inspection procedures (compared to 23 percent reported by IBM), based on one program that was rewritten and that includes major methodology changes. Inspection tools include standards, material preparation criteria, error checklists, exit criteria, and written records and statistics. The team members are the moderator, reader, inspectors, and the author. The inspection process comprises team selection, overview, preparation, inspections sessions (may be desk inspections), rework, and followup. Mr. Wenneson also defined problem recording (module inspection problem report, general problems report), problem statistics (module problem summary, module time and disposition report), and inspection statistics (inspector time report, inspection general summary, outline of rework schedule).

For FORTRAN modules, 144 problems were reported per KLOC for preliminary design, 227 for detailed design, 67 for desk-inspected code, and 83 for regular inspection. Effort for this activity was 15 person-weeks per KLOC for preliminary design, 24 for detailed design, 4 for desk-inspected code, and 9 for regular inspection. The number of previous inspections affect both the error rates and preparation and meeting time. The major error rate of 30 per KLOC for 1 previous inspection increases to 38 for 3 previous inspections. Preparation and meeting time increases from 9.2 person-weeks per KLOC for 1 previous inspection to 10 for 3 previous inspections.

In his summary, Mr. Wenneson emphasized that inspections are not a substitute for thinking; that they must be scheduled at the beginning of a project (and not just tacked on); and that participant training and customer and management support are crucial. Future plans include application to new languages and design techniques, expansion to new methodologies and support tools, inclusion of feedback to current methodologies, and expansion to other application areas.

In the following panel discussion, Mr. Wenneson stated that the system used for his example consisted of about 5 percent assembly language modules and that the assembly language numbers for design in his presentation related to the target language rather than to the design language. For downstream savings, he said that, although his statistics stop at the end of coding, other sources indicate that errors cost less to repair. Desk inspection found 80 percent of the errors found by regular inspection but cost 40 percent less. As a guildeline, he suggested that a project of less than 1000 LOC should not be split into too many pieces and that 50 to 100 LOC should be represented by 1 line of design.

The topic of the third session was Software Environments. Mr. Chris Gill of Boeing Computer Services described a research project to apply artificial intelligence to software engineering (A Knowledge-Based Software Engineering Environment Testbed). The multiyear project has completed its first year. The objectives are to determine the benefits of applying artificial intelligence to software engineering, demonstrate improvements in the software development process and in software quality, and develop a test bed for experimentation. The system consists of an integrated set of tools covering the entire life cycle (analysis, design, and production) and several areas of effort (project management, software development support, and configuration management). The knowledge base is derived from procedures and interviews. The knowledge representation deals with modeling software project concepts and links. Inference mechanisms deal with the ways this knowledge can be used to solve user development problems. The knowledge-based interface deals with the intelligent display, explanation, and interaction with the user.

After one year, a model of software development activities has been created, and the groundwork has been done in the module representation formalism to specify the behavior and structure of software objects. The model and formalism have been integrated to identify shared representation and inheritance mechanisms. Object programming has been demonstrated by writing procedures and applying them to software objects (e.g., by propagating changes) in a development system. Data-directed reasoning has been used to infer the probable cause of bugs by interpreting problem reports. Goal-directed reasoning has been used to evaluate the appropriateness of a software configuration. Plans for next year include using knowledge-based simulations to perform rapid prototyping, enhancing the user interface, using a "blackboard"

architecture to allow experts to confer, and using distributed systems to permit separate systems to act on goals sent by other systems.

In his conclusion, Mr. Gill stated that the project showed promise. It provides leverage of integration, because data are keyed in only once. There is, however, a need to apply it to real systems. In the following discussion, he indicated that the system contains several hundred rules for scheduling and task management. The current demonstration uses the graphics and reasoning (e.g., manager experience versus complexity) capabilities. Most of the current capabilities relate to specification and design.

During the panel discussion after the session, it was mentioned that there are currently seven projects using artificial intelligence approaches to software environments (five in Japan, and two in England). The system reported by Mr. Gill is the first heard of in the United States.

Ms. Ann Reedy of Planning Research Corporation described an automated product control environment developed to reduce life-cycle costs and increase automation of the software development process (Experience With a Software Engineering Environment Framework). This framework is not composed of tools, but provides for overall control, coordination, and enforcement. It provides automation of real-time status tracking and reporting; configuration management of software, documents, and test procedures; traceability of requirements and change effects; testbed generation; and component and system integration. It deals with people (managers, developers, testers, and QA), processes (phases and integration levels), and products (software, documents, and test procedures). The system was designed to be portable (currently runs on the VAX-11/780 with VMS, on ROLM and Data General with AOS/VS, on IBM with MVS, and on Intel with

XENIX). In the area of distributability and interoperability, the tool sets for different hosts may be different but the functionality is assumed to be the same (the framework only operates on tool products and does not contain tools itself). Filters and standard forms can be used for adjustment.

Ms. Reedy reported productivity figures for 3 projects ranging from 121 to 384 LOC per day. In terms of level of effort, she reported first-year resource costs for the manual environment of 56 staff-months versus 29 for the automated environment. Annual recurring costs were 60 staff-months for the manual environment versus 24 for the automated environment. Cumulative costs for 24 project-months were $900,000 for manual implementation versus $500,000 for automated implementation. After the presentation, there was some spirited discussion on the productivity figures cited.

Mr. Lloyd Baker of TRW Defense Systems Group reported on an evaluation of an integrated environment for the specification and life-cycle development of software (One Approach for Evaluating the Distributed Computing Design System (DCDS)). DCDS consists of integrated methodologies, languages, and an integrated tool set. Users can produce specifications for system requirements, software requirements, distributed architectural designs, detailed module designs, and tests. Five languages support the concepts for each of the methodologies and are used to express the requirements, designs, and tests. All languages use the same constructs and syntax. (More information on the operation of DCDS is available in the April 1985 issue of IEEE Computer Magazine.)

DCDS was compared with three other commercially available products using a list of evaluation criteria partitioned into three classes:

- Factors lending credibility to the product
- Costs of acquiring and using the product
- Benefits of the product

The criteria were weighted (high, medium, low), and the products were scored and evaluated (better, acceptable, deficient). Development costs included costs for learning the system, documenting results, and fixing errors, as well as normal development work. Mr. Baker presented the detailed evaluation results for each of the systems for 21 different factors.

The topic of the last session was Experiments with Ada. Mr. Dan Roy of Century Computing, Inc., presented an assessment of a 1200-line (of Ada code) project that used George Cherry's Process Abstraction Methodology for Embedded Large Applications (PAMELA) and DEC's Ada Compilation System (ACS) under VAX/VMS (An Ada Experiment With MSOCC Software). The requirements analysis was performed with the standard De Marco structured analysis. Ada was used as a data definition language to produce a data dictionary during the requirements phase. A special package (the TBD package) aided the top-down design of the data structure. Preliminary and detailed design templates were created and proved very useful. Ada was used as a program design language (PDL) that was then refined into detailed code in the normal staged manner. The tools and templates for Ada constructs (developed at the start of the project) had a dramatic effect on productivity and code consistency (30 LOC per day during development, 13 LOC per day from cradle to grave). Ada training was difficult and complex (none of the standard training devices alone were adequate). He tried a number

of compilers with poor results before going to ACS and achieving results reasonably approximating FORTRAN compiler speeds and acceptable quality.

Mr. Mike McClimens of MITRE Corporation described an experiment to study a standard CAIS implementation (Observations from a Prototype Implementation of the Common APSE Interface Set (CAIS)). CAIS is a tool interface to operating systems that encapsulates machine dependencies such as data base access. He first described the background and history of its development. CAIS is defined as a set of Ada package specifications and a description of associated semantics. The underlying model is a directed graph with attributes. Nodes can be files, processes, or directories. Both graph nodes and edges have attributes. CAIS provides node management, process management (spawn/invoke, abort/suspend/resume), I/O (text, direct, sequential, scroll and page for devices), and list utilities (abstract data type, heterogeneous list of items). It does not provide support for concurrency, memory management, or interrupts for Ada or scheduling, paging/segmentation, or low-level I/O for operating systems or a data base management system.

Mr. McClimans then described a number of objectives for work on the system during 1985 and the technical approach used to attain those objectives. He noted that the learning curve for CAIS will be significant and that overall conceptual consistency is good.

Dr. William Agresti of CSC described an experiment that is underway in the SEL to develop a system in parallel in Ada and in FORTRAN (Measuring Ada as a Software Development Technology in the SEL). The size of the project is estimated as 40 KSLOC (FORTRAN); it will take from 18 to 24 months to complete with a staff of seven and will require 8 to 10 staff-years of effort. Forms will be collected for the

SEL data base. A study team is providing training, planning, and evaluation. The Ada team is more experienced overall than the FORTRAN team but is less experienced in the particular application. At the time of the presentation, the Ada project was completing design and beginning code and test; the FORTRAN project was completing code and test and beginning integration and system test. The schedule difference is attributed to Ada training. The training material and approaches were described. Training included the development of a small electronic mail system to gain hands-on experience with the Ada language and took 2 months of full-time work.

Dr. Agresti provided statistics describing the training exercise. The electronic mail system was originally developed as 1000 to 2000 SLOC in SIMPL. In Ada, the system was 5730 SLOC (1400 executable statements) and took 1900 hours to develop (including 570 hours of training). The cost was 950 hours per 1000 executable statements (1360 including training) with an error rate of 9 errors per 1000 executable statements; this can be compared with 720 hours and 12 errors per 1000 executable statements for FORTRAN. The distribution of effort for design, code, and test was 60, 18, and 22 percent for Ada and 33, 33, and 34 percent for FORTRAN.

During the panel discussion at the end of the session, it was noted that object-oriented design does not replace PDL. Ada performance seems to be a major issue, and its suitability to various applications must be investigated. The rendezvous on the VAX compiler is 70 times longer than the procedure call, for example. Many of the current areas of poor performance will probably be considerably improved in future implementations, so it is not wise to make major decisions based on current implementations. Tasking and other processes may be slow, but optimization is good for

compiled code and may offset the slow performance. It was also mentioned that, in benchmark testing, The DEC Ada compiler is within 10 to 20 percent of FORTRAN speeds.

PANEL #1

RESEARCH IN THE SOFTWARE ENGINEERING LABORATORY (SEL)

V. Basili, University of Maryland
F. McGarry, NASA/GSFC

Measuring the Software Process and Product:
Lessons Learned in the SEL

Victor R. Basili
Department of Computer Science
University of Maryland

There are numerous reasons to measure the software development process and product. It is important to create a corporate memory in the software area to support planning, e.g. to answer questions about predicting the cost of a new project. We need to determine the strengths and weaknesses of the current process and product, e.g. to determine what types of errors are commonplace. We need to develop a rational for adopting and refining software development and maintenance techniques, e.g. to help us decide what techniques actually minimize current problems. We need to assess the impact of the techniques we are using, e.g. to determine whether our current approach to functional testing actually does minimize certain classes of errors, as we might believe it does. Finally, we should evaluate the quality of the software process and product, e.g. to assess the reliability of the product after delivery.

We have tried to address all of these problems to varying degrees within the Software Engineering Laboratory at NASA Goddard Space Flight Center, grouping studies into four general categories: the problem, the process, the product, and the environment. Within these categories, we have concentrated on three aspects of measurement in the SEL: visibility, quality, and technology. With regard to visibility we have tried to better understand how software is being developed by making the current practices and products as visible as possible using measurement. Areas of measurement have been based upon models of the resources, errors, environment, problem and the product. We have tried to assess the quality of the process and product by examining such characteristics as productivity, reliability, maintainability, portability and reusability. Technology has been measured in an attempt to ascertain how much, if at all, certain techniques help in the development and to isolate those practices and tools which improve productivity.

To achieve the goals related to visibility, quality and technology, we have collected a variety of data. Table 1 provides some idea of the type of data collected. The scope of activity in the SEL from 1977 through 1984 is shown in Table 2.

Visibility	Quality	Technology
Resource Data	Productivity	How much do certain techniques help?
Error Data	Reliability	
Environment Characteristics	Maintainability	
	Portability	Which tools improve productivity?
Problem Complexity	Reusability	
Product Data		

Table 1

SEL
1977 - 1984

Number of Projects	41
Number of Source Lines of Code	1.3 million
Development Cost	$11 million
Number of Data Forms	30 thousand

Table 2

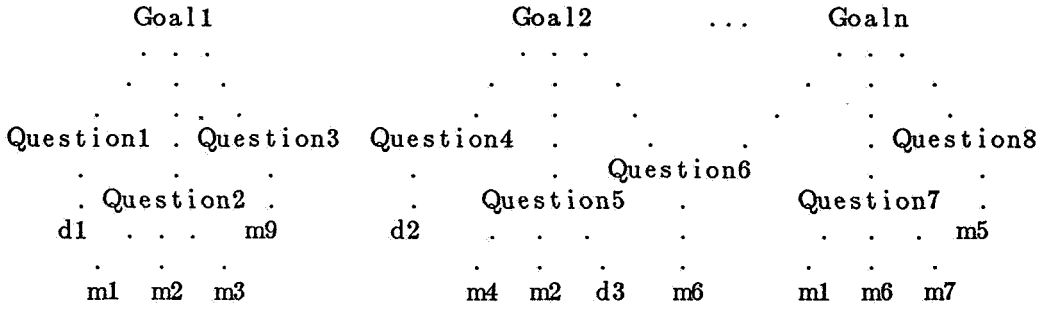
GOAL/QUESTION/METRIC PARADIGM

There have been many lessons learned in the the SEL about measurement but the most important one has been the need for a goal-driven paradigm for data collection. That is data collection must be driven top down. What you measure is based upon a carefully articulated set of goals stating what it is you want to know and whether you can gather the appropriate and valid data needed to answer your questions. Whenever we have violated these rules we either ended up collecting data that was not used or have not been successful in performing our task. For example we have discarded data, such as run analysis data, even though it may be interesting information, it was not associated with a specific goal of the laboratory. Also we have not had success in areas where there was not a carefully focused goal allowing us to control for extraneous effects, e.g. measuring the effectiveness of detailed techniques.

The approach to measurement used in the SEL has been the goal / question / metric paradigm [Basili & Weiss 1984] developed specifically to help us define the areas of study and help in the interpretation of the results of the data collection process. The paradigm does not provide a specific set of goals but rather a framework for stating goals and refining them into specific questions about the software development process and product that provide a specification for the data needed to help answer the goals.

The paradigm provides a mechanism for tracing the goals of the collection process, i.e. the reasons the data are being collected, to the actual data. It is important to make clear at least in general terms the organization's needs and concerns, the focus of the current project and what is expected from it. The formulation of these expectations can go a long way towards focusing the work on the project and evaluating whether the project has achieved those expectations. The need for information must be quantified whenever possible and the quantification analyzed as to whether or not it satisfies the needs. This quantification of the goals should then be mapped into a set of data that can be collected on the product and the process. The data should then be validated with respect to how accurate it is and then analyzed and the results interpreted with respect to the goals.

The actual data collection paradigm can be visualized by a diagram:



Here there are n goals shown and each goal generates a set of questions that attempt to define and quantify the specific goal which is at the root of its goal tree. The goal is only as well defined as the questions that it generates. Each question generates a set of metrics (mi) or distributions

of data (di). Again, the question can only be answered relative to and as completely as the available metrics and distributions allow. As is shown in the above diagram, the same questions can be used to define different goals (e.g. Question6) and metrics and distributions can be used to answer more that one question. Thus questions and metrics are used in several contexts.

Given the above paradigm, the data collection process consists of six steps:

1. Generate a set of goals based upon the needs of the organization.

The first step of the process is to determine what it is you want to know. This focuses the work to be done and allows a framework for determining whether or not you have accomplished what you set out to do. Sample goals might consist of such issues as on time delivery, high quality product, high quality process, customer satisfaction, or that the product contains the needed functionality.

2. Derive a set of questions of interest or hypotheses which quantify those goals.

The goals must now be formalized by making them quantifiable. This is the most difficult step in the process because it often requires the interpretation of fuzzy terms like quality or productivity within the context of the development environment. These questions define the goals of step 1. The aim is to satisfy the intuitive notion of the goal as completely and consistently as possible.

3. Develop a set of data metrics and distributions which provide the information needed to answer the questions of interest.

In this step, the actual data needed to answer the questions are identified and associated with each of the questions. However, the identification of the data categories is not always so easy. Sometimes new metrics or data distributions must be defined. Other times data items can be defined to answer only part of a question. In this case, the answer to the question must be qualified and interpreted in the context of the missing information. As the data items are identified, thought should be given to how valid the data item will be with respect to accuracy and how well it captures the specific question.

4. Define a mechanism for collecting the data as accurately as possible

The data can be collected via forms, interviews, or automatically by the computer. If the data is to be collected via forms, they must be carefully defined for ease of understanding by the person filling out the form and clear interpretation by the analyst. An instruction sheet and glossary of terms should accompany the forms. Care should be given to characterizing the accuracy of the data and defining the allowable error bounds.

5. Perform a validation of the data

The data should always be checked for accuracy. Forms should be reviewed as they are handed in. They should be read by a data analyst and checked with the person filling out the form when questions arise. Sample sets should be set to determine accuracy the data as a whole. As data is entered into the data base, validity checks should be made by the entering program. Redundant data should be collected so checks can be made.

The validity of the data is a critical issue. Interpretations will be made that will effect the entire organization. One should not assume accuracy without justification.

6. Analyze the data collected to answer the questions posed

The data should be analyzed in the context of the questions and goals with which they are associated. Missing data and missing questions should be accounted for in the interpretation.

The process is top down, i.e before we know what data to collect we must first define the reason for the data collection process and make sure the right data is being collected, and it can be interpreted in the right context. To start with a set of metrics is working bottom up and does not provide the collector with the right context for analysis or interpretation.

WRITING GOALS AND QUESTIONS:

In writing down goals and questions, we must begin by stating the purpose of the study. This purpose will be in the form of a set of overall goals but they should follow a particular format.

The format should cover the purpose of the study, the perspective, and any important information about the environment. The format might look like:

Purpose of Study: To (characterize, evaluate, predict, motivate) the (process, product, model, metric) in order to (understand, assess, manage, engineer, learn, improve) it. E.g. To evaluate the system testing methodology in order to assess it.

Perspective: Examine the (cost, effectiveness, correctness, errors, changes, product metrics,reliability, etc.) from the point of view of the (developer, manager, customer, corporate perspective, etc) E.g. Examine the effectiveness from the developer's point of view.

Environment: The environment consists of the following: process factors, people factors, problem factors, methods, tools, constraints, etc. E.g. The product is an operating system that must fit on a PC, etc.

Process Questions:

For each process under study, there are several subgoals that need to be addressed. These include the quality of use (characterize the process quantitatively and assess how well the process is performed), the domain of use (characterize the object of the process and evaluate the knowledge of object by the performers of the process), effort of use (characterize the effort to perform each of the subactivities of the activity being performed), effect of use (characterize the output of the process and the evaluate the quality of that output), and feedback from use (characterize the major problems with the application of the process so that it can be improved).

Other subgoals involve the interaction of this process with the other processes and the schedule (from the viewpoint of validation of the process model).

Product Questions:

For each product under study there are several subgoals that need to be addressed. These include the definition of the product (characterize the product quantitatively) and the evaluation of the product with respect to a particular quality (e.g. reliability, user satisfaction)

The definition of the product consists of:

1. Physical Attributes. e.g. size (source lines, #units, executable lines), complexity (control and data), programming language features, time space.

2. Cost. e.g. effort (time, phase, activity, program)

3. Changes. e.g. errors, faults, failures and modifications by various classes.

4. Context. e.g. customer community, operational profile.

The evaluation is relative to a particular quality e.g. reliability. Thus the physical characteristics need to be analyzed relative to these. Template questions for evaluation include:

How do you measure the quality?
Is the model used valid?
Are the measures used valid?
Are there checks?
Do they agree with the reliability data?

Thus a sample would be:

To evaluate the product (system) in order to assess its quality. Examine the reliability relative to the customer's point of view.

INVESTIGATION LAYOUT

The original goal/question/metric paradigm has been refined with experience [Basili & Selby 1984] to include a step which provides for help in planning the type of investigative analysis possible based upon the scope of the evaluation and the type of data available. Between steps 3 an 4 above is a step to plan the investigation layout and analysis methods. This step is important because it allows the questions to reflect the types of result statements that can be used in the quantitative analysis.

With all the different methods and tools available, we need to better quantitatively understand and evaluate the benefits and drawbacks of each of them. There are several different approaches to quantitatively evaluating methods and tools: blocked subject-project, replicated project, multi-project variation, and single project case study. The approaches can be characterized by the number of teams replicating each project and number of different projects analyzed as shown in Table 3.

		# of projects	
		one	more than one
# of teams per project	one	single project	multi-project variation
	more than one	replicated project	blocked subject-project

Table 3

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

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

Multi-project variation analysis involves the measurement of several projects where controlled factors such as methodology can be varied across similar projects. This is not a controlled experiment as the previous two approaches were, but allows the experimenter to study the effect of various methods and tools to the extent that the organization allows them to vary on different projects.

The case study is where most methodology evaluation begins. There is a project and the management has decided to make use of some new method or set of methods and wants to know whether or not the method generates any improvement in the productivity or quality. A great deal depends upon the individual factors involved in the project and the methods applied.

The approaches vary in cost and the level of confidence one can have in the result of the study. Clearly, an analysis of several replicated projects costs more money but will generate stronger confidence in the conclusion. Unfortunately, since a blocked subject-project experiment is so expensive, the projects studied tend to be small. The size of the projects increase as the costs go down so it is possible to study very large single project experiments and even multi- project variation experiments if the right environment can be found.

The SEL has had some experience in almost all of theses categories. A blocked subject-project study was the comparison of functional testing, structural testing and code reading [Basili & Selby 1985]. Here programs of 145 to 365 lines of code were analyzed by programmers using each of the techniques on different types of applications, e.g. a text formatter, a plotter, an abstract data type , and a database. The goal was to compare the techniques with respect to fault detection effectiveness, fault detection cost, and classes of faults detected. We were also able to compare performance with respect to the software type and the level of expertise of the programmer.

Due to cost, we have only used the replicated project analysis to a limited degree. Here comparisons have been of only two projects, e.g. comparing the development of a dynamic simulator in the standard FORTRAN and Ada [Agresti 1985]. The limitation of only two replicated developments makes the analysis more like a pair of cases studies than a true replicated project analysis. However replicated-project analysis has been used at the University of Maryland to study similar issues to the SEL on a smaller scale, e.g. the effect of a set of software development methods on the process and product [Basili & Reiter 1981], [Basili & Hutchens 1983].

A large number of projects have fit into the multi-project variation category. Various subsets of the 41 projects have been analyzed for a variety of purposes. Studies have been performed to develop and evaluate cost models [Basili & Zelkowitz 1978], [Basili & Beane 1981], [Basili & Freburger 1981], [Bailey & Basili 1981], evaluate the relationships of product and process variables [Basili, Selby & Phillips 1983], [Basili & Selby 1985a], [Basili & Panlilio-Yap 1985] , measure productivity [Basili & Bailey 1980], characterize changes and errors [Weiss & Basili 1984], predict problems based upon previous projects [Doerflinger & Basili 1985], and evaluate methodology [Bailey & Basili 1981], [Card, Church & Agresti 1986].

Many projects have been studied in isolation as cases studies, to analyze the effects of changes and errors [Basili & Perricone 1984], to measure the testing approach [Ramsey & Basili 1985], to study the modular structure of programs [Hutchens & Basili 1985].

METHODOLOGY IMPROVEMENT PARADIGM

All this leads us to the following basic paradigm for evaluating and improving the methodology used in the software development and maintenance process [Basili 1985].

1. Characterize the approach/environment.

This step requires an understanding of the various factors that will influence the project development. This includes the problem factors, e.g. the type of problem, the newness to the state of the art, the susceptibility to change, the people factors, e.g. the number of people working on the project, their level of expertise, experience, the product factors, e.g. the size, the deliverables, the reliability requirements, portability requirements, reusability requirements, the resource factors, e.g. target and development machine systems, availability, budget, deadlines, the process and tool factors, e.g. what techniques and tools are available, training in them, programming languages, code analyzers.

2. Set up the goals, questions, data for successful project development and improvement over previous project developments.

It is at this point the organization and the project manager must determine what the goals are for the project development. Some of these may be specified from step 1. Others may be chosen based upon the needs of the organization, e.g. reusability of the code on another project, improvement of the quality, lower cost.

3. Choose the appropriate methods and tools for the project.

Once it is clear what is required and available, methods and tools should be chosen and refined that will maximize the chances of satisfying the goals laid out for the project. Tools may be chosen because they facilitate the collection of the data necessary for evaluation, e.g. configuration management tools not only help project control but also help with the collection and validation of error and change data.

4. Perform the software development and maintenance, collecting the prescribed data and validating it.

This step involves the collection of data by forms, interviews, and automated collection mechanisms. The advantages of using forms to collect data is that a full set of data can be gathered which gives detailed insights and provides for good record keeping. The drawback to forms is that they can be expensive and unreliable because people fill them out. Interview can be used to validate information from forms and gather information that is not easily obtainable in a form format. Automated data collection is reliable and unobtrusive and can be gathered from

program development libraries, program analyzers, etc. However, the type of data that can be collected in this way is typically not very insightful and one level removed from the issue being studied.

5. Analyze the data to evaluate the current practices, determine problems, record the findings and make recommendations for improvement.

This is the key to the mechanism. It requires a post mortem on the project. Project data should be analyzed to determine how well the project satisfied its goals, where the methods were effective, where they were not effective, whether they should be modified and refined for better application, whether more training or different training is needed, whether tools or standards are needed to help in the application of the methods, or whether the methods or tools should be discarded and new methods or tools applied on the next project.

6. Proceed to step 1 to start the next project, armed with the knowledge gained from this and the previous projects.

This procedure for developing software has a corporate learning curve built in. The knowledge is not hidden in the intuition of first level managers but is stored in a corporate data base available to new and old managers to help with project management, method and tool evaluation, and technology transfer.

SEL EXPERIENCE

There are several areas where we believe we have been successful in the measurement area. We have been able to collect reasonably accurate effort data especially with regard to weekly effort hours. The attribution of that effort data to various phases and activities has also been reasonably successful.

We have been successful in extracting realistic histories of the errors and changes on a project but have not been so successful in capturing detailed data on the effectiveness of the various error detection techniques. The latter problem is due to the ad hoc way programmers tend to apply techniques, not always recording all their efforts and to the common use of combinations of techniques. We have been successful in capturing product characteristics but problem characteristics are more difficult to capture. This is largely because they are difficult to quantify and differentiate. We have been able to measure the relative level of the total set of methods used in a project but less effective in isolating the effects of specific methods. This is because most of the studies have been of the multi-project or case study type analysis and it has been difficult to delineate the effects of a specific technique. One successful isolation of techniques was the blocked subject-project study of testing techniques vs. reading.

With regard to the cost of the measurement program in the SEL, the data collection overhead to tasks has been about 3% of total project cost and the processing of the data has been about 5%. It is actually the analysis, interpretation and reporting of the results that have been the most expensive in the SEL. This has been in the order of 15% to 20% but includes all the research support, paper publication, report generation and technology transfer activities.

We have studied the question of what measurement can be automated, i.e. what tools can be used to relieve the impact of measurement on the development or management team. We have automated such things as computer utilization, code and changes growth, product complexity, product characteristics (e.g. size) and source code change count. We have tried to automate but failed with regard to error reporting, weekly resources, and effort by activity. Part of the lack of success has been due to the variation in the development environments, i.e. the use of different mainframes for development, the lack of consistent interactive development across projects. We have not even tried to automate information about the techniques used, resources by component, the environment, changes to the design and specifications, and problem complexity.

We have standardized on various measures of quality in the SEL. Productivity is defined as developed source lines of code (SLOC) per day. Reliability is the number of errors after unit test per 1000 SLOC. Maintainability is the average reported effort to modify or correct the software. Reusability is the percent of components reused on new projects.

RECOMMENDATIONS AND CONCLUSIONS

From our experience within the SEL we would argue that software technology can and should be measured. The measurement overhead to projects should be about 3%. You should not spend excessive effort in trying to automate the data collection process. You should not collect and store data that is not goal driven, i.e. you should collect the minimal set of data needed for the purpose. You should measure top level information for all projects and detailed data for specific experiments. It is difficult to measure the effects of specific techniques in a production environment.

It is best to use the data to characterize the environment, making the problems visible. You should set up both corporate and project goals and use the goal/question/metric paradigm to articulate the process and product needs.

REFERENCES

[Agresti 1985]
William Agresti and the SEL Staff, Measuring Ada as a Software Development Technology in the SEL, Eighth Minnowbrook Workshop on Software Performance Evaluation, Blue Mountain Lake, New York, July 30, 1985.

[Bailey & Basili 1981]
John W. Bailey and Victor R. Basili, A Meta-Model for Software Development Resource Expenditures, Proceedings of the Fifth International Conference on Software Engineering, San Diego, California, pp 107-116, 1981.

[Basili 1985]
Victor R. Basili, Quantitative Evaluation of Software Methodology, Proceedings of the First Pan Pacific Computer Conference, 1985.

[Basili & Bailey 1980]
Victor R. Basili and John W. Bailey, The Software Engineering Laboratory: Measuring the Effects of Software Methodologies within the Software Engineering Laboratory, Proceedings of the Fifth Annual Software Engineering Workshop, November 1980.

[Basili & Beane 1981]
Victor R. Basili and John Beane, Can the Parr Curve Help with Manpower Distribution and Resource Estimation Problems?, Journal of Systems and Software, pp 59-69, Volume 2, 1981.

[Basili & Freburger 1981]
Victor R. Basili and Karl Freburger, Programming Measurement and Estimation in the Software Engineering Laboratory, Journal of Systems and Software, pp 47-57, Volume 2, 1978.

[Basili & Hutchens 1983]
Victor R. Basili & David H. Hutchens, An Empirical Study of a Syntactic Complexity Family, IEEE Transactions on Software Engineering, pp 664-672, November 1983.

[Basili & Panlilio-Yap 1985]
Victor R. Basili and N. Monina Panlilio-Yap, Finding Relationships between Effort and other Variables in the SEL, 9th COMPSAC Computer and Software Applications Conference, pp 221-228, October, 1985.

[Basili & Perricone 1984]
Victor R. Basili and Barry T. Perricone, Software Errors and Complexity: An Empirical Investigation, Communications of the ACM, pp 42-52, January, 1984.

[Basili & Reiter 1981]
Victor R. Basili and Robert W. Reiter, Jr., A Controlled Experiment Quantitatively Comparing Software Development Approaches, IEEE Transactions on Software Engineering, Vol. SE-7, No. 3, pp 299-320, May 1981.

[Basili & Selby 1984]
Victor R. Basili and Richard W. Selby, Jr., Data Collection and Analysis in Software Research and Management, Proceedings of the American Statistical Association, pp 21-30, 1984.

[Basili & Selby 1985]
Victor R. Basili and Richard W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, University of Maryland Technical Report TR-1501, May 1985.

[Basili & Selby 1985a]
Victor R. Basili and Richard W. Selby Jr., Calculation and Use of an Environment's Characteristic Software Metric Set, IEEE Proceedings 8th International Conference on Software Engineering, pp 386-391, August 1985.

[Basili, Selby & Phillips 1983]
Victor. R. Basili, Richard W. Selby, Tsai-Yun Phillips, Metric Analysis and Data Validation Across FORTRAN Projects, IEEE Transactions on Software Engineering, pp 652-663, November, 1983.

[Basili & Weiss 1984]
Victor R. Basili and David M. Weiss, A Methodology for Collecting Valid Software Engineering Data, IEEE Transactions on Software Engineering, Vol. SE-10, No. 3, pp 728-738, November 1984.

[Basili & Zelkowitz 1978]
Victor R. Basili and Marvin V. Zelkowitz, Analyzing Medium Scale Software Development, IEEE 3rd International Conference on Software Engineering, pp 116-123, May 1978.

[Card, Church & Agresti 1986]
D.N. Card, V. E. Church, and W. W. Agresti, An Empirical Study of Software Design Practices, IEEE Transactions on Software Engineering, pp 264-271, February 1986.

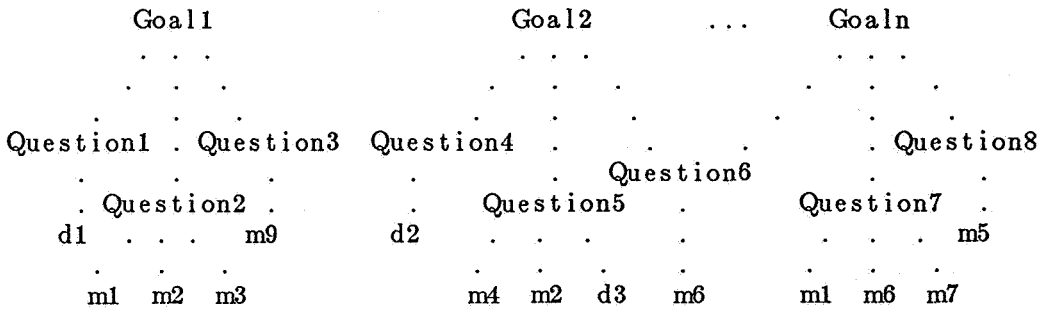
[Doerflinger & Basili]
Carl W. Doerflinger and Victor R. Basili, Monitoring Software Development Through Dynamic Variables, IEEE Transaction on Software Engineering, pp 978-985, September 1985.

[Hutchens & Basili 1985]
David H. Hutchens and Victor R. Basili, System Structure Analysis: Clustering with Data Bindings, IEEE Transactions on Software Engineering, pp 749-757, August, 1985.

[Ramsey & Basili 1985]
James Ramsey and Victor R. Basili, Analyzing the Test Process Using Structural Coverage, 8th Internation Conference on Software Engineering, pp 306-311, August, 1985.

[Weiss & Basili 1985]
Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory, IEEE Transactions on Software Engineering, pp 157-168, February 1985.

Here is the goal, question, metric hierarchy:



Here there are n goals shown and each goal generates a set of questions that attempt to define and quantify the specific goal which is at the root of its goal tree. The goal is only as well defined as the questions that it generates. Each question generates a set of metrics (mi) or distributions of data (di). Again, the question can only be answered relative to and as completely as the available metrics and distributions allow. As is shown in the above diagram, the same questions can be used to define different goals (e.g. Question6) and metrics and distributions can be used to answer more that one question. Thus questions and metrics are used in several contexts.

Given the above paradigm, the data collection process consists of six steps:

Visibility	Quality	Technology
Resource Data	Productivity	How much do certain techniques help?
Error Data	Reliability	
Environment Characteristics	Maintainability	
	Portability	Which tools improve productivity?
Problem Complexity	Reusability	
Product Data		

Table 1

How do you measure the quality?
Is the model used valid?
Are the measures used valid?
Are there checks?
Do they agree with the reliability data?

	# of projects	
	one	more than one
# of teams: one	single project	multi-project variation

per project	more than one	replicated project	blocked subject-project

Table 3

THE VIEWGRAPH MATERIALS

FOR THE

VIC BASILI PRESENTATION FOLLOW

MEASURING THE SOFTWARE PROCESS AND PRODUCT: LESSONS LEARNED IN THE SEL

VICTOR R. BASILI
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

WHY MEASURE SOFTWARE?

CREATE A CORPORATE MEMORY (SUPPORT PLANNING)

E.G., HOW MUCH WILL A NEW PROJECT COST?

DETERMINE STRENGTHS AND WEAKNESSES OF THE CURRENT PROCESS AND PRODUCT

E.G., ARE CERTAIN TYPES OF ERRORS COMMONPLACE?

DEVELOP A RATIONALE FOR ADOPTING/REFINING TECHNIQUES

E.G., WHAT TECHNIQUES WILL MINIMIZE CURRENT PROBLEMS?

ASSESS THE IMPACT OF TECHNIQUES

E.G., DOES FUNCTIONAL TESTING MINIMIZE CERTAIN ERROR CLASSES?

EVALUATE THE QUALITY OF THE PROCESS/PRODUCT

E.G., WHAT IS THE RELIABILITY OF THE PRODUCT AFTER DELIVERY?

3 ASPECTS OF MEASURES IN THE SEL

VISIBILITY	QUALITY	TECHNOLOGY
RESOURCE DATA	PRODUCTIVITY	HOW MUCH DO CERTAIN TECHNIQUES
ERROR DATA	RELIABILITY	HELP?
ENVIRONMENT CHARACTERISTICS	MAINTAINABILITY	
PROBLEM COMPLEXITY	PORTABILITY	WHICH TOOLS IMPROVE PRODUCTIVITY?
PRODUCT DATA	REUSABILITY	

CLASSES OF PROJECT DATA

RESOURCE DATA

- WEEKLY HRS. BY PERSON
- HOURS BY ACTIVITY
- HOURS BY COMPONENT
- COMPUTER UTILIZATION
-
-
-

ERROR/CHANGE DATA

- SOFTWARE FAILURES
  - BY TIME
  - TYPE
  - MAGNITUDE
- SOFTWARE CHANGES
- DESIGN CHANGES
- SPECIFICATION CHANGES

PROJECT CHARACTERISTICS

- PHASE DATES
- SIZE (LINES OF CODE, COMPONENTS)
- SIZE BY TIME
- ENVIRONMENT
- PROBLEM COMPLEXITY
- PRODUCT COMPLEXITY
-
-
-

'TECHNIQUES'

- TOOLS USED
- TECHNIQUES APPLIED
- ORGANIZATION STRUCTURE(IV&V, CPT, ..)
- TRAINING PROVIDED
-
-

PROJECTS STUDIED IN SEL

1977-1984

NUMBER OF PROJECTS	41
TOTAL PROJECTS SIZE	1.3 M SLOC
DEVELOPMENT COST	$11M (84 DOLLARS)
DATA 'FORMS' COLLECTED	30,000

MAJOR LESSON LEARNED

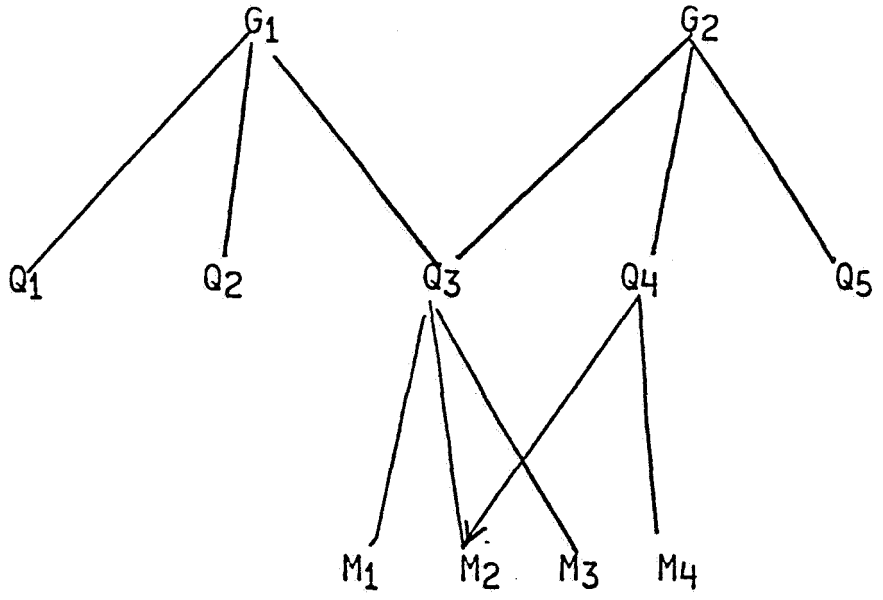
DEVELOP A GOAL-DRIVEN PARADIGM FOR DATA COLLECTION

EVIDENCE

HAVE THROWN AWAY DATA GATHERED BOTTOM UP, E.G. RUN ANALYSIS

HAVE NOT HAD SUCCESS IN AREAS WHERE THERE WAS NOT APPROPRIATE FOCUS,E.G., EFFECTIVENESS OF DETAILED TECHNIQUES

GOAL/QUESTION/METRIC PARADIGM



MANAGEMENT-ORIENTED GOAL
(CHARACTERIZE ERRORS)

SPECIFIC QUESTION
OR HYPOTHESIS
(WHAT PHASE WAS GREATEST
SOURCE OF ERRORS?)

QUANTITATIVE METRIC
OR DISTRIBUTION
(ERROR DISTRIBUTION BY PHASE)

	Q_1	Q_2	Q_3	...	Q_M
G_1	M_1, M_2		M_1, M_2		
			M_3		
G_2				M_2, M_1	
G_3					
⋮					⋮
G_N				..	

SEL

DATA COLLECTION METHODOLOGY

1. ESTABLISH THE GOALS OF DATA COLLECTION; E.G., CHARACTERIZE CHANGES DURING SOFTWARE DEVELOPMENT.

2. DEVELOP A LIST OF QUESTIONS OF INTEREST; E.G., WHAT PERCENTAGE OF THE CHANGES WERE MODIFICATIONS AND ERRORS?

3. DETERMINE THE METRICS AND DISTRIBUTIONS NEEDED TO ANSWER THE QUESTIONS.

4. DESIGN AND TEST DATA COLLECTION FORM.

5. COLLECT AND VALIDATE DATA.

6. ANALYZE AND INTERPRET THE DATA

SAMPLE GOALS

ON TIME DELIVERY

HIGH QUALITY PRODUCT

HIGH QUALITY PROCESS

CONTAINS NEEDED FUNCTIONALITY

SALABLE PRODUCT

CUSTOMER SATISFACTION

CHARACTERIZE ERRORS AND CHANGES TO LEARN
FROM THIS PROJECT

LOW COST

TIMELINESS

. . .

CHARACTERIZING GOALS

1. CHARACTERIZE RESOURCE USAGE ACROSS THE PROJECT
2. CHARACTERIZE CHANGES AND ERRORS ACROSS LIFE CYCLE
3. CHARACTERIZE THE DIMENSIONS OF THE PROJECT
4. CHARACTERIZE THE EXECUTION TIME ASPECTS
5. CHARACTERIZE THE ENVIRONMENT

- QUALITY GOALS
- PRODUCTIVITY GOALS
- MAINTENANCE GOALS
- TOOL AND METHOD EVALUATION GOALS
- COST-ESTIMATION GOALS

  ETC.

Quantitative Analysis Methodology

- Methodology for data collection & quantitative analysis

  1. **Formulate** goals
  2. **Develop** and refine subgoals & questions
  3. **Establish** appropriate metrics
  4. **Plan** investigation layout & analysis methods
  5. **Design** & test data collection scheme
  6. **Perform** investigation concurrently w/ data validation
  7. **Analyze** data

- Goal/question/metric paradigm defines analysis purpose, required data, and context for interpretation

- Questions are coupled with measurable attributes and reflect the types of result statements from quantitative analysis

- Identifies aspects of a well-run analysis

- Intended to be applied to different types of studies from a variety of problem domains

Analysis Classification: Scopes of Evaluation

#Teams per Project	#Projects	
	One	More Than One
One	Single Project	Multi-Project Variation
More Than One	Replicated Project	Blocked Subject-Project

GOAL SETTING TEMPLATE

PURPOSE OF STUDY:

TO (CHARACTERIZE, EVALUATE, PREDICT, MOTIVATE) THE (PROCESS, PRODUCT, METRIC) IN ORDER TO (UNDERSTAND, ASSESS, MANAGE, ENGINEER, LEARN, IMPROVE, COMPARE) IT

E.G., TO EVALUATE THE SYSTEM TEST METHODOLOGY IN ORDER TO ASSESS IT.

PERSPECTIVE:

EXAMINE THE (COST, EFFECTIVENESS, RELIABILITY, CORRECTNESS, MAINTAINABILITY, EFFICIENCY, ETC.) FROM THE POINT OF VIEW OF THE (DEVELOPER, MANAGER, CUSTOMER, CORPORATION, ETC.)

E.G., EXAMINE THE EFFECTIVENESS FROM THE DEVELOPER'S POINT OF VIEW.

ENVIRONMENT:

LIST THE VARIOUS PROCESS FACTORS, PROBLEM FACTORS, PEOPLE FACTORS, ETC.

HIERARCHY OF PERSPECTIVES

DOMAIN	CONCERNS
1) INDUSTRY-WIDE	- TECHNOLOGICAL CAPABILITY, INTERNATIONAL COMPETITION
2) CORPORATE	- PROFIT, MARKET POSITION
3) UNIT MANAGEMENT	- RESOURCE ALLOCATION
4) PROJECT MANAGEMENT	- PROGRESS AGAINST MILESTONES
5) PROJECT TEAM	- INTEGRATION OF INDIVIDUAL PRODUCTS
6) INDIVIDUAL	- PRODUCT QUALITY, WORK RATE

GOAL AREA: PROCESS QUALITY

PURPOSE:

PERSPECTIVE:

ENVIRONMENT:

DEFINITION OF THE PROCESS:

- QUALITY OF USE
- DOMAIN OF USE
  - KNOWLEDGE OF DOMAIN
  - VOLATILITY OF DOMAIN
- COST OF USE
- EFFECTIVENESS OF USE
  - RESULTS
  - QUALITY OF RESULTS

FEEDBACK FROM USE

- LESSONS LEARNED
- MODEL VALIDATION
- INTEGRABILITY WITH OTHER TECHNIQUES

EXAMPLE

PURPOSE OF STUDY: TO EVALUATE THE SYSTEM TEST METHODOLOGY IN ORDER TO ASSESS IT'S EFFECT

PERSPECTIVE: EXAMINE THE EFFECTIVENESS FROM THE DEVELOPER'S POINT OF VIEW

DEFINITION OF PROCESS:

1. QUALITY OF USE
   - 1.1 HOW MANY REQUIREMENTS ARE THERE?
   - 1.2 WHAT IS THE DISTRIBUTION OF TESTS OVER REQUIREMENTS?
     - NUMBER OF TESTS/REQUIREMENT
   - 1.3 WHAT IS THE IMPORTANCE OF TESTING EACH REQUIREMENT?
     - RATE 0-5
   - 1.4 WHAT IS THE COMPLEXITY OF TESTING EACH REQUIREMENT?
     - RATE 0-5
     - SUBJECTIVE
     - FANOUT TO COMPONENTS AND/OR NAMES
   - 1.5 IS Q1.2 CONSISTENT WITH Q1.3 AND Q1.4?

EXAMPLE (CONT'D)

2. DOMAIN OF USE

KNOWLEDGE:

2.1 HOW PRECISELY WERE THE TEST CASES KNOWN IN ADVANCE?

RATE 0-5

2.2 HOW CONFIDENT ARE YOU THAT THE RESULT IS CORRECT?

VOLATILITY:

2.3 ARE TESTS WRITTEN/CHANGED CONSISTENT WITH Q1.3 AND Q1.4?

2.4 WHAT PERCENT OF THE TESTS WERE RERUN?

3. COST OF USE

3.1 COST TO MAKE A TEST

3.2 COST TO RUN A TEST

3.3 COST TO CHECK A RESULT

3.4 COST TO ISOLATE THE FAULT

3.5 COST TO DESIGN AND IMPLEMENT A FIX

3.6 COST TO RETEST

EXAMPLE (CONT'D)

4. EFFECTIVENESS OF USE

QUALITY OF RESULTS

4.1 HOW MANY FAILURES WERE OBSERVED?

4.2 WHAT PERCENT OF TOTAL ERRORS WERE FOUND?

4.3 WHAT PERCENT OF THE DEVELOPED CODE WAS EXERCISED?

4.4 WHAT IS THE STRUCTURAL COVERAGE OF THE ACCEPTANCE TESTS?

RESULTS:

4.5 HOW MANY ERRORS WERE DISCOVERED DURING EACH PHASE OF DEVELOPMENT ANALYZED BY CLASS OF ERROR AND IN TOTAL?

4.6 WHAT IS THE NUMBER OF FAULTS PER LINE OF CODE AT THE END OF EACH PHASE? ONE MONTH, SIX MONTHS, ONE YEAR?

4.7 WHAT IS THE COST TO FIX AN ERROR ON THE AVERAGE AND FOR EACH CLASS OF ERROR AT EACH PHASE?

4.8 WHAT IS THE COST TO ISOLATE AN ERROR ON THE AVERAGE AND FOR EACH CLASS OF ERROR AT EACH PHASE?

GOAL AREA: HIGH QUALITY PRODUCT

PRODUCT:

PURPOSE OF STUDY:

ENVIRONMENT:

DEFINITION OF PRODUCT:

- PHYSICAL ATTRIBUTES
- COST
- CHANGES AND ERRORS
- CONTEXT
  - CUSTOMER COMMUNITY
  - OPERATIONAL PROFILES

PERSPECTIVE:

- MAJOR MODEL(S) USED:
- VALIDITY OF THE MODEL FOR THE PROJECT
- VALIDITY OF THE DATA COLLECTED
- MODEL EFFECTIVENESS
- SUBSTANTIATION OF THE MODEL

IMPROVING METHODOLOGY, PRODUCTIVITY AND QUALITY
THROUGH PRACTICAL MEASUREMENT

1. CHARACTERIZE THE ENVIRONMENT

2. SET UP THE GOALS FOR IMPROVEMENT
   E.G., HIGHER QUALITY, LOWER COST, ON-TIME DELIVERY

3. REFINE AND ADJUST APPROACH/ENVIRONMENT TO
   SATISFY THE GOALS

4. BUILD THE SYSTEM, COLLECT AND VALIDATE THE DATA

5. INTERPRET AND ANALYZE THE DATA TO CHECK IF THE
   GOALS ARE SATISFIED
   EVALUATE METHODOLOGY, PRODUCTIVITY AND QUALITY, ETC.

6. GO TO STEP 1, ARMED WITH NEW KNOWLEDGE

SEL SUCCESSES/FAILURES

EFFORT DATA

- WEEKLY EFFORT HOURS CAN BE ACCURATELY CAPTURED
- EFFORT BY PHASE AND ACTIVITY CAN BE IMPROVED

ERROR/CHANGE DATA

- CAN EXTRACT REALISTIC HISTORY OF ERRORS AND CHANGES
- CANNOT CAPTURE DETAILED TECHNIQUE INFORMATION (FOR ERROR DETECTION)

PROJECT CHARACTERISTICS

- PRODUCT CHARACTERISTICS CAN BE ACCURATELY CAPTURED
- PROBLEM CHARACTERISTICS DIFFICULT TO CAPTURE

TECHNIQUES

- CAN MEASURE RELATIVE LEVEL OF TOTAL METHODOLOGY
- DIFFICULT TO ISOLATE EFFECTS OF SPECIFIC METHODS

COST OF DATA COLLECTION

- OVERHEAD TO TASKS DOES NOT HAVE TO EXCEED 3%

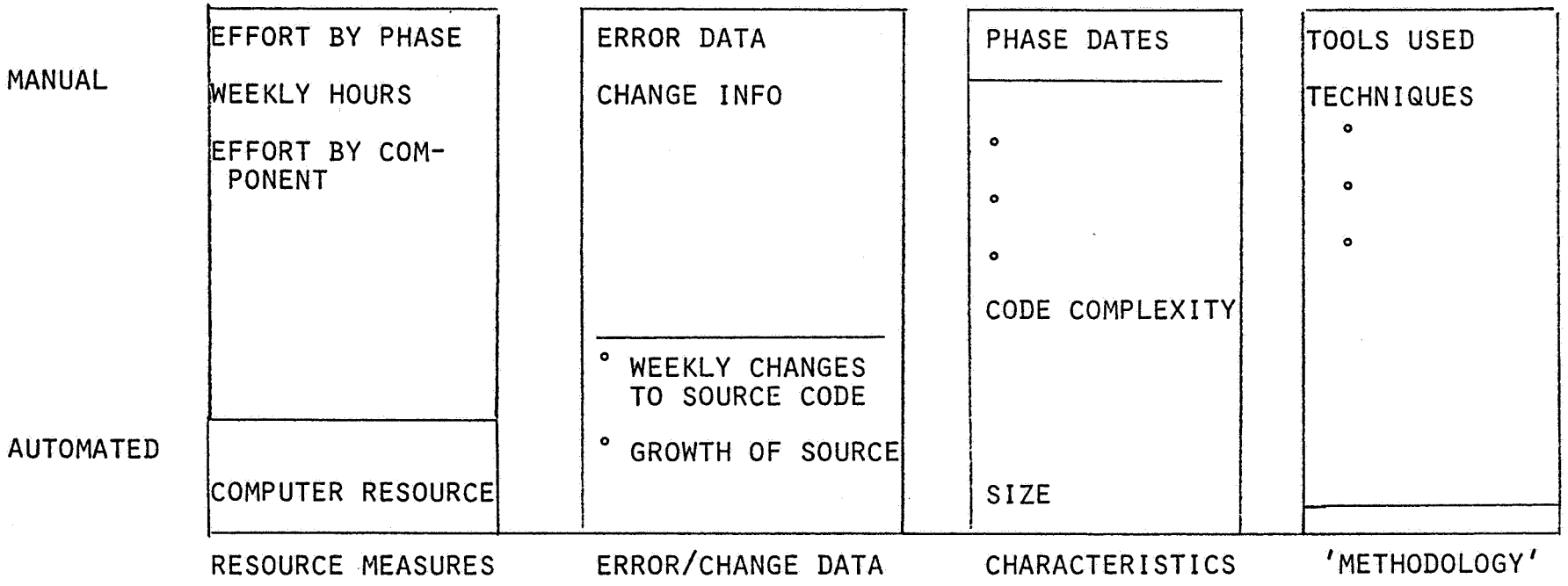
- PROCESSING OF DATA CAN BE CUT TO 5%

- ANALYSIS, INTERPRETATION AND REPORTING

  MOST EXPENSIVE

  - 15 - 20% IN SEL
  - INCLUDES RESEARCH SUPPORT
    PAPER PUBLICATION
    TECHNOLOGY TRANSFER

CAN WE AUTOMATE* MEASUREMENT?



AUTOMATED

- COMPUTER UTILIZATION
- CODE CHANGES/GROWTH
- PRODUCT COMPLEXITY
- PRODUCT CHARACTERISTICS (SIZE...)
- SOURCE CODE CHANGE COUNT

NOT AUTOMATED

TRIED/BUT FAILED	ALWAYS MANUAL
ERROR REPORTING	TECHNIQUES USED
WEEKLY RESOURCES	RESOURCES BY COMPONENT
EFFORT BY ACTIVITY	ENVIRONMENT
	CHANGES TO DESIGN/SPECS
	PROBLEM COMPLEXITY

* IMPLIES TOOL OR PROCEDURE TO RELIEVE ANY IMPACT TO DEVELOPMENT TEAM OR MANAGEMENT

MEASURES OF SOFTWARE 'QUALITY' IN SEL

° PRODUCTIVITY		DEVELOPED SLOC/PERSON DAY
° RELIABILITY		NUMBER OF ERRORS (AFTER UNIT TEST)1000 SLOC
° MAINTAINABILITY	EFFORT TO CHANGE EFFORT TO REPAIR	AVERAGE REPORTED EFFORT TO MODIFY (CORRECT)SOFTWARE
° REUSABILITY		% OF COMPONENTS REUSED ON NEW PROJECTS

SPECIFIC SEL EXPERIENCE

° SOFTWARE (TECHNOLOGY) CAN AND SHOULD BE MEASURED

° MEASUREMENT OVERHEAD TO DEVELOPMENT PROJECTS ABOUT 3+%

° DON'T SPEND EXCESSIVE EFFORT IN TRYING TO AUTOMATE COLLECTION

° DO NOT COLLECT/STORE 'DATA' THAT IS NOT GOAL-DRIVEN
(COLLECT MINIMUM SET OF DATA)

° DIFFICULT TO MEASURE SPECIFIC SOFTWARE TECHNIQUES

° MEASURE TOP LEVEL INFORMATION FOR ALL PROJECTS - DETAILED DATA FOR SPECIFIC EXPERIMENTS (E.G., TESTING TECHNIQUES)

OVERALL RECOMMENDATION

USE DATA TO CHARACTERIZE THE ENVIRONMENT, MAKING PROBLEMS VISIBLE

SET UP CORPORATE AND PROJECT GOALS AND USE GOAL/QUESTION/DATA PARADIGM TO ARTICULATE PROCESS AND PRODUCT NEEDS

N86-30359

STUDIES AND EXPERIMENTS IN THE* SOFTWARE ENGINEERING LAB (SEL)

BY
FRANK E. MCGARRY
NASA/GSFC
AND
DAVID N. CARD
COMPUTER SCIENCES CORPORATION (CSC)

ABSTRACT

The Software Engineering Laboratory (SEL) is an organization created nearly 10 years ago for the purpose of identifying, measuring and applying quality software engineering techniques in a production environment (Reference 1). The members of the SEL include NASA/GSFC (the sponsor and organizer), University of Maryland, and Computer Sciences Corporation. Since its inception the SEL has conducted numerous experiments, and has evaluated a wide range of software technologies. This paper describes several of the more recent experiments as well as some of the general conclusions to which the SEL has arrived.

1.0 Background (Chart 1)

Over the past 9 years, the SEL has conducted studies in 4 major areas of software technology:

1. Software Tools and Environments
2. Development Methods
3. Measures and Profiles
4. Software Models

Most of these studies have been conducted by utilizing specific approaches, tools or models to production software problems within the flight dynamics environment at Goddard. By extracting detailed information pertaining to the problem, environment, process and product, the SEL has been able to gain some insight into the relative impact that the various technologies may have on the quality of the software being developed.

More detailed descriptions of the overall measurement process as well as the SEL studies may be found in References 1, 2, and 3. This brief paper will describe some of the more recent, specific experiments that have been conducted by/in the SEL and just what types of insight may be provided in areas of:

1. Tools and Environments
2. Software Testing
3. Design Measures
4. General Trends

*The work described in this paper has been extracted from reports and studies carried out by members of the SEL.

TYPE OF SOFTWARE: SCIENTIFIC, GROUND-BASED, INTERACTIVE GRAPHIC, MODERATE RELIABILITY AND RESPONSE REQUIREMENTS

LANGUAGES: 85% FORTRAN, 15% ASSEMBLER MACROS

COMPUTERS: IBM MAINFRAMES, BATCH WITH TSO

PROJECT CHARACTERISTICS:	AVERAGE	HIGH	LOW
DURATION (MONTHS)	16	21	13
EFFORT (STAFF-YEARS)	8	24	2
SIZE (1000 LOC)			
DEVELOPED	57	142	22
DELIVERED	62	159	33
STAFF (FULL-TIME EQUIVALENT)			
AVERAGE	5	11	2
PEAK	10	24	4
INDIVUALS	14	29	7
APPLICATION EXPERIENCE (YEARS)			
MANAGERS	6	7	5
TECHNICAL STAFF	4	5	3
OVERALL EXPERIENCE (YEARS)			
MANAGERS	10	14	8
TECHNICAL STAFF	9	11	7

FIGURE 1. FLIGHT DYNAMICS SOFTWARE

The Flight Dynamics environment typically is a FORTRAN environment building software systems ranging in size from 10,000 to 150,000 lines of code - (see Figure 1).

2.0 Software Tools/Environments* (Chart 2 and Reference 4)

One of the more interesting studies that was conducted within the past several years, was one in which an attempt was made to measure the impact of several development approaches (related to environment support) on the quality of software within the flight dynamics discipline.

The three points of study include:

1. Software Tools
2. Computer Support
3. Number of Terminals/Programmer

The quality of the product was measured using 4 attributes including:

1. Productivity - Number of developed lines of code per man month.

2. Reliability - Number of errors reported per 1,000 lines of code.

3. Effort to Change - (Average number of man hours required to make a software modification).

4. Effort to Repair (Average number of man hours required to correct an identified error)

2.1 Experiment Description (Chart 3)

In carrying out the study, a review of all projects for which detailed project history data was available and complete was undertaken. From the completed 50 projects, 14 were selected because of the quality and completeness of the relevant data and more importantly because of the general similarity of complexity of problems that the software was attempting to solve.

Fourteen projects ranging in size from 11,000 lines of code to 136,000 lines of code were selected. These projects had information describing the environment under which they were developed and additional information such as the number and quality of automated tools utilized and the number of interactive terminals available to the programming staff.

*Lead investigators of this work included F. McGarry and J. Valett of NASA/GSFC and D. Hall of NASA/HQ.

The 14 projects selected all dealt with tasks in solving attitude determination and control related problems. The projects were completed between the years 1978 to 1984.

The projects also had detailed information as to manhours, size, error history, and effort required to make all changes and corrections to the software.

2.2 Project Variations (Chart 4)

In attempting to characterize each of the development projects, a ranking scheme was used for this particular study. It was found that the availability of terminals ranged from a low of less than 1 per 8 programmers to a high of better than 1 per 2 programmers.

There were a total of 21 tools considered in this study that were applied by at least some of the projects studied. Such tools as documentation aids, preprocessors, test generators and program optimizers were among the tools considered.

It was also found that the distribution of level of use for tools ranged from a low of only 1 or 2 automated tools being used, to a high of more than 8 automated tools being used. These tools also were rated as far as the actual usage by the particular project and also there was a rating for each tool of the assessed 'quality' of the particular tool. Quality here was rated for each tool on a scale of 1 to 5 and was a subjective rating determined by the software manager.

There were a total of 11 characteristics that made up the computer support measure. These 11 included:

- Terminal Accessibility
- Turn around time
- Compiler Speed
- System Reliability (2 measures)
- Direct Storage
- Offline Storage
- Interactive Availability
- Terminals/programmers
- Avg. CPU Utilization
- Accessibility of all resources

2.3 Study Results (Chart 5)

The results of this particular study were encouraging on the one hand and quite perplexing on the other.

2.3.1 Tool usage results showed that as the number and quality

of automated tools increased, there were significant increases in 3 of the 4 quality measures used in this study:

1. Productivity increased as tool usage increased

2. Maintainability (effort to change/effort to repair) improved as the number and quality of tools increased.

3. Reliability did not seem to be significantly impacted in this one particular study.

2.3.2 Computer Environment

Although all of the experimenters felt that there would be significant increases in all quality measures as the overall quality of computer support increased, none of the measures proved to be significant for this one particular study. It could not be shown that an improved computer support environment (at leastas far as the way the SEL described support environment) directly, favorably impacted the four quality measures used by the SEL.

This particular study is still undergoing further analysis.

2.3.3 Terminal Usage

The most perplexing result of this experiment study was the one in which the SEL attempted to assess the impact that increased number of terminals would have on the four measures described.

Although the experimenters expected to observe an increase in both productivity and software reliability as the number of terminals made available increased, the study found just the opposite. Both productivity and reliability of software decreased as the ratio of terminals available increased. There was no significance in the results for maintainability (effort to change/effort for repair).

Numerous suggestions have been put forth in attempting to explain this phenomena. Some felt that the increased terminal usage possibly was not properly accompanied with interactive support tools in the particular environment.

Another idea was that the increased terminal availability without proper training for the programmers led to a less disciplined approach by the programmers.

There are several other possible explanations of the results and for that reason, this particular study has been continuing and will be attempting to more thoroughly analyze this data as well as the additional projects that have been completed in this environment.

3.0 Software Testing

A second general set of studies that has been conducted over the past several years within the SEL has been directed toward gaining insight into approaches to testing software. Since this phase of the development life cycle had previously been determined to consume at least 30 percent of the development resources (Reference 5), it was deemed as a critically important discipline to study. Two major experiments were conducted during 1984 and 1985 in an attempt to:

1. Determine the overall coverage of software in the typical testing scenario utilized in the flight dynamics software development.

2. Investigate the relative merits of three standard testing approaches:

- o functional testing
- o structural testing
- o code reading

3.1 Test Coverage* (Chart 6 and Reference 6)

The first experiment on testing was designed to determine the extent to which typical testing techniques within the flight dynamics environment amply exercised the software that had been built. This particular environment utilizes functional testing during both the system test phase as well as the acceptance test phase.

By instrumenting a major flight dynamics system, then by executing the series of both system tests and acceptance tests - experimenters could first determine the coverage attained in the test phases. Next, the experimenters monitored the operational execution of this same software over a period of months to determine the extent to which portions of the completed software were utilized. Finally, the experimenters analyzed uncovered errors in an attempt to determine if the errors occurred in portions of the system that had not been exercised during the

*The lead investigator for this work was Jim Ramsey of Univ. of MD

test phase of development. The software studied was a major subsystem of a mission planning tool and consisted of 68 modules (Fortran subroutines) with 10,000 lines of code. There were 10 functional tests making up the acceptance test plan for the subsystem and during the operational phase, the experimenters monitored 60 operational execution of the software.

3.1.1 Test Coverage Results (Chart 7)

The managers of the flight dynamics development systems noted that the approach to testing had historically been quite good (relatively few errors found in operations) and they expected that the coverage found for this one experiment would be quite high (few modules would be not executed). The results of the experiment showed that for the 10 functional tests executed, only 75 percent of the 68 modules were executed and less than 60 percent of the total executable code was covered in the tests.

Additionally, the series of operational executions showed that a slightly higher percentage of both number of modules and lines of code were executed for this series of 60 executions.

Finally, all of the error reports were reviewed to determine in which portion of the system the errors had occurred. It was found that 8 errors had been recorded during the extended operational phase of the software, but it was found that none of the reported errors occurred in software that had not been executed during the acceptance test phase.

This initial study seemed to indicate that the functional testing approach was properly leading to correct portions of the system being executed and it also was very representative of the operational usage of the software.

The results of this study indicated that further investigations into the various approaches to testing may be worthwhile to determine just which approaches were most effective in uncovering errors in the software itself.

3.2 Software Testing Techniques* (Chart 8 and Reference 7)

Another study was conducted where three programs were seeded with a number of faults and 32 professional programmers from NASA/GSFC and from Computer Sciences Corporation (CSC) participated in an experiment to determine which techniques were effective in uncovering these faults.

The three testing approaches included:

*The lead investigator for this study was Rick Selby of Univ. of MD

- o Functional Testing
- o Structural Testing
- o Code Reading

All programmers participated in applying each of the three techniques.

When performing functional tests, the programmers were required to use the functional requirements along with test results to isolate faults - they were not to look at the source code itself until after testing was completed.

Those programmers performing structural testing used the source code and test results but did not use the functional requirements.

Code reading was carried out with no executions of the software. Those performing code reading reviewed the requirements and also looked at the source code.

3.2.1 Testing Technique Results (Charts 9 and 10)

The results of this experiment indicated that code reading is the most effective of the three testing techniques studied. This technique uncovered an average of 61 percent of all seeded faults while functional testing uncovered 51 percent and structural testing uncovered 38 percent.

Before the test, most of the managers in the SEL felt that code reading would prove to be a very effective testing technique, although they also felt that it would probably be the most costly in manhours to apply; but the results of the experiment indicated that code reading also was the most cost effective technique (3.3 faults per manhour vs 1.8 faults per manhour for structural and for functional testing). It was also noteworthy that, before the experiment, less than 1 out of 4 persons participating in the experiment predicted that code reading would be the most effective approach.

An additional observation that was made after the testing results were compiled was that there seemed to be a difference in the relative effectiveness of each of the testing approaches as the size of the software being tested increased. For the smaller program, code reading was by far the most effective technique, but for the larger program, functional testing seemed to be quite effective. This observation may indicate that there should be a size limit on how much code is utilized in a code reading exercise. Further tests are planned for these studies.

4.0 Software Measures

Over the past 6 to 8 years, the SEL has defined, studied, and evaluated numerous measures applicable to software development and management (References 8, 9, 10). Most of these measures have focused on one phase of the software life cycle - the code/unit test phase. In an attempt to define and apply measures in earlier phases of the life cycle, the SEL has been reviewing several approaches to qualifying or measuring aspects of the software during the specifications phase and during the design phase. Work on the specification phase was reported at the Ninth Software Engineering Workshopand may be foundinreference 11 and 12. One additional piece of work that has been conducted for the design phase will be discussed here.

4.1 Software Design Measures* (Charts 11 and 12 Reference 13, 14)

In an attempt to qualify software designs, a study was conducted to determine if module strength may be utilized as a guideline for software modularizatien. Although the definitions of strength may be well understood, the parameter may not be easy to determine based solely on a structure chart or data flow diagram which may be produced during the design phase of software development.

For the purposes of this study, strength is defined as the 'singleness of purpose' that a software module inherently contains. Singleness of purpose is a subjective parameter assigned at design time by the developer/manager. From a list of potential functionality that a component may have (e.g. computational, control, data processing, etc.) the programmer determines which functions that module contains. High strength would be attributed to those components which have but a single function to perform, medium to 2 and low strength would have three of more functions to perform.

The study examined 450 Fortran modules (from 4 systems) which were built by approximately 20 different developers.

Typical SEL data, which includes detailed cost and error data for all modules was also available for all of the modules. The 450 modules used for this study had a fairly even distribution in size as well as in design strength. Small modules (104 of the 450) were those with up to 31 executable statements, medium (148 of 450) were those with up to 64 executable statements and there were 151 large modules which had more than 64 executable statements.

*The lead investigators for this study were D. Card and G. Page of CSC and F. McGarry of NASA/GSFC

The objective of the study was to determine if strength of modules as determined at design time was related to the cost and reliability of the completed product.

4.2 Results of the Study on Strength (Charts 13, 14, 15)

The results of the study in the SEL indicated that module strength is indeed a reasonable criteria for defining software modularization. When examining the reliability of the 450 modules, it was found that 50 percent of the high strength modules had zero defects while for medium strength modules 36 percent had zero defects and low strength modules only 18 percent of the modules had zero defects. Similar trends were found for the modules of medium error proneness (up to 3 errors per 1000 lines of code) and for modules having a high error rate (over 3 errors per 1000 lines of code).

The distribution of the 'buggy' modules (over 3 errors per 1000 lines of code) was shown to tend more toward low strength as opposed to high strength. Forty-four percent of the buggy modules had low strength while only 20 percent of the buggy modules were found to have high strength.

Several additional observations were made while conducting this particular study. When the characteristics of the individual programmers were reviewed, it was found that those programmers who produced high quality software (low error rate and high productivity) tended to design modules of high strength but they also did not show a preference for writing modules of any specific size. Good programmers generated modules of size that seemed to best suit their design and they did not artificially constrain themselves to writing small modules.

5.0 General Trends and Observations

Over the past several years, the SEL has conducted numerous studies and experiments in an attempt to better understand the impact that various software techniques may have on producing improved software. In addition to the specific studies conducted such as the ones briefly discussed in sections 2, 3, and 4, the SEL has observed general trends in the development and measurement of software. The observations include such points as trends in software reuse, trends in utilization of improved software development technology, and the overall impact of improved developed techniques in the cost and reliability of software over a long period of observation time. Some of these general observations are summarized here.

5.1 Trends in Computer Use and Technology Application (Charts 16, 17)

From data that has been collected on nearly 60 projects over the past 9 years, one trend that has been noted is the tendency to make heavier and heavier usage of available computer support. In 1977 and 1978, computer use averaged approximately 100 runs per 1000 lines of developed source code while in 1982 and 1983 the average use increased to nearly 250 runs per 1000 lines of source. This trend continues to increase within the flight dynamics environment being studied.

Simultaneously, it was noted that the use of more and more structured development practices, improved management approaches and overall higher quality software engineering has continually increased. Each project has been rated on its application of over 200 software techniques (see reference 15) in an attempt to quantify the overall level of development and management technology utilized for a project. The aggregate of the total set of techniques applied results in a rating termed the Software Technology Index. From an average index of less than 100 in 1976 to 1978, it was found that the overall development techniques have increased to an average of over 140 in the 1980's. This seems to point to improved training, better discipline, improved access to tools and possibly better informed management practices.

Although both parameters (computer use and software technology index) seemed to generally increase over the past 7 or 8 years, there is no observed correlation between these two factors.

5.2 Trends in Software Reuse (Chart 18)

Another general observation that was made from the detailed development data collected by the SEL, was that the reuse of software has shown general trends of increase. Typical software systems in the years 1977 to 1979 averaged about 15 or 20 percent reused code while in the 1982 to 1984 timeframe the average reuse has increased to 30 to 35 percent.

Although this reuse is certainly tending in the right direction, the SEL has not conducted detailed studies to determine what the driving factors are in improving the percentage of reuse. The trends are probably indicative of improvements in design technique as well as numerous other factors, but studies have just recently been initiated in the SEL to determine how the trend can be improved at a even faster pace.

It has also been observed in the SEL data that there does not

seem to be a direct relationship between projects that are rated as having a high software technology index and having a high rate of software reuse. But this may not be a surprise since one would expect that high technology usage would lead to follow on systems being able to pick up or reuse software produced by the projects using disciplined approaches for development and management.

5.3 Impact of Development Technologies (Chart 19)

Probably the most basic goal that the SEL has, is to determine the impact that specified software development/management techniques have on the cost and reliability of software. With nearly 60 projects having been closely monitored over the past 8 or 9 years,the SEL attempted to look at general trends inthe reliability and cost of these projects as measured against the software technology index computed for each of these projects. The 200 parameters factored into this index represent everything from structured techniques to disciplined management approaches to configuration control procedures. It is one attempt to characterize each of the projects with a single value.

This technology index correlates very well(r = .82)with reliability of software in the SEL. Those projects with a higher rating of good development practices were the projects with the lower fault rates of the product.

Unfortunately, the impact of this technology index on productivity is quite unclear. The first general observation that has been made is that there is not a clear favorable impact on development cost (cost per line of code) with projects with higher values of this technology index. Studies are continuing in an attempt to more objectively compute this technology rating so that a more conclusive statement can be made. Some researchers also have suggested that it is not to be unexpected that the specific development cost may not decrease but since the reliability has improved and the overall software structure has improved, the maintenance activity will be the beneficiary of the overall cost savings, not the development cost.

5.4 Can Software Technology be Measured? (Chart 20 and Reference 3)

Another major question that software engineers address is whether or not software technology can be measured at all. By utilizing reliability as one major aspect of software quality, the SEL attempted to determine to what extent software development/ management practices could be measured.

There are three levels of development practices which the SEL has hoped and attempted to measure. First, there are individual specific techniques such as the use of structured code or chief programmer team or the use of PDL in design, etc.

Second, there is the usage of a software methodology which is a combination of several methods into a single disciplined approach. This could be the set of methods known as structured techniques which reflect the use of 6 or 8 individual practices such as top down development, structured code, code reading and usage of Unit Development Folders (UDF).

Finally, the attempt has been made to measure the impact of the total technology index which encompasses all disciplined management/development practices. This signifies the level to which the project has attempted to apply recommended software development techniques.

The results of this study indicated:

1. An individual technique cannot be effectively measured in a production environment such as the one in which the SEL is conducting studies. ($r = .37$ is a typical value found in correlating PDL usage and reliability).

2. Disciplined methodologies (combining techniques into a single disciplined approach) can be measured ($r = .65$ for one particular study) and the approaches called Modern Programming Practices (6 techniques) has a significant, measurable, favorable impact on software reliability.

3. Total Software Technology can be measured ($r = .82$ for this one study) and higher levels of applied technology have a marked favorable impact on the reliability of software.

The trends and observations noted here are based on approximately 8 years of data collection and experimentation within the SEL. Approximately 55 projects have been studied and the research is continuing and will continue in the future.

Many of the results are inclusive, but with each experience and study, greater insight is provided into the overall characteristics of the software development process.

REFERENCES

1. Software Engineering Laboratory, SEL 81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et. al, February 1982.

2. SEL, 81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et. al, August 1982.

3. SEL, 86-002, Measuring and Evaluating Software Technology, D. N. Card, F. E. McGarry, J. Valett, to be published

4. McGarry,F.; Valett, J.; and Hall, D., 'Measuring the Impact of Computer Resource Quality on the Software Development Process and Product', Proceedings of the Hawaiian International Conference on Systems Sciences, January 1985

5. McGarry, F., 'What Have We Learned in 6 Years', Proceedings of the Seventh Annual Software Engineering Workshop, December 1982

6. Ramsey, J., and V. R. Basili, 'Analyzing the Test Process Using Structural Coverage', Proceedings of the Eighth International Conference on Software Engineering, August 1985

7. SEL 85-0001, Comparison of Software Verification Techniques, D. Card,R. Selby, F. McGarry, et. al, April 1985

8. SEL, 82-004, Collected Software Engineering Papers: Volume 1, July 1982

9. SEL 83-003, Collected Software Engineering Papers: Volume 11, November 1983

10. SEL 85-003, Collected Software Engineering Papers: Volume 111, November 1985

11. SEL 84-003, Investigation of Specification Measures for the Software Engineering Laboratory, W. Agresti, V. Church, F. McGarry, December 1984

12.Agresti, W.; 'An Approach to Developing Specification Measures; Proceedings from the Ninth Annual Software Engineering Workshop, November 1984

13. Card, D.; Page, G; McGarry, F.; 'Criteria for Software Modularization, Proceedings of the Eighth International Conference on Software Engineering, August 1985

14. Agresti, W.; Card, D.; Church, V.; 'Status Report on Specification and Design Metrics Studies', CSC, December 1985

15. SEL 82-001, 'Evaluation of Management Measures of Software Development', D. Card, G. Page, F. McGarry, September 1982

THE VIEWGRAPH MATERIALS

for the

F. McGARRY PRESENTATION FOLLOW

STUDIES AND EXPERIMENTS IN THE SOFTWARE ENGINEERING LABORATORY

December 4, 1985

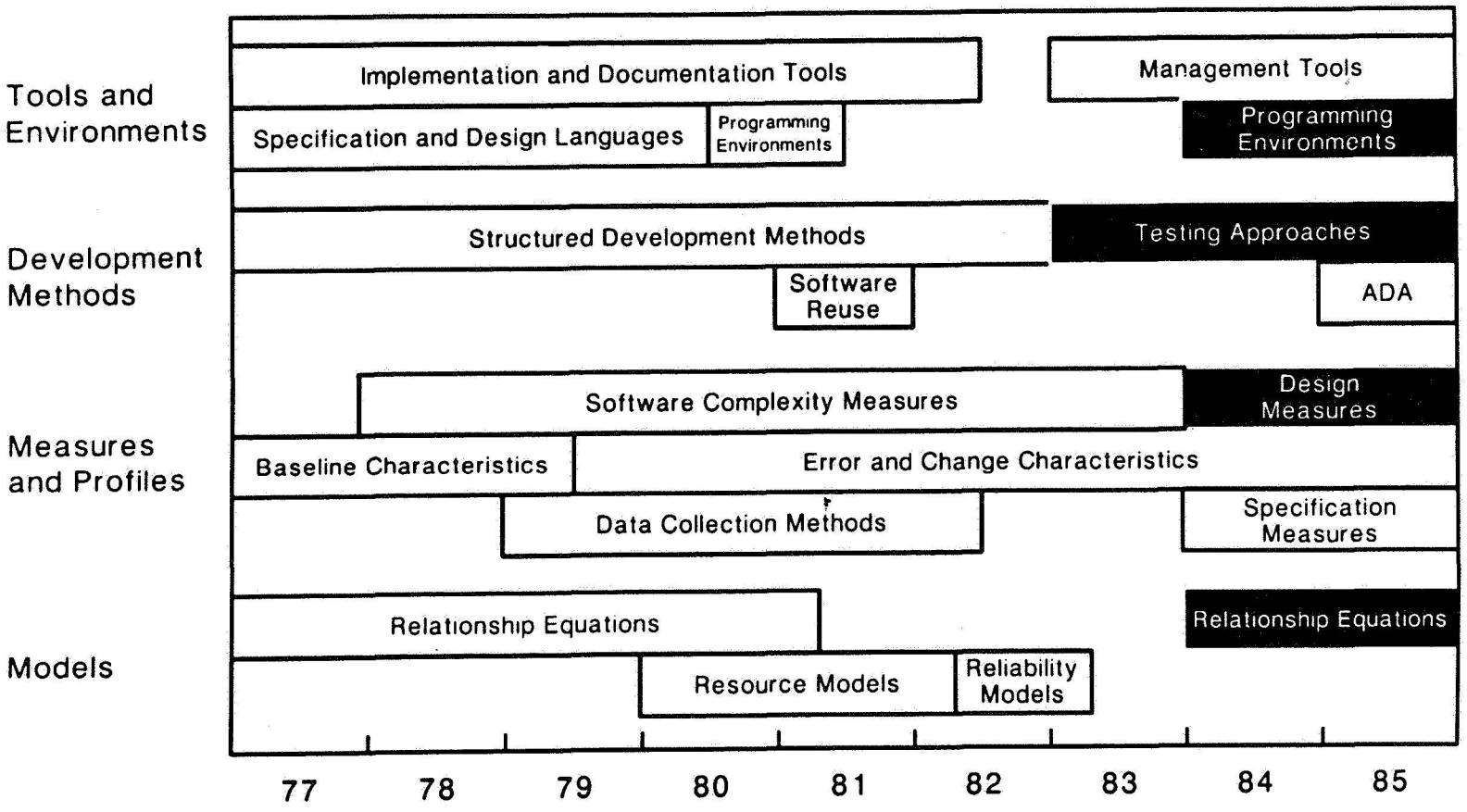
Tenth Annual Software Engineering Workshop

86A0553.04

F. McGarry
NASA/GSFC

CHART 0

SEL RESEARCH TIMELINE



86A0553.13

F. McGarry
NASA/GSFC

CHART 1

MEASURING THE EFFECTS OF ENVIRONMENT ON SOFTWARE DEVELOPMENT

1. Effect of Software Tools (eg. Design Aids, Editors, Auditors)
2. Effect of Overall Computer Support (eg. Turnaround Time, Interactive vs. Batch)
3. Effect of Number of Terminals/ Programmer

ON . . .

- Productivity (LOC/ Man-Month)
- Reliability (Errors/ KLOC)
- Effort to Change (Time to Change Software)
- Effort to Repair (Time to Repair Errors)

86A0553.01

F. McGarry
NASA/GSFC

EXPERIMENT

- 14 Projects of Same General Type in Varying Environments
- Project Size Varies From 11 KLOC to 136 KLOC
- Projects Rated on Various Parameters, Giving Indication of Quality of Environment
- Examined for Correlations Between Ratings and 4 Measures

86A0553.09

ENVIRONMENT VARIATIONS

		Low Rating	High Rating
	Terminals/ Programmer	1/8	1/2
	Number of Tools	2	8
Tools	Tool Usage	Very Low	Very High
	Tool Quality	Very Low	Very High
	Turnaround Time	>1 Day	<2 Hours
Computer	Response Time	>20 Seconds	<5 Seconds
Environment	Batch vs. Interactive	All Batch	All Interactive
	•		
	•		
	•		

86A0553.10

F. McGarry
NASA/GSFC

RESULTS

	Productivity	Reliability	Effort to Change	Effort to Repair
Tool Support	+	0	+	+
Computer Environment	0	0	0	0
Terminals Per Programmer	-	-	0	0

+ = Positive Correlation - for Reliability, Effort To Change, and Effort To Repair - This Implies Lower Errors or Time

- = Negative Correlation

0 = No Correlation

86A0553.07

F. McGarry
NASA/GSFC

TEST COVERAGE

Objective

- Determine Characteristics of Functional Testing in One Environment (Acceptance Testing)
  - % of Code Executed
  - % of Modules Executed
- Compare Acceptance Test Profile With Operational Usage
  - % Code and Modules Executed
  - Profiles of Errors Found

Data for Study

- 1 Flight Dynamics Program
- 68 Modules
- 10K SLOC
- 10 Functional Acceptance Tests
- 60 Operational Use Cases

86A0553.12

F. McGarry
NASA/GSFC

TEST COVERAGE RESULTS

	Acceptance Test	Operational Use
% Code Executed (Total)	56%	65%
% Code Executed (Every Test)	18%	10%
% Module Executed (Total)	75%	80%
% Modules Executed (Every Test)	42%	27%

- 8 Faults Uncovered During Maintenance (Not in Untested Modules)
- Acceptance Tests Were Very Representative of Operational Usage
- For This Environment - Functional Testing Is Good Approach

86A0553.02

F. McGarry
NASA/GSFC

STUDIES OF SOFTWARE TESTING TECHNIQUES

32 Professional Programmers (GSFC and CSC)

3 Expertise Levels: Advanced, Intermediate, Junior

3 Fortran Programs: Seeded With Faults

2 Computers for Online Testing: IBM 4341, VAX 11/780

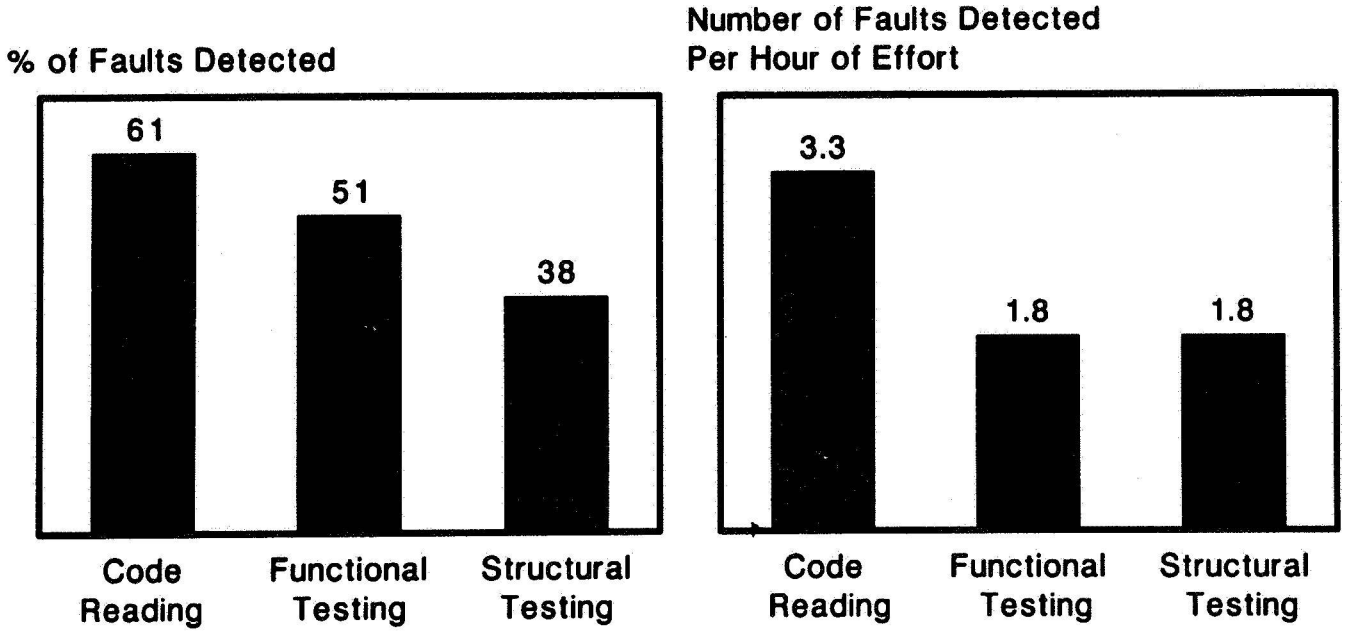
3 Verification Techniques:

	Code Reading	Functional Testing	Structural Testing
View Program Specification	Yes	Yes	After Testing
View Source Code	Yes	After Testing	Yes
Execute Program	No	Yes	Yes

86A0553.11

F. McGarry
NASA/GSFC

SOFTWARE TESTING RESULTS



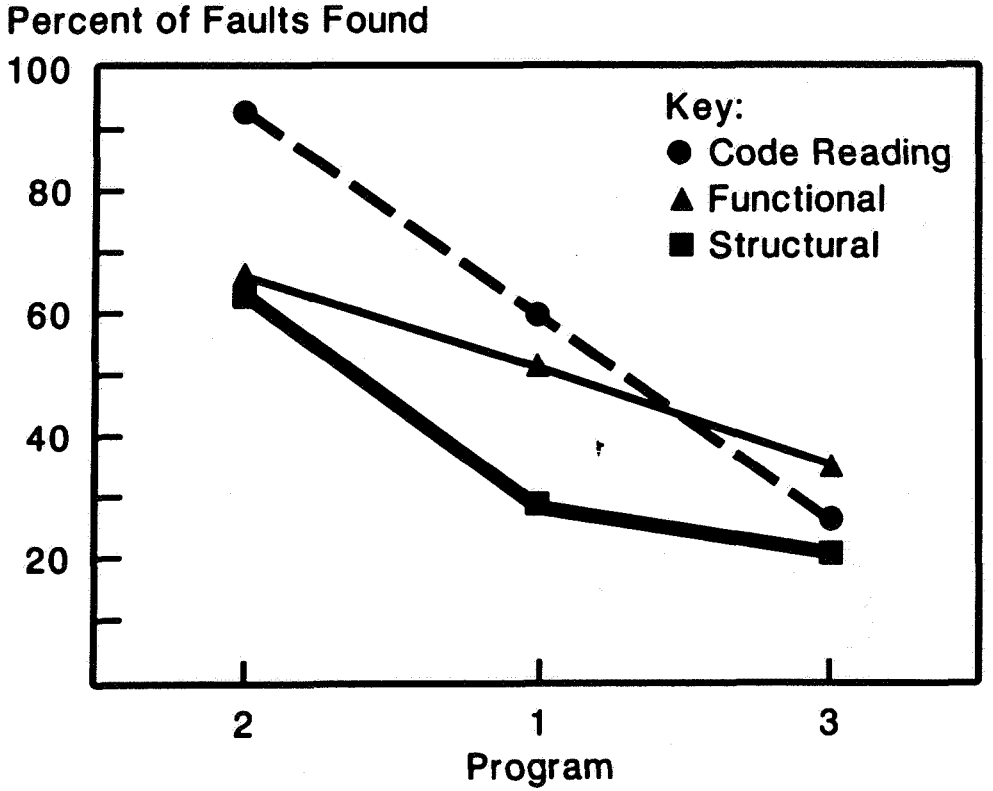
Code Reading Proved To Be the Best Technique in Terms of the Total Number of Faults Detected and the Faults Detected Per Hour of Effort

Prior To the Experiment Only 23% of the Subjects Believed Code Reading To Be the Most Effective Technique

Advanced Subjects Performed Code Reading and Structural Testing Better Than Intermediate or Junior Subjects

86A0553.16

TESTING TECHNIQUES VS. PROGRAM SIZE



Functional Testing May be More Effective for Larger Programs

86A0553.14

Note: Programs Ordered According to Size

F. McGarry
NASA/GSFC

SOFTWARE DESIGN MEASURES

Objective

- Evaluate Strength* and Size as Criteria for Software Modularization

Data for Study

- 450 Fortran Modules
- Approximately 20 Different Developers
- Detailed Cost and Error Data on All Modules
- Detailed Design Descriptions By Programmers at Design Time

* Types (and Numbers) of Functions Performed by Module - Determined by Programmer

86A0553.05

F. McGarry
NASA/GSFC

SOFTWARE DESIGN MEASURES

Strength Distribution

Module Strength	Number of Fortran Modules	Mean Executable Statements	Mean Decisions Per Executable Statement
Low	90	77	0.29
Medium	176	60	0.32
High	187	48	0.32

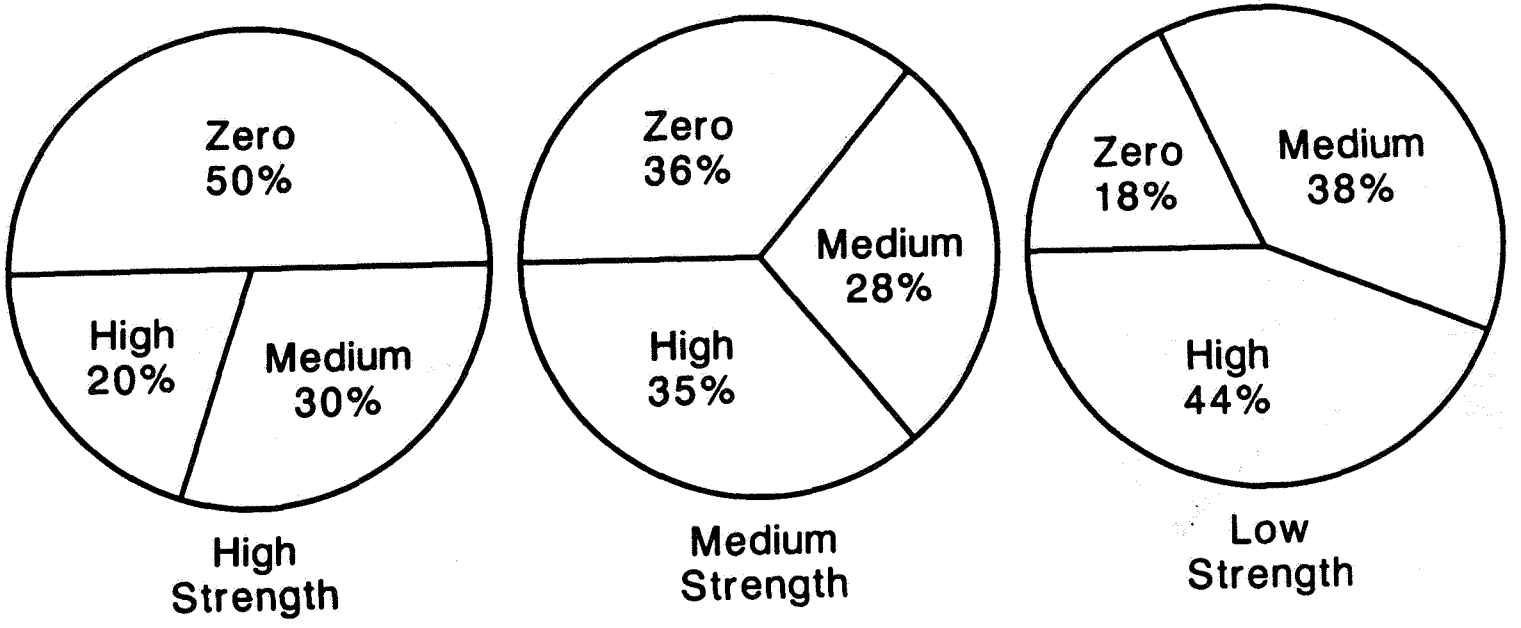
Size Distribution

Module Size	Number of Fortran Modules	Executable Statements	Mean Decisions Per Executable Statement
Small	154	1 to 31	0.31
Medium	148	32 to 64	0.31
Large	151	65 or More	0.32

86A0553.15

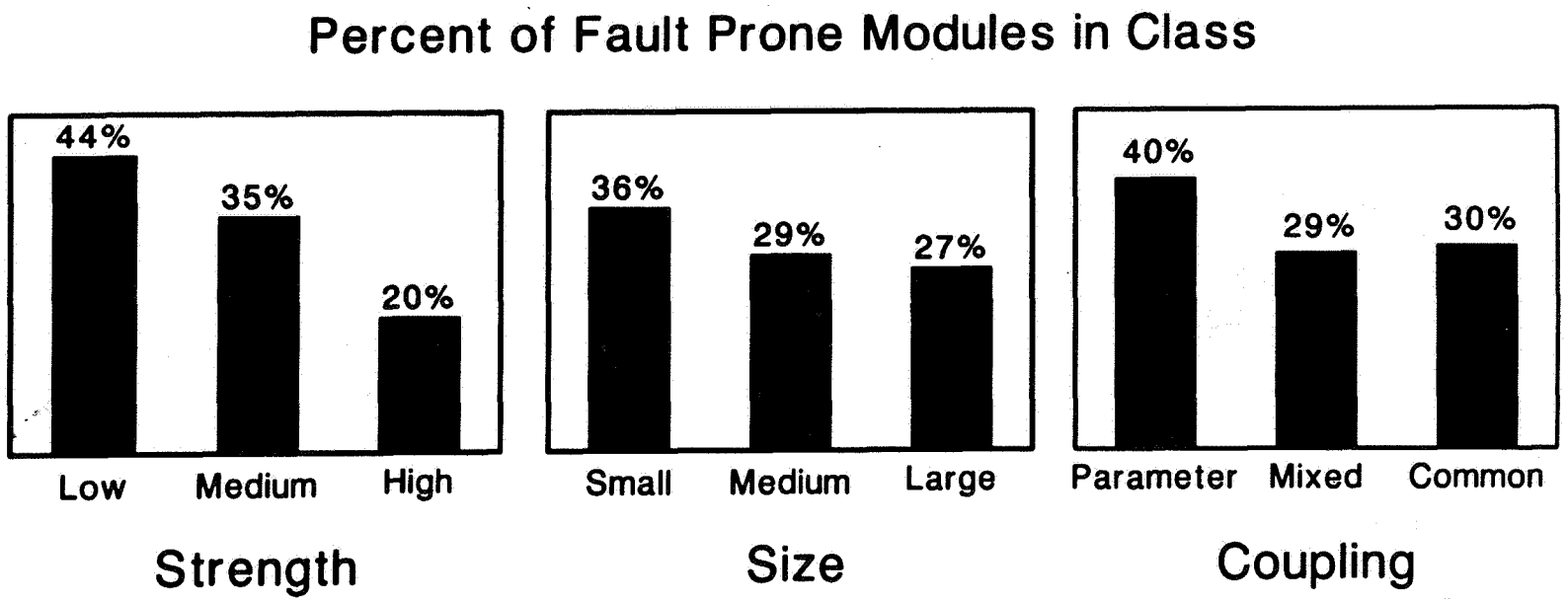
F. McGarry
NASA/GSFC

FAULT RATE FOR CLASSES OF MODULE STRENGTH



86A0553.19

DESIGN CHARACTERISTICS



C-2

86A0553.20

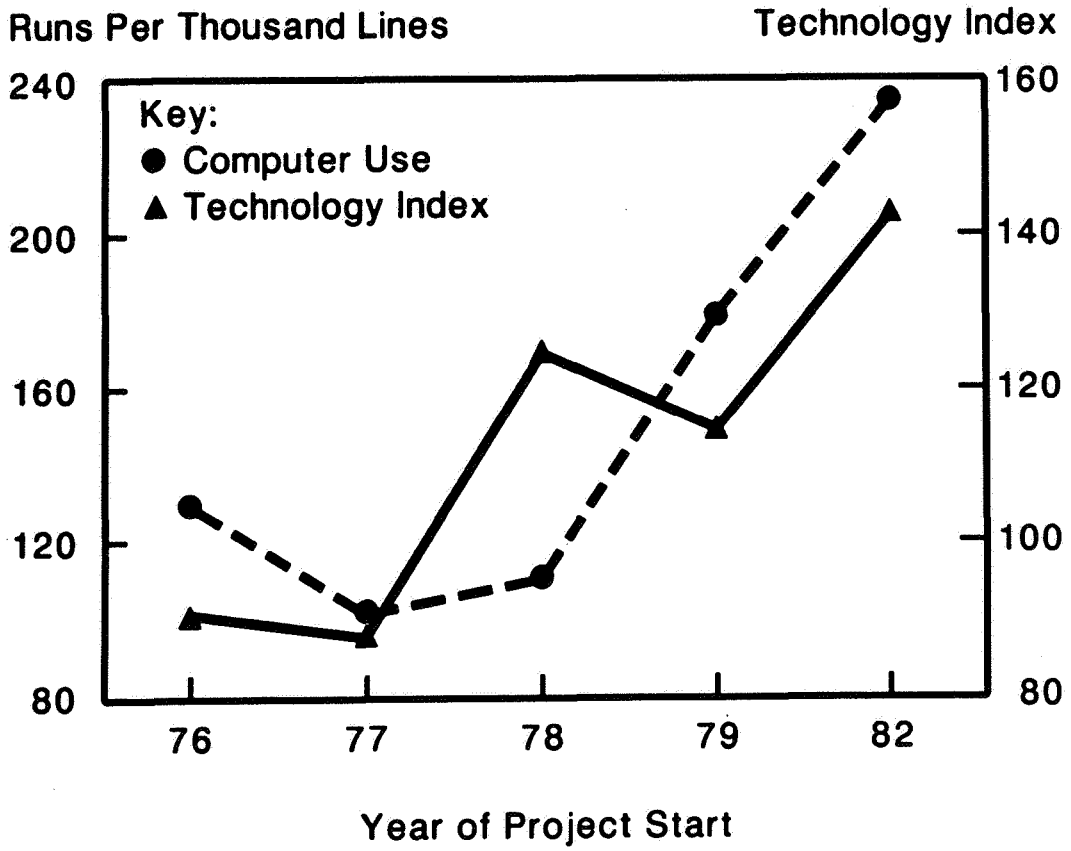
CHART 14

DESIGN MEASURES SUMMARY

- Good Programmers Tend To Write High-Strength Modules
- Good Programmers Show No Preference for Any Specific Module Size
- Overall, High-Strength Modules Have *A Lower Fault Rate and Cost Less* Than Low-Strength Modules
- Overall, Large Modules Cost Less (Per Executable Statement) Than Small Modules
- Fault Rate Is Not Directly Related to Module Size

86A0553.08

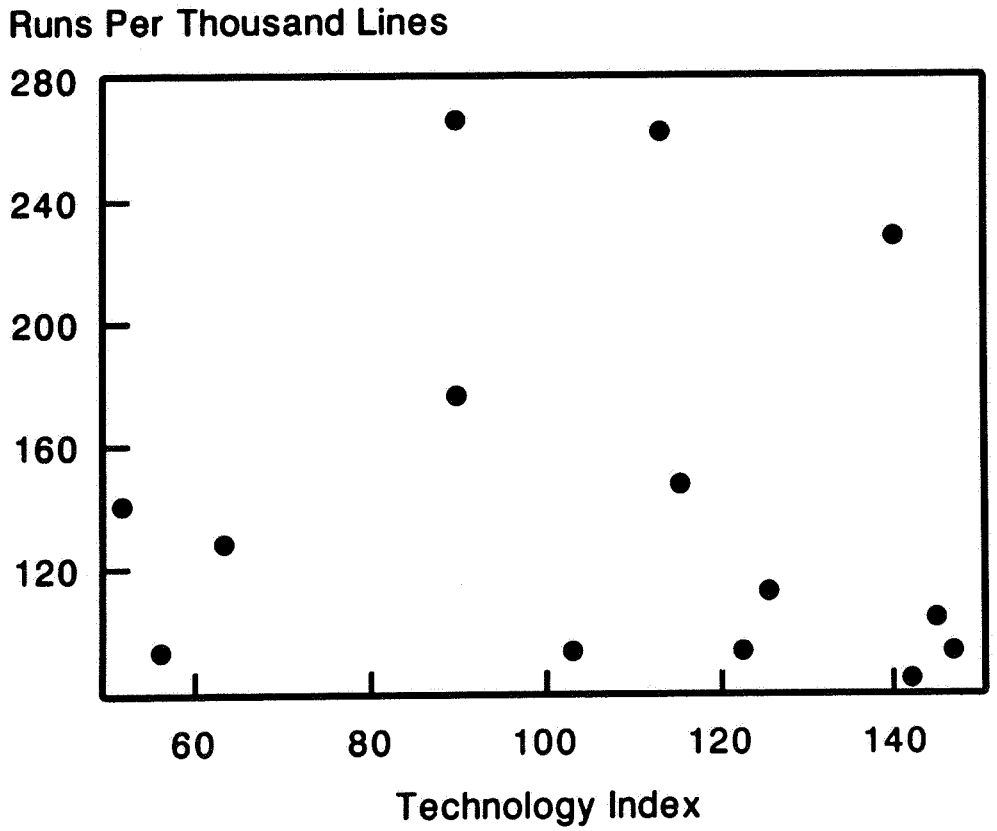
COMPUTER USE AND TECHNOLOGY TIME TRENDS



86A0553.18

F. McGarry
NASA/GSFC

EFFECT OF TECHNOLOGY ON COMPUTER USE

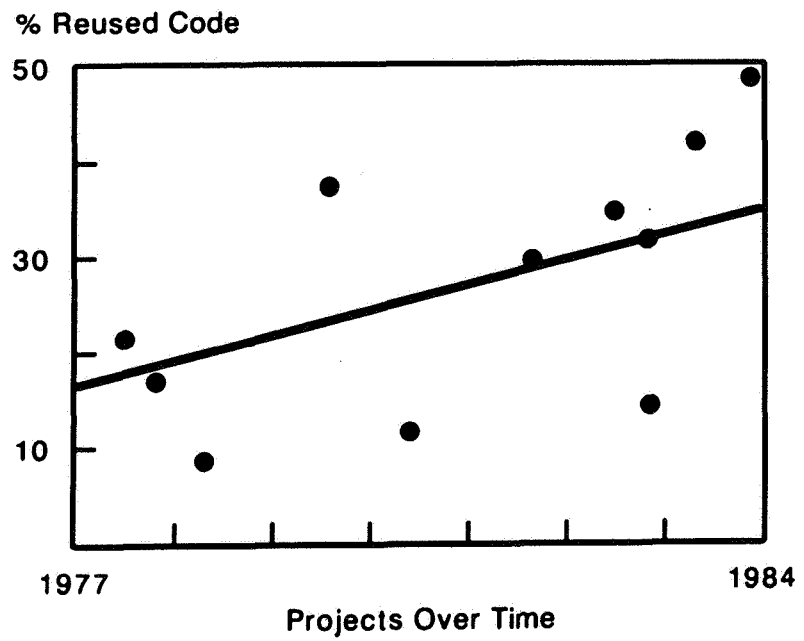


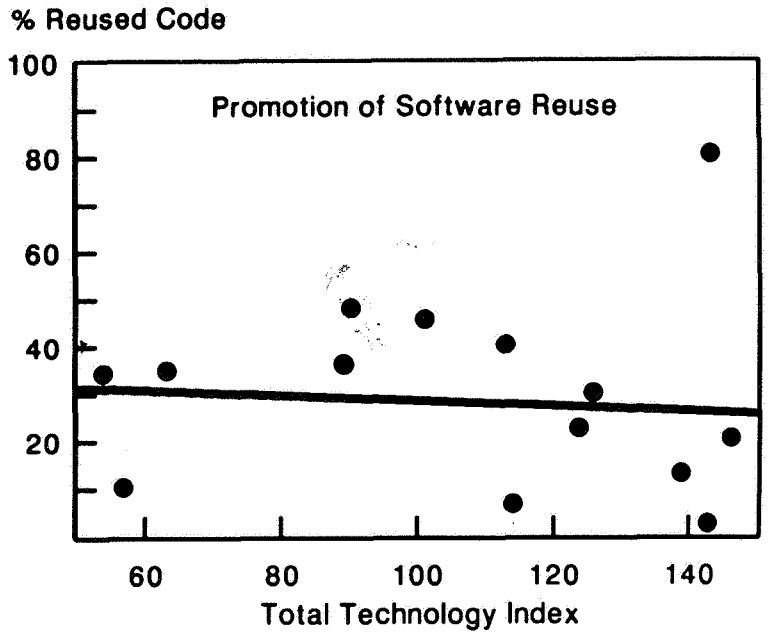
No Obvious Correlation Between Computer Use and Technology Use

86A0553.21

F. McGarry
NASA/GSFC

TRENDS IN SOFTWARE REUSE (BASED ON 15 PROJECTS OF SIMILAR CHARACTERISTICS)



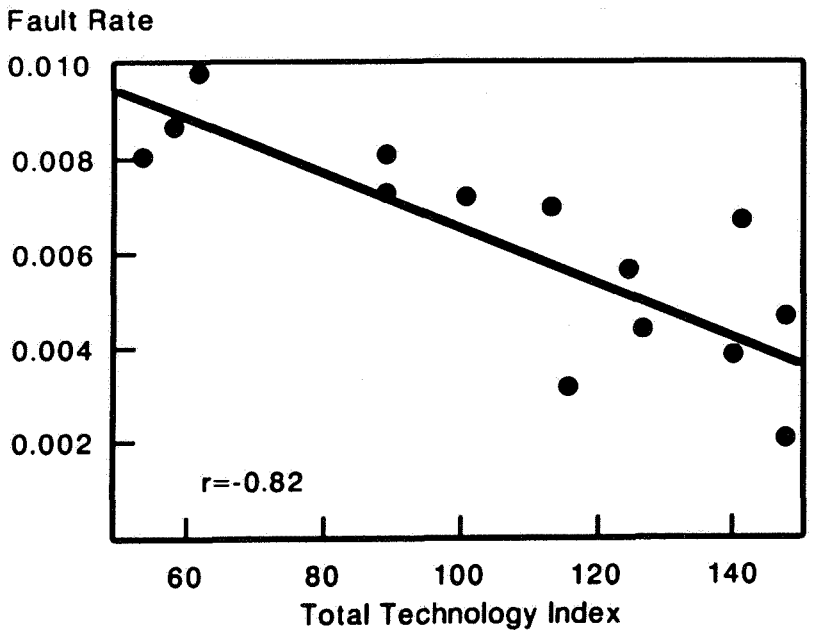


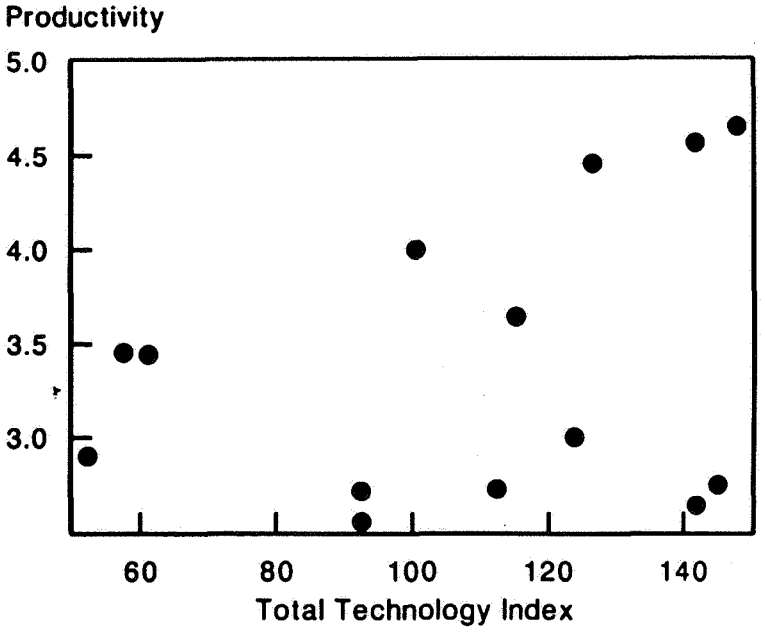
- Software Reuse Has Significant Potential as a "Technology"
- Software Reuse Has Been Increasing Without Directed Efforts
- Currently We Don't Understand Why Software Reused (or Not)

86A0553.22

F. McGarry
NASA/GSFC

EFFECTS OF DEVELOPMENT TECHNOLOGIES

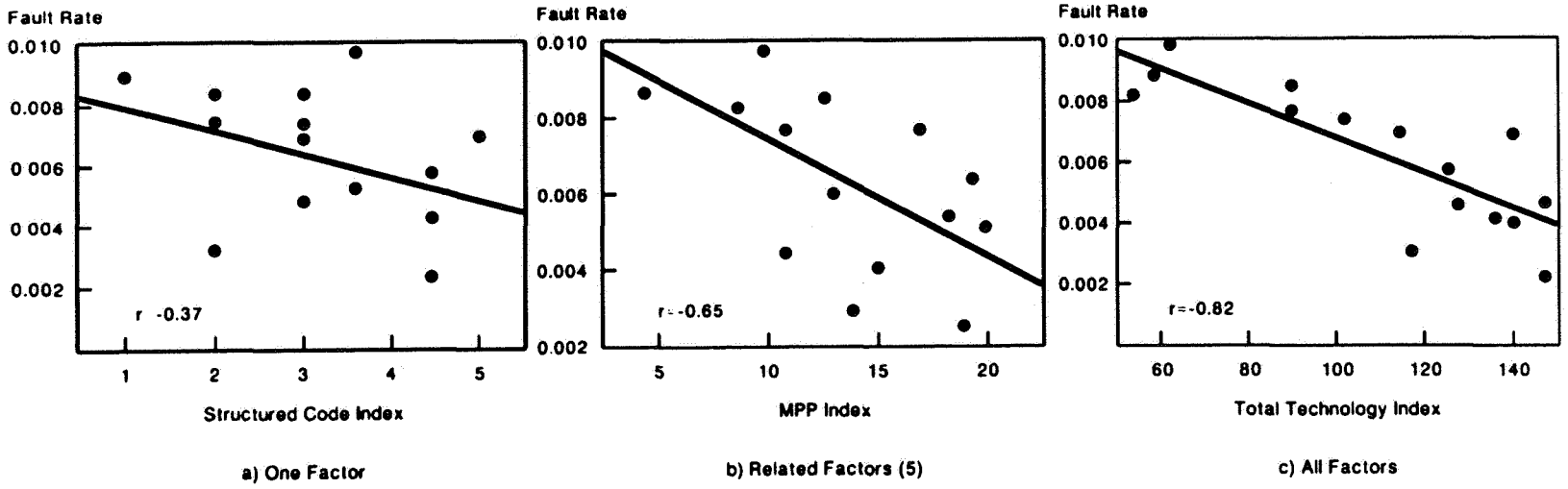




- Reliability Can Be Favorably Impacted
- Productivity Is Sensitive to too Many Other Factors

86A0553.17

EFFECT OF TECHNOLOGY USE ON SOFTWARE RELIABILITY



- Individual Techniques Are Difficult To Measure
- Integrated Methodologies Favorably Impact Quality

86A0553.23

PANEL #2

TOOLS FOR SOFTWARE MANAGEMENT

D. Reifer, Reifer Consultants Inc.
J. Valett, NASA/GSFC
J. Knight, University of Virginia
G. Wenneson, Informatics General Corporation

SOFTWARE MANAGEMENT TOOLS: LESSONS LEARNED FROM USE

Donald J. Reifer, President
Reifer Consultants, Inc.
25550 Hawthorne Blvd.
Torrance, California 90505

Abstract: Over the last five years, considerable progress has been made in the area of software resource estimation, management and control. Numerous tools have been developed and been put into use that allow managers to better plan, schedule and control the allocation of the time, workforce and material needed to develop their software products for NASA applications. Currently, over 300 commercially available software project management tools exist including about 180 project scheduling and control packages for an IBM personal computer-based workstation[1]. In addition, numerous tools exist for estimating software costs, measuring software progress through earned value concepts which rely on reporting milestone completions, maintaining configuration integrity over the software product data bases and measuring software quality. The literature is full of promises and details when it comes to these tools and it becomes confusing when you try to sort out what they really can and can't do when you read the sales fiction. In addition, much of the experience associated with transitioning these tools onto operational projects where managers are trying to use such aids to reduce the time it takes them to plan and control the delivery of their complex software products has not been recorded or shared.

The purpose of this presentation is to remedy this situation by discussing the author's recent experiences in inserting software project planning tools like those mentioned above onto more than 100 projects producing mission critical software. The author will briefly summarize the problems the software project manager faces and then will survey the methods and tools that he has at his disposal to handle them. He will then discuss experiences his firm and users of the RCI developed Project Manager's Workstation (PMW) and the SoftCost-R cost estimating package have had over the last three years. Finally, he will report the results of a survey conducted by his firm which looked at what could be done in the future to overcome the problems experienced and build a set of usable tools that would really be useful to and used by managers of software projects.

THE PROJECT MANAGER'S WORKSTATION

The Project Manager's Workstation (PMW) was a prototype system that was built 3 years ago for a military client to research the following issues:

1. What tools does a software manager really need and what tools will he really use on the job?

2. What are the criteria which govern the acceptability of management tools by managers, not computer scientists?

3. Can management data be bridged between commercial tools developed by different manufacturers and resident on different machines?

---

[1] P. Kane, J. Bruscino, T. Pillsbury, D. Reifer and B. Strahan, Project Management Tool Survey Report, Note RCI-TN-145, 29 March 1985.

The PMW is a collection of management tools that runs on a dual floppy IBM personal computer with 512 KB. It has the following capabilities: resource planning, scheduling and control via a Work Breakdown Structure (WBS); Gantt and PERT chart (tabular and graphical) preparation and drawing; user-oriented report generation for cost-to-completes, schedule-to-completes and earned value determination; local bridges to packages like 1-2-3 and dBase on the personal computer and global bridges to packages like PAC-II and VUE on mainframes; and a personal time manager which allows relational development and searches of action item lists, calendars, distribution lists and telephone lists.

The PMW was designed as a rapid prototype with both usability and technical capability in mind. We hoped to learn from it as we put it into prototype use within organizations who were willing to try to employ it on their projects. It has been distributed to over 200 people over the last 3 years. Each user was required to attend a hands-on course on the system where he/she was taught how to use the package for managing a software project. A generic WBS was developed and inserted into the package to guide its users in consistent work task identification and cost data collection.

Recently, RCI surveyed the users of the package to get their feedback and to understand what their real requirements were when it came to project management tools. It was interesting to learn the following:

- The man/machine interface design makes or breaks the system. The user interface must be easy to learn and easy to use. It should be picture-oriented, function key driven and menu-based. Tool designers shouldn't assume managers know how to type, use a computer and/or will read manuals. They won't based upon our experience. To combat this, the package must have built-in "HELP" and safeguards against inappropriate usage.

- Most managers object to project management systems because they are required to do a lot of data input. Managers do not have the time, desire or skill to do it and often, don't do it right. Subordinates don't have the knowledge or the experience to do it correctly. Therefore, the system must support both working together to relieve the manager of the drudgery of getting the first set of workable plans into the system. To combat this, many tool designers should looking at "games" and should try to adapt their concepts to making data inputting "fun".

- Most vendors do not mechanize all the features and functions they put in their manuals. This makes it extremely difficult to interface packages together into an integrated system. File interchange performance is the critical issue because management users will not tolerate lengthy delays in getting responses to their questions. In the development of the PMW, we had to drop about half of the candidate packages from consideration and build our own modules to replace them as a result. Tool designers should therefore only rely on a core set of capabilities when they plan to use commercial packages.

- Global bridging or linking a micro-based tool to a mainframe-based system is much more difficult than first expected. Vendors do not

like to give you the file interchange formats and reverse engineering is the only alternate solution to getting this needed information. As a consequence, it took us 3 times more effort than originally planned to provide this capability. Tool designers should not count on the vendors of packages to make their jobs easy. Instead, they should adopt a standard file format like DIF and consider only packages that implement it.

- According to our users the most useful tools were work planning oriented, the most used tools were time management oriented and the most wanted tools were "what-if" oriented. This is not surprising and should be factored into future system designs.

- Because the state-of-the-art is moving towards networking, managers wanted to evolve their tools so that they could interrelate what their people were doing at different sites via their management tools. According to their wish lists, they wanted to do things like schedule a meeting on their people's calendar electronically and to preview deliverables in their work units libraries via remote inquiry privileges.

SOFTCOST-R

In another effort, RCI developed a cost estimating package based upon the work of Dr. Robert Tausworthe called SoftCost-R[2]. In essence, RCI spent six person years of effort to productize the experimental work done for the Jet Propulsion Laboratory. SoftCost-R is hosted on an IBM personal computer and versions exist for all of its models including the PC/XT and PC/AT. The primary feature RCI implemented was usability. Learning from our PMW experiences, we built a user-friendly screen editor to make the package easy to learn and easy to use. Since we introduced our product earlier this year, over 20 organizations have acquired it and are using it to predict their costs. Most of these organizations work on small to medium-sized projects developing software for embedded applications. The capabilities of SoftCost-R are similar to other parametric and statistical cost models on the market today like COCOMO, PRICE/S and SLIM. The key difference has to do with the ease with which the management user can employ the model to answer the "what if" questions he so desperately needs to answer.

Again, RCI surveyed its users and members of its development team to determine what lessons could be derived from its experiences to-date. This was very valuable to us because we were in the midst of planning enhancements to our current product and wanted to factor these lessons into our future releases. It was interesting to learn:

- The number one issue on the minds of management when it comes to costing is sizing. How can one determine in advance how big the program will be when you don't have the foggiest idea of what the system architecture will be was one of the comments heard during one of our interviews. While some research in this area is underway, managers will be reluctant to accept the results of cost models unless some of it pans out.

---

[2] Robert C. Tausworthe, Deep Space Network Software Cost Estimation Model, JPL Publication 81-7, 15 April 1981.

- Most of our users employed at least two cost models to cross check each's results. The most popular model was COCOMO and most of our users employed it manually from the book.[3] The reason for this popularity seemed to be its availability. Unfortunately, many users in our survey did not seem to fully understand the model's scope or limitations and were misusing it on the job.

- Calibrating a cost model to the organization using it is the hard part. Most organizations using our model did not have cost data available to either calibrate the model or validate its accuracy. Even if they had data, it was hard to make any sense out of it. Less than 5% of our users collected cost data as a norm and few had a framework in place for cost estimating. While cost models like SoftCost-R forced these organizations to gather data, most of it was not statistically homogeneous. Models must therefore be architected so that their calibration points and sensitivities are known and easily altered. In addition, the model must come with a known calibration data base in order for its users to have enough confidence in the model to believe its results.

- Non-management user's put too much reliance on models. Because a model gives them an answer, many believe it is right and don't do any more homework.

- Management user's tend to be more skeptical and don't believe the results of models even if they are perfectly calibrated to their projects and their environments (which they are not). Often, this is because managers really don't want to know the truth - the software is going to cost more than they expected and they don't have sufficient budget allocated for it.

- Many simple and mundane packaging concepts can make a model acceptable to a management user who will sacrifice capability to get something he can get answers from. Good user engineering goes a long way with managers who neither have the time nor the desire to become professional parameticians.

CONCLUSIONS

While the results reported seem logically and self-apparent, few seem to have paid attention to them in the past. Considerable attention needs to be paid to the packaging of tools when they are exported to production organizations from tool developers. The author sincerely hopes that this presentation will stimulate renewed emphasis on this important topic. Afterall, the results are based upon a survey of over 200 management users and are not only the author's opinion.

---

[3] Barry W. Boehm, Software Engineering Economics, Prentice-Hall, 1981.

THE VIEWGRAPH MATERIALS

for the

D. REIFER PRESENTATION FOLLOW

SOFTWARE MANAGEMENT TOOLS: LESSONS LEARNED FROM USE

4 DECEMBER 1985

PRESENTATION AT THE 10th ANNUAL
NASA/GSFC SOFTWARE ENGINEERING WORKSHOP

D. Reifer
Reifer Consultants
6 of 22

Reifer Consultants, Inc.

25550 Hawthorne Boulevard, Suite 208/Torrance, California 90505

PURPOSE OF BRIEFING

- To discuss our experiences with our software management tools
  - Project Manager's Workstation (PMW)
  - SoftCost-R Estimating Package
- To communicate the lessons we have learned - the hard way through use
- To hopefully influence your efforts and help you avoid the mistakes we have made

THE MANAGEMENT PROCESS

BEGINS BY:

- PLANNING
- ORGANIZING
- STAFFING

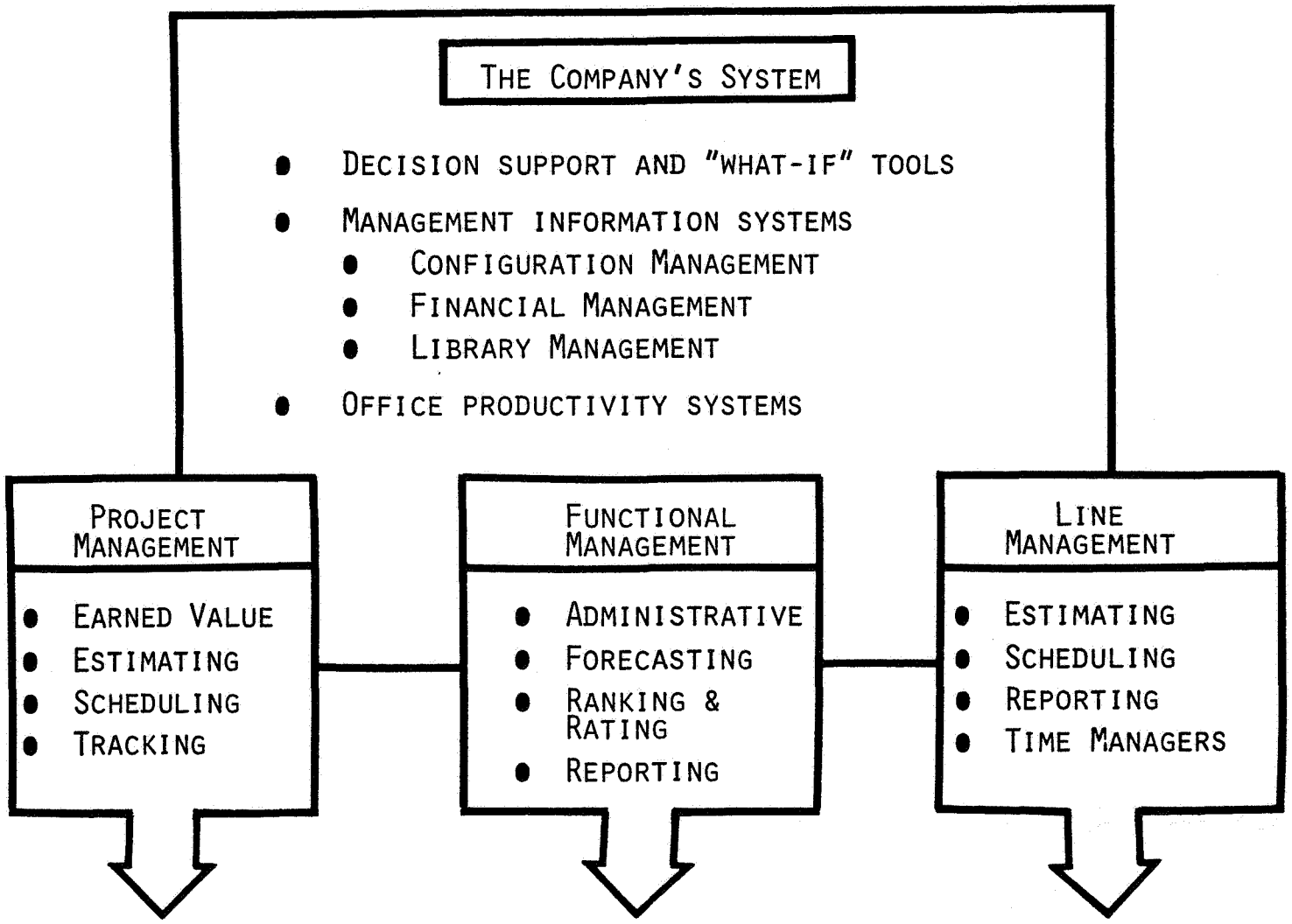
→ THE TEAM - AND THEN

- COMMUNICATING
- MOTIVATING
- INTEGRATING
- MEASURING
- CONTROLLING
- & DIRECTING

THE EFFORTS OF THE TEAM

THROUGH AN INTERATIVE PROCESS

NECESSARY MANAGEMENT TOOLS



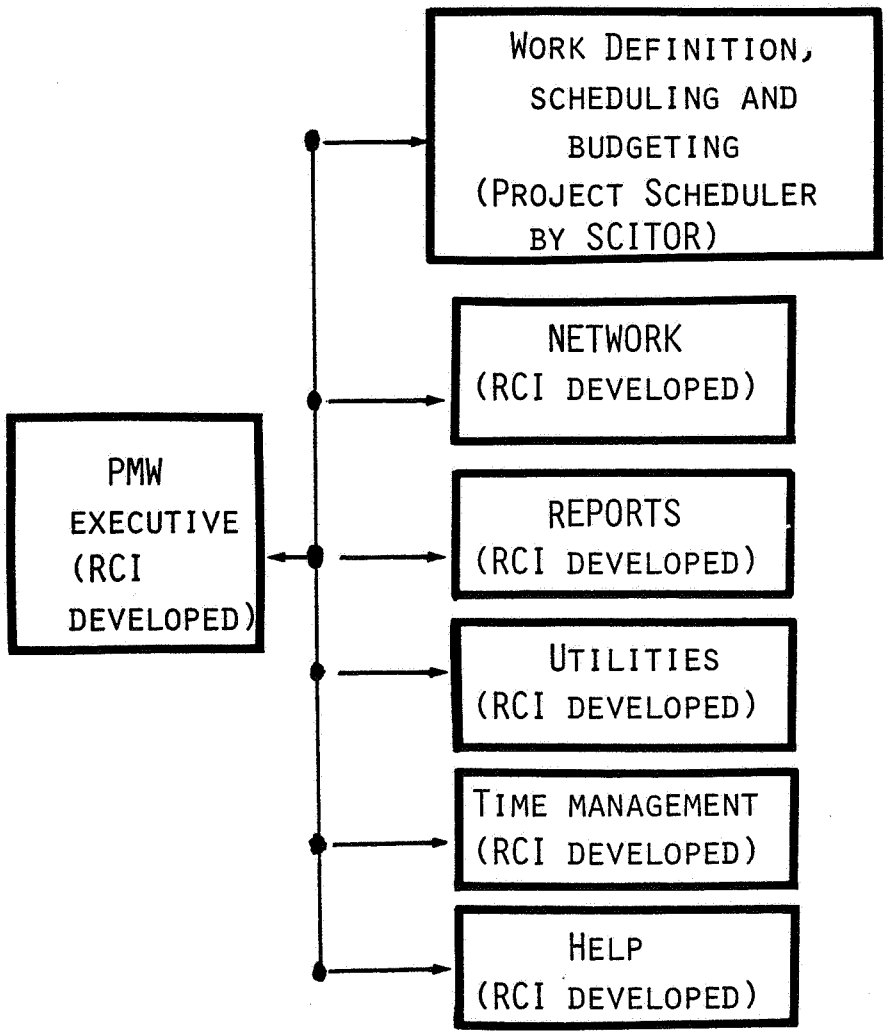
OVER 300 PACKAGES EXIST TO SUPPORT THESE FUNCTIONS

PMW: AN OVERVIEW

PMW IS AN EXPERIMENTAL, INTEGRATED PROJECT MANAGEMENT PACKAGE

- DEVELOPED TO RESEARCH THE ISSUES ASSOCIATED WITH WHAT TOOLS MANAGERS NEED AND HOW DATA COULD BE BRIDGED BETWEEN MACHINES
- USES A PACKAGE CALLED "PROJECT SCHEDULER" AS ITS NUCLEUS FOR WORK STRUCTURING AND SCHEDULING
- ADDS VALUE BY INCORPORATING AN ADVANCED MANAGER/MACHINE INTERFACE, GRAPHICAL PERT, TIME MANAGEMENT FUNCTIONS AND IMPROVED REPORTING CAPABILITIES
- IS SCREEN-ORIENTED AND USES FUNCTION KEYS TO SIMPLIFY THE INTERFACE
- PRODUCES A VARIETY OF REPORTS INCLUDING:
  - ACTION ITEM LIST
  - ADDRESS BOOK
  - COST-TO-COMPLETE
  - CRITICAL PATH
  - KEY MEETING & BRIEFINGS
  - RESOURCE PROFILES
  - SCHEDULE-TO-COMPLETE
  - WORK BREAKDOWN STRUCTURE

PMW: FUNCTIONAL CAPABILITIES



- CREATE WORK BREAKDOWN STRUCTURE
- GANTT CHART FOR WBS ELEMENTS
- COST FOR WBS ELEMENTS

- NETWORK CHART

- GANTT CHARTS
- PROJECT COSTS
- COST-TO-COMPLETE

- WBS
- • •

- PAC II INTERFACE
- LOTUS "1-2-3" INTERFACE

- ADDRESS BOOK
- CALENDAR OF APPOINTMENTS
- MEETING/BRIEFING LIST
- ACTION ITEM LIST
- COMMANDS
- MODE ASSISTANCE
- TUTORIAL

PMW: LESSONS LEARNED I

- THE MANAGER/MACHINE INTERFACE MUST BE USER-FRIENDLY
  - PICTURE-ORIENTED, FUNCTION KEY DRIVEN AND MENU-BASED
- DON'T ASSUME MANAGERS KNOW HOW TO TYPE, USE A COMPUTER AND/OR WILL READ MANUALS
  - THE PACKAGE MUST BE EASY TO LEARN AND MUST HAVE BUILT-IN SAFEGUARDS AND "HELP"
- THERE IS NO WAY AROUND THE PROBLEMS OF INITIAL DATA ENTRY
  - MANAGERS DON'T HAVE THE TIME, DESIRE OR SKILL TO DO IT
  - SUBORDINATES DON'T HAVE THE KNOWLEDGE OR EXPERIENCE TO DO IT
  - THE TOOL MUST SUPPORT BOTH WORKING TOGETHER TO GET THE JOB DONE PERHAPS USING GAMES
- VENDORS DO NOT MECHANIZE ALL THE FEATURES/FUNCTIONS IN THEIR MANUALS

PMW: LESSONS LEARNED II

- VENDORS DO NOT MAKE IT EASY TO INTERFACE PACKAGES TO OTHER PACKAGES
  - FILE INTERCHANGE PERFORMANCE IS THE CRITICAL ISSUE BECAUSE USERS WILL NOT TOLERATE DELAYS IN GETTING RESPONSES TO THEIR QUESTIONS
- GLOBAL BRIDGING OR LINKING A MICRO-BASED TOOL TO A MAINFRAME-BASED SYSTEM REQUIRES LARGE EFFORTS
  - EVERY ORGANIZATION IS RUN DIFFERENTLY
- THE MOST USEFUL TOOLS ARE WORK PLANNING ORIENTED
- THE MOST USED TOOLS ARE TIME MANAGEMENT ORIENTED
- THE MOST WANTED TOOLS ARE "WHAT-IF" ORIENTED
- THE STATE-OF-THE-ART IS MIGRATING TO NETWORK-ORIENTED
- THE PMW-II SHOULD TAKE ADVANTAGE OF FACTS AND TRENDS

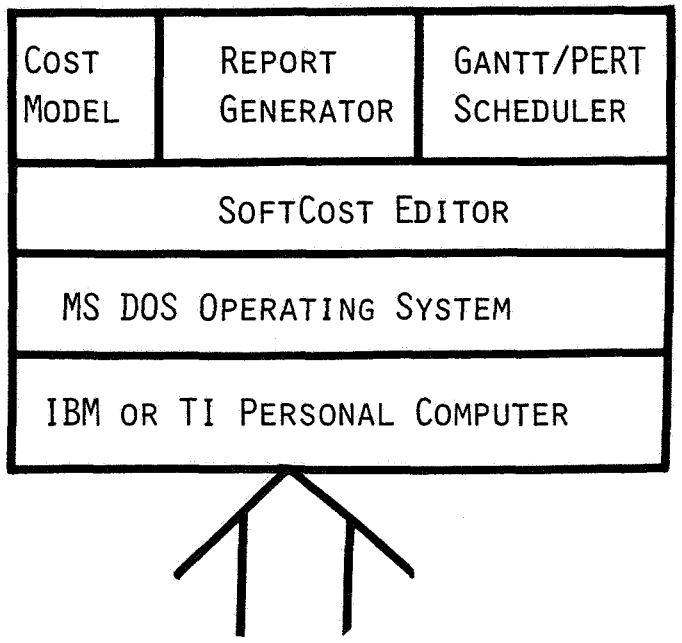
SOFTCOST-R: AN OVERVIEW

- DEVELOPED WITH USER-FRIENDLINESS AND ACCURACY IN MIND
- USES ABOUT 60 SIZING AND PRODUCTIVITY FACTORS TO PREDICT RESOURCES, DURATION AND STAFFING REQUIREMENTS
- COMPUTES CONFIDENCE FACTOR FOR DELIVERING ON-TIME AND WITHIN BUDGET FOR ANY PROPOSED EFFORT AND DURATION
- PROVIDES POWERFUL "WHAT-IF" ANALYSIS AND PLOTTING FEATURES
- GENERATES SCHEDULE AND RESOURCE ESTIMATES FOR ABOUT 50 TASKS MAKING UP A PROJECT
- PRODUCES A STANDARD WORK BREAKDOWN STRUCTURE FOR SOFTWARE DEVELOPMENT TASKS
- IS SCREEN-ORIENTED AND PERMITS ALL MODELS PARAMETERS TO BE CHANGED BY SIMPLE EDITING PROCESSES

SOFTCOST-R: OVERVIEW DIAGRAM

INPUTS

- MASTER MENU
- SIZING SCREEN
- COMPLEXITY FACTORS
- MANAGEMENT FACTORS
- STAFFING FACTORS
- ENVIRONMENTAL FACTORS
- HELP SCREENS



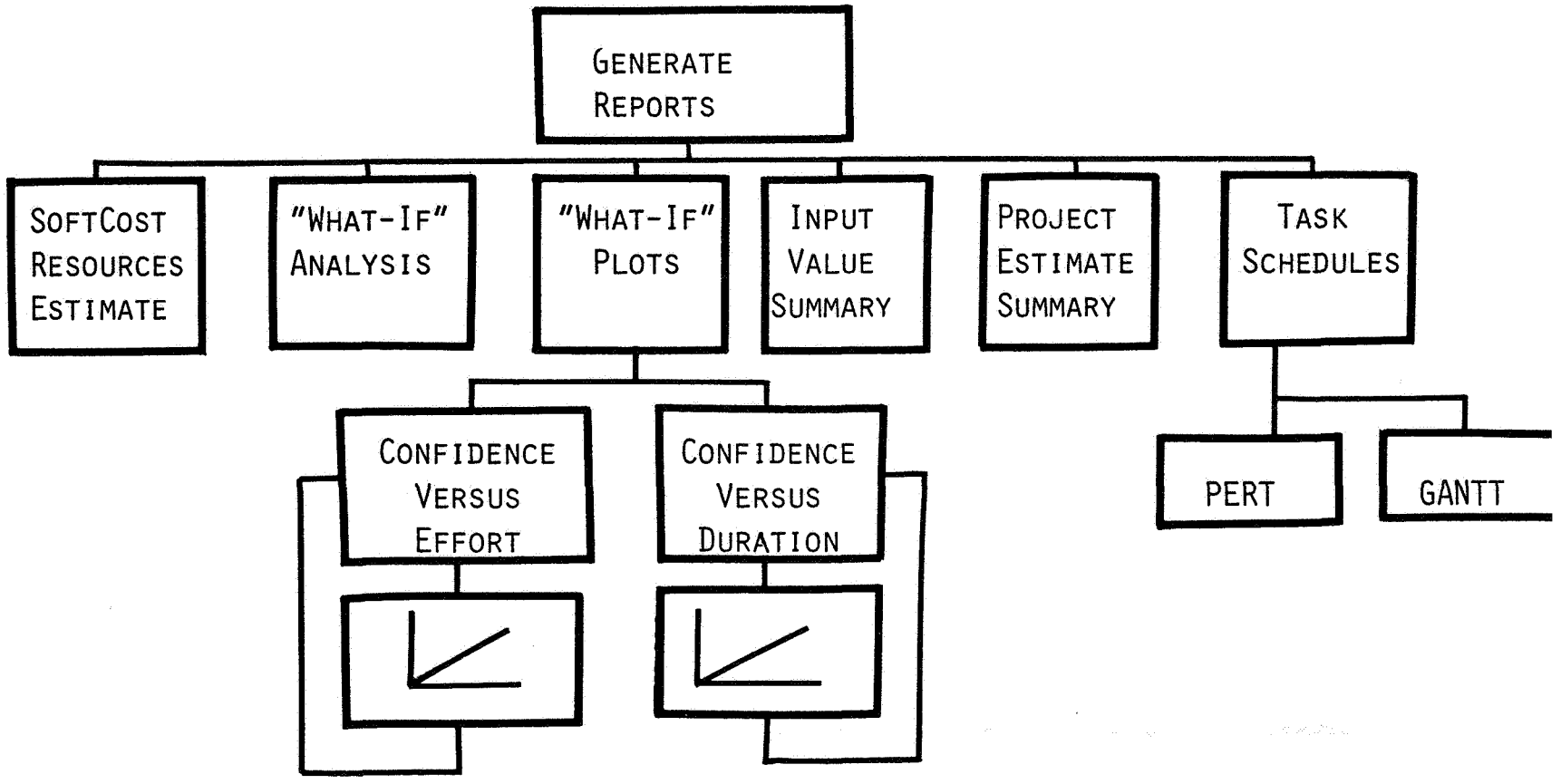
OUTPUTS

- SOFTWARE RESOURCES ESTIMATE
- "WHAT-IF" ANALYSIS REPORTS
- "WHAT-IF" PLOTS
- INPUT VALUE SUMMARY REPORT
- PROJECT ESTIMATE SUMMARY REPORT
- TASK SCHEDULES

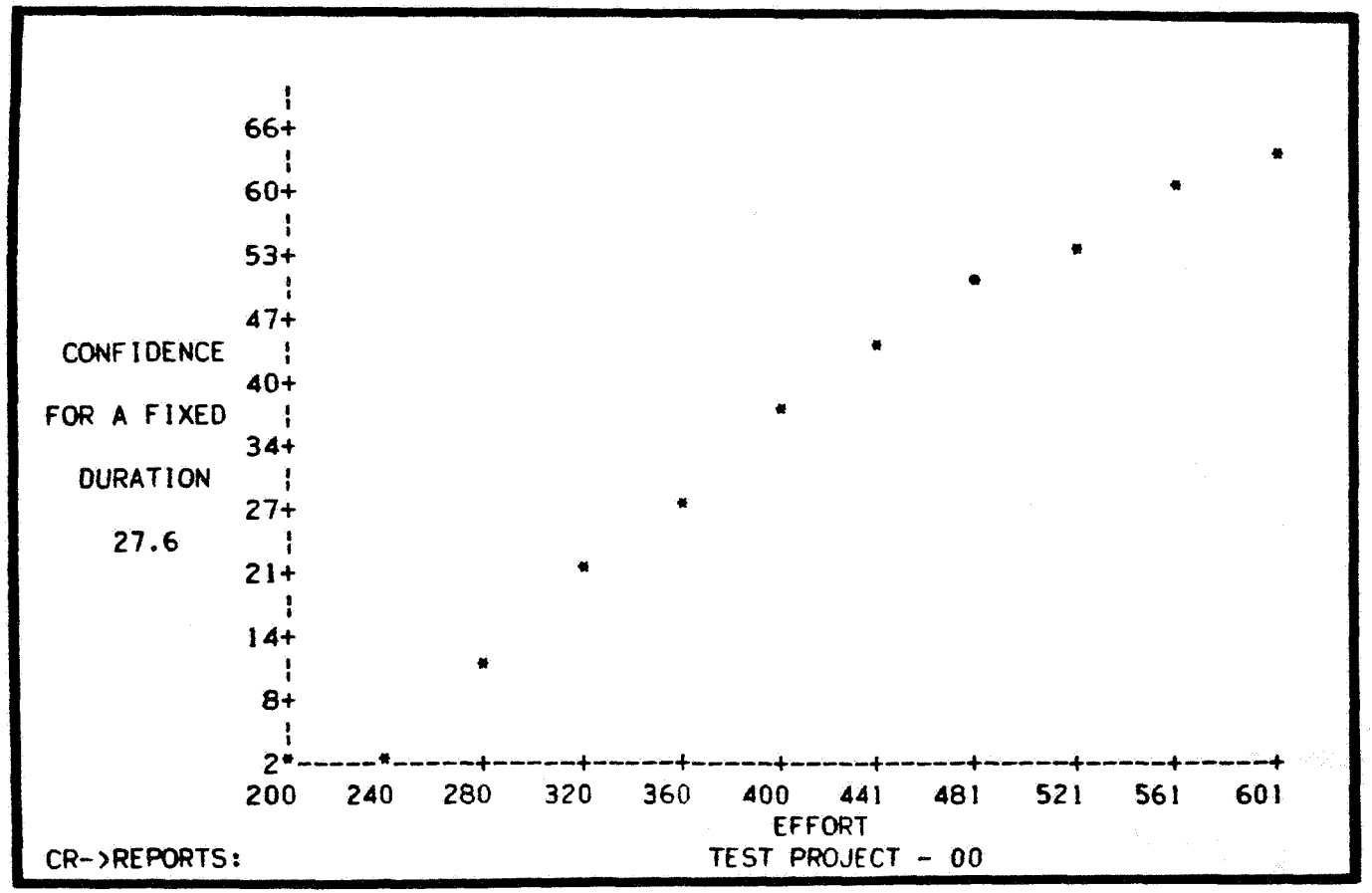
SOFTCOST-R IS EASY TO USE

- EASE OF USE AND LEARNING ARE PRIMARY DESIGN GOALS
- SOFTCOST-R IS MENU-ORIENTED USING FILL-IN-THE-BLANKS NOTATION AND FUNCTION KEYS
- SOFTCOST-R USES "HELP" LINES ON A SPLIT SCREEN TO LOCALIZE THE DEFINITION OF TERMINOLOGY AND ASSIST USERS IN INTERPRETATION
- SOFTCOST-R HAS BUILT-IN LIMIT CHECKS AND IS HARD TO MISUSE
- SOFTCOST-R IS DELIVERED WITH AN EXAMPLE CALLED "TEST" INTEGRATED INTO ITS USER'S MANUAL AND DELIVERED WITH THE PACKAGE
- SOFTCOST-R IS VALIDATED PRIOR TO SHIPMENT AND IS MAINTAINED UNDER STRICT LIBRARY CONTROL
- USERS HAVE BEEN ABLE TO GENERATE THEIR FIRST ESTIMATE WITHIN ONE HOUR OF GETTING ON THE SYSTEM

SOFTCOST-R: REPORT GENERATION FLOW



SOFTCOST-R: "WHAT IF" PLOTS



SOFTCOST-R: LESSONS LEARNED 1

- ORGANIZATIONAL PRECONDITIONING IS NEEDED
  - MOST ORGANIZATIONS DID NOT HAVE COST DATA AVAILABLE TO EITHER CALIBRATE THE MODEL OR VALIDATE ITS ACCURACY
  - EVEN IF THEY HAD DATA, IT WAS HARD TO MAKE SENSE OUT OF IT
  - FEW ORGANIZATIONS COLLECTED COST DATA AS A NORM AND FEW HAD AN ESTABLISHED FRAMEWORK IN PLACE FOR COST ESTIMATING
  - COST MODELS FORCED ORGANIZATIONS TO GATHER COST DATA AND ESTABLISH AN ESTIMATING FRAMEWORK
  - IN SOME CASES, COST MODELS FORCED CHANGES IN BUSINESS PRACTICES THAT SEEMED DISRUPTIVE BUT REALLY WERE NOT

- CALIBRATING THE MODEL TO THE ORGANIZATION IS THE HARD PART
  - MODELS MUST BE ARCHITECTED SO THEIR CALIBRATION POINTS AND SENSITIVITIES ARE KNOWN AND EASILY ALTERED
  - ORGANIZATIONS ARE DYNAMIC AND MODELS MUST REPLICATE THEM

- THE MOST POPULAR MODEL SEEMS TO BE COCOMO BECAUSE OF ITS AVAILABILITY
  - MOST USERS EMPLOY IT MANUALLY WITHOUT UNDERSTANDING ITS SCOPE OR LIMITATIONS

SOFTCOST-R: LESSONS LEARNED 2

- USER'S OFTEN PUT TOO MUCH RELIANCE ON MODELS
  - JUST BECAUSE A MODEL SAYS IT DOESN'T MAKE IT RIGHT
- THE WEAK LINK IN ALL COST MODELS IS SIZING
  - SOFTCOST'S SIZING FORMULAS ARE JUST AS SUSPECT AS OTHER MODELS
- USER'S DO NOT ALWAYS BELIEVE THE RESULTS
  - EVEN WHEN PERFECTLY CALIBRATED, USER'S DON'T WANT TO BELIEVE THE TRUTH (ESPECIALLY MANAGERS)
- MANY SIMPLE AND MUNDANE PACKAGING CONCEPTS CAN MAKE A MODEL USER FRIENDLY
  - TOO OFTEN, MODELS ARE DEVELOPED FOR CAPABILITY INSTEAD OF USABILITY
  - GOOD HUMAN ENGINEERING GOES A LONG WAY IN MAKING A PRODUCT ACCEPTABLE
- USER'S NEED HELP IN WORKLOADING (ALLOCATING WORKFORCE TO A SCHEDULE) AS MUCH AS THEY NEED HELP IN ESTIMATING
- MOST USER'S REALLY DO NOT KNOW HOW TO ALLOCATE THEIR EFFORTS OR TO ARCHITECT THEIR WORK PLANS
  - MODELING PACKAGES MUST BE DESIGNED TO HELP IN THIS AREA

TOOLS IN DEVELOPMENT

- ASSET IS A PC-BASED SIZING TOOL WHICH EXTENDS THE FUNCTION POINT WORK OF ALBRECHT AND GAFFNEY INTO THE REALMS OF SCIENTIFIC AND REAL-TIME SOFTWARE
  - STATISTICALLY-BASED AND EMPIRICALLY VALIDATED AGAINST A DATA BASE OF 15 PROJECTS

- PMW-II IS A PC/AT MANAGEMENT TOOL SYSTEM BEING DEVELOPED SPECIFICALLY FOR PROJECT MANAGERS
  - AI FRONTEND OPERATING UNDER GEM TO IMPROVE USER-FRIENDLINESS
  - INTEGRATES ASSET, SOFTCOST AND PROJECT WITH A SPREADSHEET FOR "WHAT-IF" ANALYSIS

- SOFTCOST-ADA IS A NEW VERSION OF SOFTCOST-R SPECIFICALLY CALIBRATED FOR ADA DEVELOPMENTS

ALL OF THESE EFFORTS BUILD ON OVER THREE YEARS OF USER EXPERIENCE

IN CONCLUSION

- I'VE DISCUSSED OUR EXPERIENCE WITH SOFTWARE MANAGEMENT TOOLS
- THE MAJOR LESSONS WE HAVE LEARNED INCLUDE:
  - PAY AS MUCH ATTENTION TO PACKAGING AS YOU DO TO FUNCTIONS OR FEATURES
  - MAKE YOUR SYSTEM "MANAGER-FRIENDLY" NOT "PROGRAMMER-FRIENDLY"
  - PROVIDE "WHAT-IF" CAPABILITIES AND A LOT OF SMALL USEFUL TOOLS
  - DON'T ASSUME VENDORS DELIVER WHAT'S ADVERTISED
  - WORRY ABOUT BRIDGING BETWEEN PACKAGES AND DON'T ASSUME IT IS EASILY DONE
  - REALIZE TOOLS MAY ACT AS THE CATALYST FOR ORGANIZATIONAL CHANGE

DEASEL : An Expert System for Software Engineering

by Jon D. Valett and Andrew Raskin

ABSTRACT

For the past ten years, the Software Engineering Laboratory [1] (SEL) has been collecting data on software projects carried out in the Systems Development Branch of the Flight Dynamics Division at NASA's Goddard Space Flight Center. Through a series of studies using this data, much knowledge has been gained on how software is developed within this environment. Two years ago work began on a software tool which would make this knowledge readily available to software managers. Ideally, the Dynamic Management Information Tool (DynaMITe) will aid managers in comparison across projects, prediction of a project's future, and assessment of a project's current state. This paper describes an effort to create the assessment portion of DynaMITe.

1.0 Backround

Assessing the state of a software project during development is a difficult problem, but its solution contributes to the success of the project. By determining a project's weaknesses early in its life cycle, problems can be dealt with quickly and effectively. For the software manager to perform this assessment he needs easy access to detailed, accurate information (knowledge) regarding past projects within the development environment. He then incorporates this information with his own knowledge of software engineering to make some assessment of a project's strengthes and weaknesses. The DynaMITe Expert Advisor for the SEL (DEASEL) is the first version of an expert system that attempts to simulate this process.

2.0 Developing and Using Rules

Basically, DEASEL assesses an ongoing project by attempting to answer a simple question such as "How is my project doing?" To answer this question DEASEL utilizes a knowledge base of rules for evaluating software projects. This knowledge base consists of rules derived from two sources: the SEL database and experienced software managers. DEASEL uses these rules along with data on the project of interest, to give the manager a relative rating of the quality of that project.

2.1 Corporate Memory

Of course, a major effort in the development of the DEASEL system was the actual collection of knowledge. To derive rules from the corporate memory, former studies [2,3,4,5,6,7,8] performed by the SEL were reviewed to find relationships that affect the quality of a software project. That is, many studies of data concerning the SEL environment have been done within the last ten years. These studies give some idea of the cause and effect of technologies and methodologies on a software project. Thus, relationships like "increasing tool use will increase productivity" are found. Because of the interdependencies amoung the items the strength of each relationship is then determined. For example, many different factors may influence productivity, therefore the determination of which of these have the most and which the least influence must be made. This has been a long and difficult process because of the amount of data and the problems with determining what data is relevant to the assessment process.

2.2 Knowledge from Software Managers

The other source of knowledge is the experienced software managers, who have certain "rules of thumb" they use to evaluate a software project. They are questioned to obtain this subjective information which is then used along with the more objective material to produce the knowledge base. Again the determination of the strengthes of the relationships must be performed. The entire process of collecting knowledge is long and difficult and has only just begun for the DEASEL project.

2.3 Representing the Rules

After collecting a preliminary set of knowledge, thought began on how to actually represent this knowledge. The initial work on knowledge representation for DEASEL was directed at using standard expert system techniques, including if-then production rules. But soon the discovery was made that knowledge regarding the assessment of a software project's development is more naturally represented in a different manner. In fact, the overall conclusion drawn from an assessment is quite different from that drawn by a traditional expert system. The difference lies in the type of question answered by DEASEL. The traditional medical expert system, such as the often cited MYCIN [9], answers a question like "What disease does patient X have?" Then, given some data on the patient the system determines the disease. DEASEL, on the other hand, must answer the question "How is project X doing?" Thus, it must give a rating to the system based on the facts given to it. The analagous question in the medical domain would be "How is patient X's health?"

In order for DEASEL to answer the question "How is project X doing?", it needs two different types of knowledge. The first type of knowledge is the assertions which relate to the specific

project in question. This includes the facts known about the project as it currently stands. The second type of knowledge is the detailed representation of how different facts affect the overall development process of a project. These are the more general "rules" on what affects the quality of a software project. These rules are set up based on the knowledge described earlier from the data base and the software manager. They are used to describe all of the factors which affect a software project's quality and all the sub-factors that affect those factors, etc. For this reason this system of knowledge representation, which is unique to DEASEL, is called factor-based. Each rule in the factor-based representation scheme specifies a system and its factors (sub-systems) and the weight (strength of the relationship) each factor has on the system. Thus, between the specific assertions about the project and the general rules concerning software development within the SEL environment DEASEL can rate a project.

2.4 An Example Rule

To explain how this rating process works, here is an example rule from DEASEL's knowledge base:

```
The factors that affect Computer_Environment_Stability are
        1) Operating_System_Stability        .3
        2) Software_Tool_Stabiltiy           .2
        3) Hardware_Stability                .4
        4) Computer_Env_Proc_Stability       .1
```

The number associated with each factor is a weight, and the sum of the weights must always total one. This rule states that the four listed factors have an affect on the quality of the Computer_Environment_Stability. The rule's weights indicate that Hardware_Stability is the most important factor in the assessment of Computer_Environment_Stability, while Computer_Env_Proc_Stability is the least important factor. DEASEL uses the ratings of all four factors to determine a rating for Computer_Environment_Stability.

2.5 Deriving Conclusions

DEASEL's overall assessment process consists of trying to assign a rating to each of the quality indicators specified via the knowledge base. Obviously just answering the question "How is project X doing?" will not give the manager specific enough information about his project. Therefore, the knowledge base specifies the top level factors DEASEL should rate. Currently, the knowledge base has four such quality indicators: reliability, predictability, stability, and controlled development. Thus DEASEL actually gives information (a rating) on each of these four indicators which gives the manager an assessment of how his project is doing in these areas. In order

to rate these four factors DEASEL must find the rules which relate to these factors and assign a rating to these rules. That is, DEASEL reaches a conclusion on what it believes is the rating of these indicators. For DEASEL to do this it must first reach the conclusions on the factors which affect these indicators. Of course, these factors may have rules which specify their assessment, so this process continues until all of the necessary conclusions are reached.

DEASEL reaches conclusions in one of three ways:

1) The conclusion can be an assertion from the knowledge base.
2) DEASEL can infer the conclusion based on other conclusions and its rule base.
3) If both 1) and 2) fail, it can ask the user to supply the conclusion.

The three types of conclusions combine to allow DEASEL to make its assessment of the supplied quality indicators. The basic process is to first find a rule for one of the quality indicators then to resolve all of the conclusions necessary to reach a conclusion for that indicator. This process continues by reaching conclusions in each of the three ways, until all the conclusions are resolved.

To fully understand the rating process one must also understand how these conclusions are reached. A conclusion is reached when a rating has been assigned to a factor in the knowledge base. A rating is defined as a number between zero and one, the higher the rating the better the factor's quality. A rating of .5 would be average or normal. Note that the ratings always indicate quality, for example a rating of .7 for error rate as a factor would indicate a lower than normal error rate. In addition, every conclusion has an associated certainty. A certainty is the probability that the conclusion's rating is correct within some fixed delta. Currently, DEASEL sets delta at 0.1.

All three types of conclusions have both a rating and a certainty. Type 1 conclusions are really the assertions described earlier. Currently, the asssertions are entered by hand into the knowledge base. In the future this process will be automated and will be done by the DynaMITe tool, via the SEL data base. The certainties for these conclusions are generally very high (around .9) because the ratings are basically comparisons between real data and average or normal numbers. Conclusions of type 2 are computed using the following formulae:

$$\text{Rating} = \sum_{i=1}^{N} (\text{Rating of factor}(i) \times \text{Weight of factor}(i))$$

$$\text{Certainty} = \sum_{i=1}^{N} (\text{Certainty factor}(i) \times \text{Weight of factor}(i))$$

where n is the number of factors in the rule

Thus, a rule for a certain factor is given a conclusion by using these formulae to calculate its rating and certainty. The schema used here should look familiar to anyone with knowledge of

probability. In its typical application, however, each of the factors in the system being rated must be independent. In the complex and unfamiliar domain of software engineering, such an assumption may be incorrect. Our computations could therefore be slightly or grossly in error depending on how much the knowledge base violates this constraint. Future DEASEL knowledge engineers must keep this in mind when creating and modifying the rule base. Type 3 conclusions are necesssary when the system cannot use type 1 or type 2 conclusions. In order for the system to complete an assessment it must have conclusions for all the factors in the knowledge base. Since expert systems must deal with incomplete knowledge, whenever DEASEL cannot reach a conclusion for a factor it assumes a normal rating (.5) with a certainty of .2. Note that the .2 is the probability that the rating will be correct within + or - delta, which in effect makes for a meaningless conclusion. Whenever DEASEL is forced to do this, it makes a note to ask the user if the conclusion can be provided. Thus, the user can later provide the answers to questions about the incomplete knowledge. Once these questions are answered, DEASEL gives the rating supplied by the user a certainty of 1.0.

2.6 Current DEASEL Capabilities

The capabilities of the current DEASEL system include allowing the user to obtain an assessment of his project, if some assertions exist for that project. After the initial assessment is given the user has three options 1) asking for an explanantion, 2) answering questions about his project, and 3) playing what-if games. For any conclusion, the user can ask for an expalnantion of how the conclusion was reached. The explanation consists of the conclusions DEASEL reached about the factors of the original conclusion. That is, the user is able to ask DEASEL what caused it to reach any specific rating for any factor. This process can continue as the user asks for explanations of the factors previously reported on, and so on. Earlier we mentioned that DEASEL makes a note of type 3 conclusions. The user may opt to answer these questions as he wishes. He may also respond to the questions by indicating he does not know the answer. In this case, DEASEL maintains the meaningless conclusion reached earlier. Answering questions is encouraged because it leads to more certain conclusions. What-if games aid the manager in evaluating the effects of changes he may wish to make in his project. This process allows the user to enter controls into the system, by actually changing conclusions. That is, the user can see what will happen if he changes certain conclusions in the knowledge base. After changing one or more conclusions he can then reassess the project, to determine the affects of these changes. This is an important feature of the DEASEL system, because it allows the manager to determine how he might be able to improve his software project.

3.0 Summary

Although the current version of DEASEL does begin to attack the problem of project assessment, much more work is needed to make the system a useful tool. Three potential directions exist for future work: adding to and verifying the rule base, verifying the accuracy of the assessment process, and automating the creation of the assertion portion of the rule base. All of these areas will require time and effort to complete, but are necessary for successfully determining the validity of this project. Obviously, DEASEL is but an initial attempt at solving the problem of automating the process of assessing the state of an ongoing software project. DEASEL has, however, given some insight into the problem and ways to solve it. Hopefully this initial work will lead to techniques for solving the problem more completely.

REFERENCES

1. SEL-81-104, The Software Engineering Laboratory, D.N. Card, F.E. McGarry, G. Page, et al., February 1982

2. SEL-83-002, Measures and Metrics for Software Development, D.N. Card, F.E. McGarry, G. Page, et al., March 1984

3. SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freuberger and V.R. Basili, May 1979

4. McGarry, F.E., Valett, J., and Hall, D,. Measuring the Impact of Computer Resource Quality on the Software Development Process and Product, Proceedings of the Hawaiian International Conference on Systems Sciences, January 1985

5. SEL-85-001, Comparison of Software Verification Techniques, D. Card, R. Selby, F.E. McGarry, et al., April 1985

6. SEL-82-004, Collected Software Engineering Papers: Vol I, July 1982

7. SEL-83-003, Collected Software Engineering Papers: Vol II, November 1983

8. SEL-85-003, Collected Software Engineering Papers: Vol III, November 1985

9. Shortliffe, E.H., Computer-Based Medical Consultations: Mycin, Elsevier, North Holland, New York, 1986

THE VIEWGRAPH MATERIALS

for the

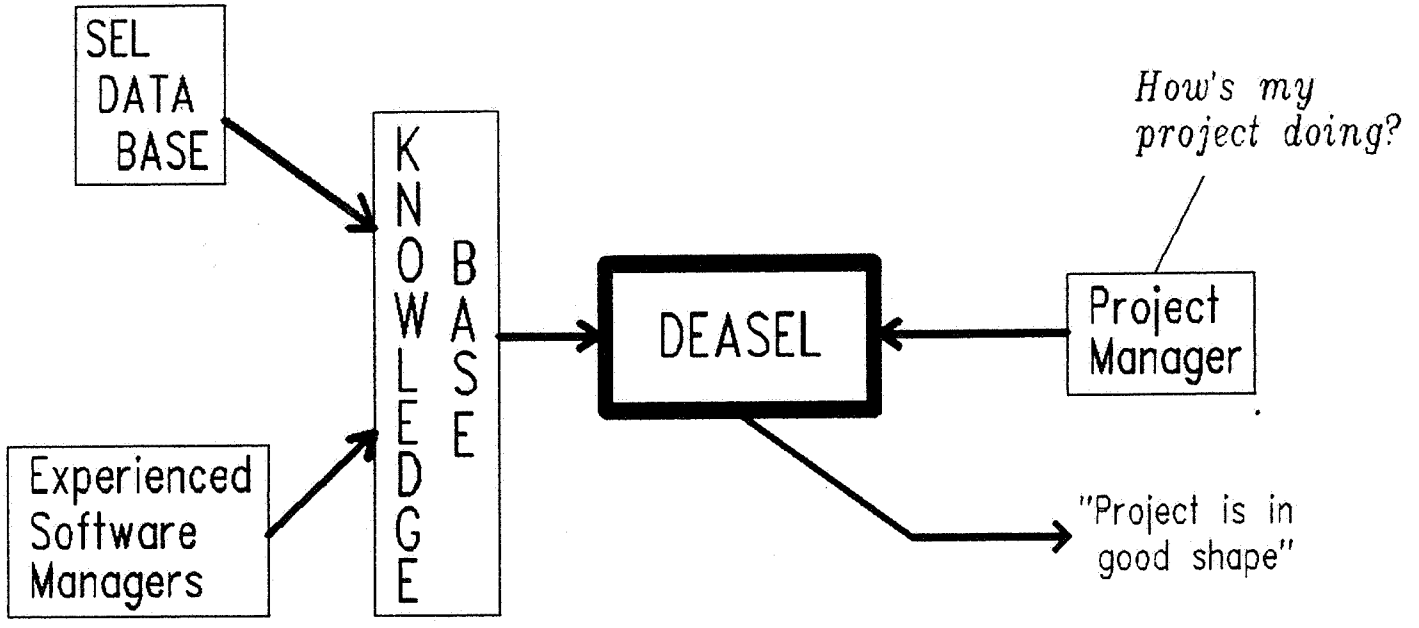
J. VALETT PRESENTATION FOLLOW

DEASEL : An Expert System for Software Engineering

Jon Valett
and
Andrew Raskin

NASA / GSFC

DEASEL



DEASEL is an experimental system integrating corporate memory and the knowledge of software managers to automatically analyze and assess a current software project.

KEY ISSUES

1. *Can information relevant to assessing a software project be extracted?*

2. *Can we easily represent this information?*

3. *Can this information be integrated into a useful software tool?*

COLLECTING KNOWLEDGE

From Corporate Memory

- Look for relationships

  eg.

  increased tool use $\longrightarrow$
  increased productivity

- Strength of relationships
  (weights)

- "A lot" of knowledge

From Software Manager

- Subjective information

- Strength of relationships
  (weights)

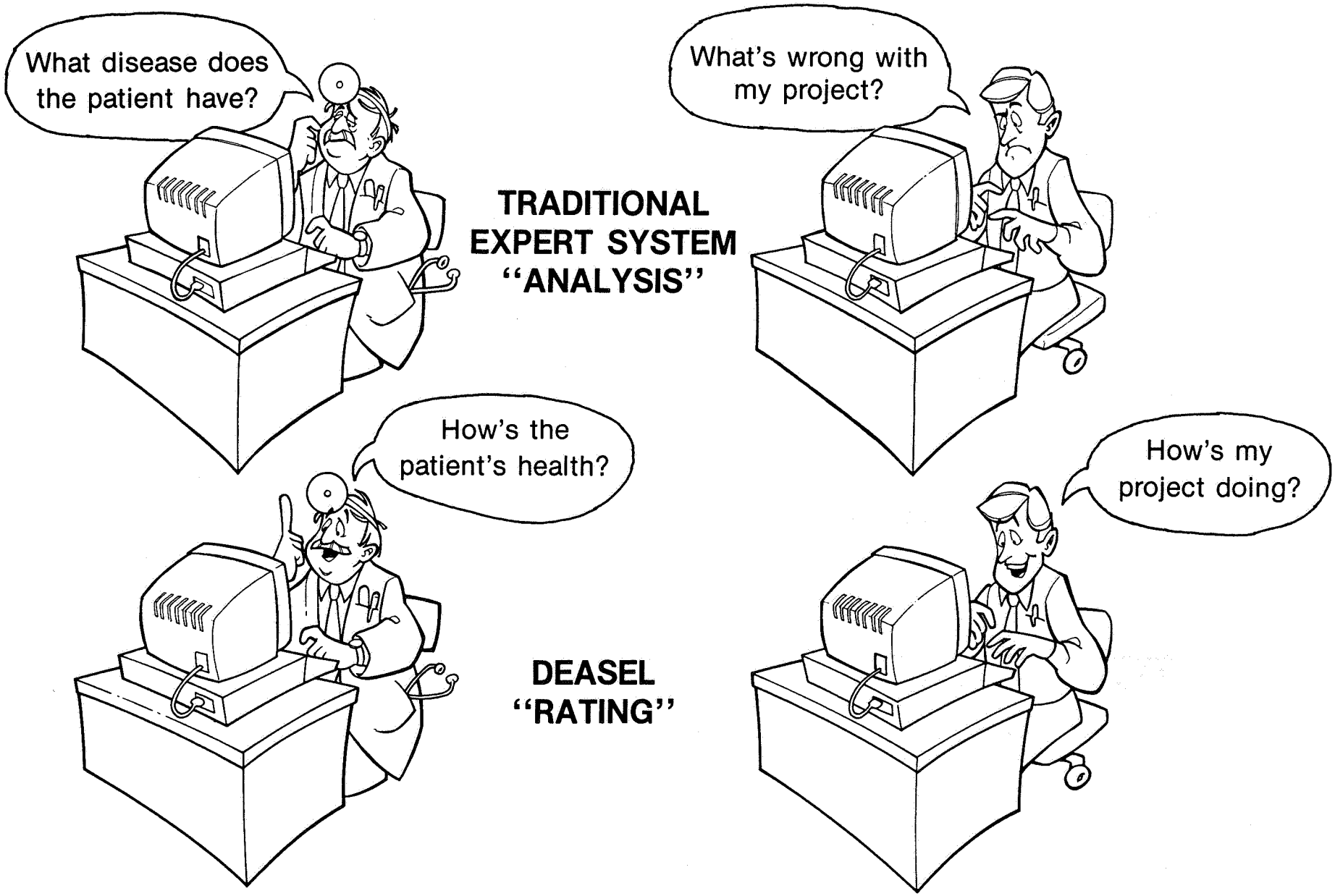
  eg.

  stability of a project affected by
  - no. of spec. changes minor importance
  - staffing stability very important
  - no. of design changes important

THE QUESTION

THE MEDICAL DOMAIN

THE SOFTWARE ENGINEERING DOMAIN



J. Valett
NASA/GSFC

TO ANSWER THE QUESTION . . .

Use

ASSERTIONS about the specific project

eg.

Change rate of code is above normal

Number of design changes is normal

And

RULES on software management

eg.

The main factors that influence reliability are

1. Change rate of code
2. Design stability

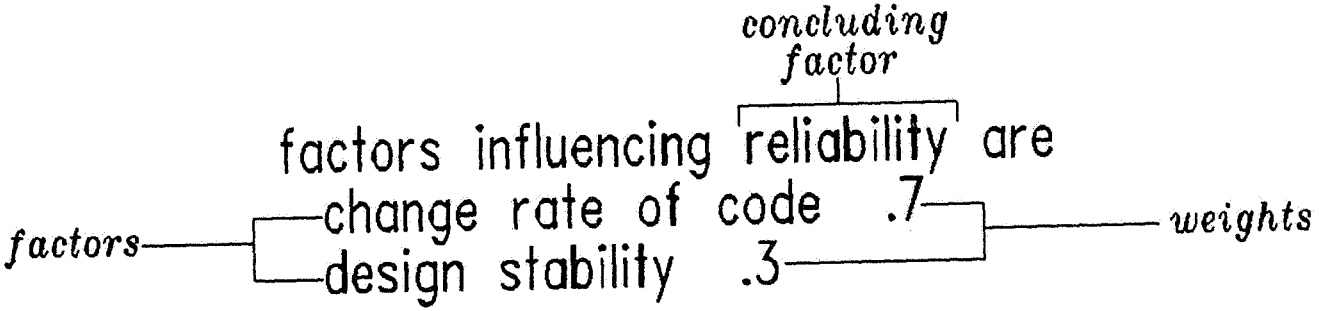
To

ASSESS the project on

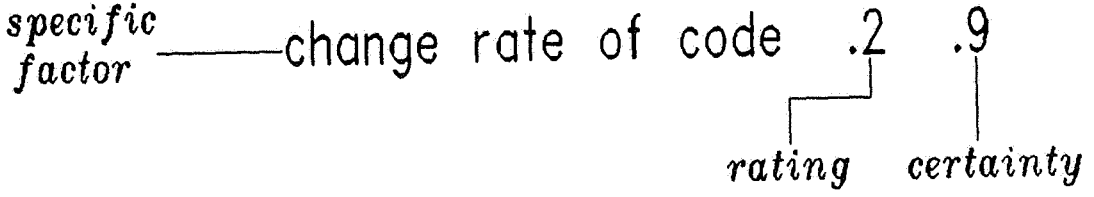
- Reliability
- Predictability
- Stability
- Controlled Development

THE KNOWLEDGE BASE

Rules – general



Assertions – specific project



THE RATING PROCESS

A Simple Example

Factors of Reliability are

Change Rate of Code .7
Design Stability .3

Key

RULES
ASSERTIONS
CALCULATIONS

THE RATING PROCESS

A Simple Example

Factors of Reliability are

Change Rate of Code .7 .14 .63 ←—— Change Rate of Code .2 .9
Design Stability .3

Factors of Design Stability are

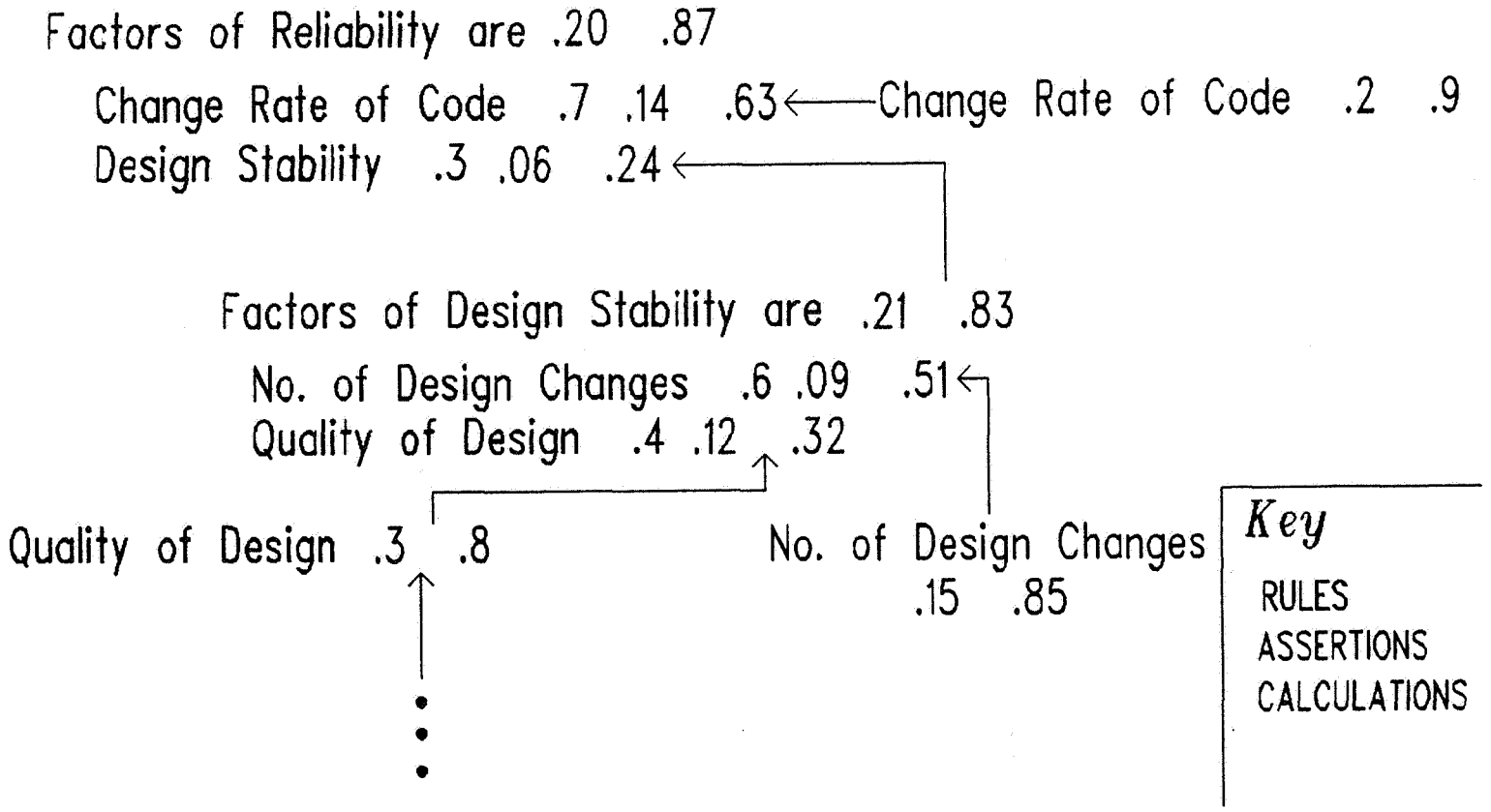
No. of Design Changes .6
Quality of Design .4

Key

RULES
ASSERTIONS
CALCULATIONS

THE RATING PROCESS

A Simple Example



CURRENT SYSTEM CAPABILITIES

- About 25 rules on code and unit test

Assess a Project *"How's my project doing?"* Reliability is Below Normal with High certainty	Explain a Rating *"Why is my reliability below normal?"* Change Rate is Very High and Design Stability is Normal
Answer Questions What is the quality of your project's design? *"Excellent"*	Play "What-if" Games *"What if my change rate was very low?"* Reliability would then be rated normal

PLANS

- *Add rules from other phases*
- *Validate existing rules*
- *Validate current assessment process*
- *Automate generation of assertions*

KEY ISSUES

1. *Can information relevant to assessing a software project be extracted?*

   - Yes, but is difficult and time consuming

2. *Can we easily represent this information?*

   - We think so, and hope our strategy is effective

3. *Can this information be integrated into a useful software tool?*

   - It can be integrated, but too early to determine usefulness

AN EXPERIMENTAL EVALUATION OF ERROR SEEDING AS A PROGRAM VALIDATION TECHNIQUE

John C. Knight Paul E. Ammann
Department of Computer Science
University of Virginia
Charlottesville, Virginia.

A Summary

Submitted To The Tenth Annual Software Engineering Workshop
Goddard Space Flight Center
Greenbelt, Maryland.

The *error seeding* technique was originally proposed by Mills [1] as a method for determining when a program has been adequately tested using functional or random testing. The procedure resulted from a desire to apply statistical methods to the problem of predicting the number of errors in a program in the hope that the number of errors discovered during testing could be used to estimate the number of remaining undetected errors. The method involves deliberately introducing or *seeding* artificial errors into a program and subsequently testing that program.

Error seeding has the desirable property that it is apparently simple to employ and it provides a stopping condition for testing. Unfortunately, it has the major drawback that, in order to work effectively and for the existing statistical model to apply, it relies upon the following three assumptions:

(1) Indigenous errors, those introduced by the programmer, are all approximately equally difficult to locate.

(2) Seeded errors are approximately as difficult to locate as indigenous errors.

(3) Errors, whether indigenous or seeded, do not interfere with one another.

A priori there is no reason to believe that any of these assumptions hold. The first and third seem reasonable. However, error seeding has been criticized on the basis of the second assumption. It seems unlikely that realistic seeded errors can be generated but no definitive, empirical evidence for any of the assumptions has been gathered previously. We have performed an experiment designed to check the validity of each of the underlying assumptions. In particular, we were interested in evaluating very simple, syntax-based algorithms for generating seeded errors.

Briefly, as part of a separate experiment [2, 3], twenty-seven Pascal programs have been written independently by *different* programmers to a single specification. Thus all twenty-seven are intended to perform the same function, the processing of radar data in a simple antimissile system. As part of the other experiment, the programs have been subjected to one million tests, and a great deal is known about the indigenous errors present in the programs. These programs represent an excellent starting point for an experiment with error seeding. Any results obtained can be averaged thereby eliminating any bias attributable to individual programmers.

In the error seeding experiment, seventeen of the twenty-seven programs were selected at random, errors were seeded into all seventeen, and the resulting programs were tested. The algorithms used for seeding errors were very simple: two algorithms modified the bounds on **for** statements, three algorithms modified the Boolean expression in **if** statments, and one algorithm deleted assignment statements. Each of these algorithms was applied four times to each of the 17 programs for a total of 408 modified programs, each of which contained one seeded error. The programs were tested using 25,000 of the 1,000,000 test cases from the previous experiment.

The metric used for evaluating the seeded errors was the mean time to failure (MTF). The MTF for a particular program containing a seeded error is defined as the average number of test cases executed between detected failures. The MTF's for the seeded errors had a wide range. Some seeded errors caused a failure on every test case; some had a very small number of failures in 25,000 test cases; and others caused *no failures at all* in 25,000 test cases. We conclude that it *is* possible to generate seeded errors that are arbitrarily difficult to locate, albeit at the expense of creating others that are easy to locate. These results suggest, surprisingly, that it is possible to comply with the second assumption listed above.

An examination of the MTF's of the *indigenous* errors revealed a similar wide range of failure rates. In fact, there was a very strong resemblance in mean time to failure between the resilient seeded errors and the indigenous errors. However, in neither case were errors equally likely to be discovered, in conflict with the first assumption cited above.

Finally it was discovered during the experiment that in two cases a seeded error corrected, or partially corrected, an indigenous error. Clearly, the implication is that assumption three above was violated. We conclude that the first and third assumptions, those that seem most believable, are in fact violated, and that the second, the one that seems totally unreasonable, can be complied with. Using the data from this experiment, the underlying model of error seeding can be modified and error seeding made a useful, practical technique.

REFERENCES

(1) Mills, H.D., "On The Statistical Validation of Computer Programs", in *Software Productivity*, Little Brown, Toronto.

(2) Knight, J.C., and N.G. Leveson, "A Large-Scale Experiment In N-Version Programming", Proceedings of the *Ninth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, November 1984, Greenbelt, MD.

(3) Knight J.C., and N.G. Leveson, "A Large Scale Experiment In N-Version Programming" Digest of Papers FTCS-15: *Fifteenth Annual Symposium on Fault-Tolerant Computing*, June 1985, Ann Arbor, MI.

Quality Assurance Software Inspections at NASA Ames
Metrics for Feedback and Modification

Greg Wenneson, Informatics General Corporation

Software Inspections are a set of formal technical review procedures held at selected key points during software development for the purpose of finding defects in software documents. Inspections are a Quality Assurance tool and a Management tool. Their primary purposes are to improve overall software system quality while reducing lifecycle costs and to improve management control over the software development cycle. The Inspections process can be customized to specific project and development type requirements and are specialized for each stage of the development cycle.
For each type of Inspection, materials to be inspected are prepared to predefined levels. The Inspection team follows defined roles and procedures and uses a specialized checklist of common problems in reviewing the materials. The materials and results from the Inspection have to meet explicit completion criteria before the Inspection is finished and the next stage of development proceeds. Statistics, primarily time and error data, from each Inspection are captured and maintained in a historical database. These statistics provide feedback and feedforward to the developer and manager and longer term feedback for modification and control of the development process for most effective application of design and quality assurance efforts.

HISTORY

Software Inspections were developed in the early mid-1970s at IBM by Dr. Mike Fagan, who was subsequently named software innovator of the year. Fagan also credits IBM members O.R.Kohli, R.A.Radice and R.R.Larson for their contributions to the development of Inspections. In the IBM Systems Journal [1], Fagan described Inspections and reported that in controlled experiments at IBM with equivalent systems software development efforts, significant gains in software quality and a 23% gain in development productivity were made by using Inspections based reviews at the end of design and end of coding (clean compile) rather than structured walkthroughs at the same points. Fagan reported that the Inspections caught 82% of development cycle errors before unit test, and that the inspected software had 38% fewer errors from unit test through seven months of system testing compared to the walkthrough sample with equivalent testing. Fagan also cites an applications software example where a 25% productivity gain was made through the introduction of design and code inspections. As further guidelines for using Inspections, IBM published an Installation Management Manual [2] with detailed instructions and guidelines for implementing Inspections.

Inspections were introduced to NASA/Ames Research Center in 1979 by Informatics General Corporation on the Standardized Wind Tunnel System (SWTS) and other pilot projects. The methods described by IBM were adapted to meet the less repetitious character of Ames applications and research/development software as compared to that of IBM's systems software development. Though not able to duplicate IBM's controlled environments and experiments, our experience at Ames of gains in quality and productivity through using Inspections have been similar. From a developed Wind Tunnel software application which had been reviewed in structured walkthroughs and then later was rewritten and reviewed using

Inspections, the Inspected version had 35-65% less debug and test time and about 40% fewer post-release problems. Inspections implemented prior to unit test have been shown to detect over 90% of software's lifetime problems. Inspection results have been sufficiently productive in terms of increased software quality, decreased development times, and management visibility into development progress, that Inspections have been integrated into Informatics' development methodology as the primary Quality Assurance defect removal method.

When Inspections were first implemented at Ames, only design and code Inspections were introduced. The scope and usage has expanded so that currently, Inspections are used to review both system level and component level Goals (requirements) Specifications, Preliminary Design, Detailed Design, Code, Test Plans, Test Cases, and modifications to existing software. Inspections are used on most Informatics staffed development tasks where the staff level and environment are appropriate. Inspections implementation and usage at Ames are described in NASA Contractor Report 166521 [3]. Within Informatics contracts outside of the Ames projects, Inspections are also used to review Phase Zero (initial survey and inventory of project status), Project Goals, and Requirements Specifications generated through structured analysis.

PARTICIPANTS

The Inspectors operate as a team and fill five different types of roles. The Author(s) is the primary designer, developer, or programmer who prepares the materials to be inspected. The author is a passive Inspector, answering questions or providing clarification as necessary. The Moderator directs the flow of the meetings, limiting discussion to finding errors and focusing the sessions to the subject. The moderator also records the problems uncovered during the meetings. A Reader paraphrases the materials, to provide a translation of the materials different from the authors' viewpoint. One or more additional Inspectors complete the active components of the team. A limited number of Observers, who are silent non-participants, may also attend for educational or familiarizing purposes. Of the team members, the moderator and a reader are the absolute minimum necessary to hold an Inspection.

Team composition and size are important. Composition using knowledgeable designers and implementors having similar background or from interfacing software enable cross training of group members; understanding is enhanced and startup time is lessened. However, team members must be sufficiently different so that alternate viewpoints are present. Fagan recommends a four member team composed of a moderator and the software's designer, implementor, and tester. Our experience is that the most effective team size seems to be three to five members, exclusive of author and observers; more than this is a committee, less may not have critical mass for the process. We also try to keep the team together for all of the software's Inspections.

TOOLS

Written tools are used by the participants during the Inspections process to assist in the preparation, the actual sessions, and the completion of the Inspection. Standards are necessary as guidelines for preparing both design and coding products. The Entrance Criteria for inspection materials define what materials are to be inspected at each type of Inspection, the level of detail of preparation, and other prerequisites for an Inspection to occur. Checklists of categories (Data Area Usage, External Linkages, etc.) of various types of problems to look for are used during the sessions to help locate errors and focus attention on areas of project

concern. The Checklists are also used by the author during his preparation of materials and by the inspectors while they are studying the materials. Exit Criteria define what must be done before the Inspection is declared complete and the materials can proceed to the next stage of development. Each of these tools will have been customized for each projects type of development work, language, review requirements, and emphasis that will be placed on each stage of the development process.

PROCEDURES

An Inspection is a multi-step sequential process. Prior to the Inspection, the Author prepares the materials to the level specified in the Entrance Criteria (and to guidelines detailed in the project development or coding standards). The moderator examines the materials and, if they are adequately prepared, selects team members and schedules the Inspection. (IBM lists these preparations as the Planning step.) The Inspection begins with a short educational Overview session of the materials presented by the author to the team. Between the overview and the first Inspection session, Preparation of each Inspector by studying the materials occurs outside of the meetings. In the actual Inspection sessions, the Reader paraphrases while the Inspectors review the materials for defects; the Moderator directs the flow of the meetings, ensures the team sticks only to problem finding, and records problems on a Problem Report form along with the problem location. Checklists of frequent types of problems for the type of software and type of Inspection are used during the preparation and Inspections sessions as a reminder to look for significant or critical problem areas. After the Inspection sessions, the moderator labels errors as major or minor, tabulates the Inspection time and error statistics, groups major errors by type, estimates the rework time, prepares the summaries, and gives the error list to the author. The author Reworks the materials to correct problems on the problem list. Follow-up by the moderator (or re-inspection, if necessary) of the problems ensures that all problems have been resolved.

In certain cases, a desk Inspection or "desk check" may be a more effective use of time than a full Inspection. Desk Inspections differ from normal Inspections in that during the preparation period each inspector individually records errors found and a single Inspection session is held to resolve ambiguities in the problems. The moderator compiles all collected error reports to produce a single report. All other Inspection steps proceed normally. Desk Inspections can be appropriate for code or design that the team is familiar with and that has already been through previous Inspections. Desk Inspections do not have the group synergy generated during "normal" Inspections. The SWTS Inspections database for FORTRAN code Inspections indicates that the desk check has an 80% error detection rate but only takes 40% of the time required of a full Inspection.

STATISTICS

The statistics captured from the Inspection and tabulated by the moderator consist of time and error values. The time statistics are average per person preparation time (excluding the author) and Inspections sessions meeting time, both normalized to a thousand lines of code (KLOC). The error statistics are the numbers of major and minor errors detected, also normalized to a KLOC. As part of the tabulating and summarizing process, error distributions of major errors by Checklist headings are recorded and summarized for the Inspection as a whole. The tabulated statistics are entered into a database as weighted averages by size in lines of design or code and keyed by expected implementation language and type of Inspection. The SWTS Inspections database currently contains almost 250 entries of data for FORTRAN and Assembler languages for the Goals (Functional Requirements), Preliminary

Design, Detailed Design, and Code (desk and non-desk check) types of Inspections held on developed Wind Tunnel System software from 1980 through 1985. Over half of the entries are for code Inspections. Figure 1 contains summary figures from the database. The database summaries provide guidelines from which general conclusions and assumptions can be drawn. The database was generated as a development and management tool from several related SWTS project's Inspections and not from tightly controlled experiments. As such, when comparing individual Inspections figures to the database figures, variances from one-half to twice the average amounts summarized from the database are not considered extraordinary.

STATISTICS USE

The Inspections statistics in their raw and weighted forms can be used by the author, the design team and manager, the project manager, and Software Engineering as feedback, feedforward, and control mechanisms for individual, team, project and Inspections process behavior modification for future work to achieve better results. In addition, the statistics can be used in the current project and for future work and projects for tracking, estimating, planning, and scheduling of development and QA work.

The author uses the statistics to determine immediately what is deficient in inspected design or code and, over the longer term, patterns and general problem areas on which to focus attention for future work. The problem list, besides providing a working list of detected problems, includes locations of what needs to be fixed before the next development stage can proceed. Additionally, a distribution of major errors by checklist category across each module provides warning signals of error prone modules and high or higher density error rates by error type. A history of high error rates of certain error types also provides a pointer to design areas which need more work or training to develop or better understand.

The programming team and manager use error distribution by type and module from individual Inspections and Inspections of related software to locate common problem areas and thus focus future work and communication to diminish these. Error rates higher than normal for the group as a whole or error distributions in particular areas may indicate a group misunderstanding or a misstatement of the requirements. Higher error densities in modules interfacing to existing (or new) software, for example, can alert and direct effort to understanding the interface or provide warning to another group to clarify or improve that interface. For the designer and the team manager, lines of design (or lines of code, depending on development stage) and complexity per module give immediate feedback for design considerations of module size, cohesion, and coupling; this additionally provides an opportunity to ensure that modules are not proliferating from one design stage to the next. The completion of any individual Inspection along with module quantity and sizing gives quantitative and qualitative feedback for validity of component estimating, scheduling, and tracking information.

The Project Manager utilizes the statistics to help locate trends in various problem categories and help the team improve performance through group meetings or education. The statistics provide a quantitative evaluation of software correctness and allow prediction, based on Inspections held, of error prone sections of design or code, in order to concentrate development, QA, and testing resources on the most important areas. Additionally, each Inspection's results can be "validated" to ensure proper procedures were followed and the results are legitimate as compared to the project database. As an example, for a FORTRAN detailed design inspection, time

SUMMARY OF INFORMATICS SWTS PROJECT INSPECTIONS STATISTICS

Type of Inspect'n	Lang.	Total Number Held	Total No "Lines" Inspected	DENSITY-OF-PROBS. Per 1000 Lines Major	Minor	Total	TIME-PER-PERSON Per 1000 Lines Meet'g	Prep'n	Total
CODE - NON-DESK	ALL Lang	94	51186	22.0	59.9	81.9	4.6	4.0	8.7
Only	FORTRAN	90	49389	22.4	60.4	82.8	4.6	4.1	8.7
	ASSEMBLY	4	1797	10.1	44.5	54.6	5.0	2.6	7.7
CODE - DESK	ALL Lang	47	23206	21.0	51.3	72.3	3.9	-	3.9
	FORTRAN	43	21308	19.1	48.1	67.2	3.7	-	3.7
	ASSEMBLY	4	1898	42.6	87.6	130.3	6.3	-	6.3
DETAILED DESIGN	ALL Lang	44	10349	76.74	144.6	221.3	14.5	9.8	24.3
	FORTRAN	40	9205	83.1	143.4	226.5	14.5	9.2	23.7
	ASSEMBLY	4	1144	25.3	153.9	179.2	14.3	14.4	28.7
PRELIMINARY DESIGN	ALL Lang	43	13268	68.1	107.5	175.7	10.8	5.4	16.1
	FORTRAN	41	12570	54.3	89.8	144.1	9.1	5.5	14.6
	ASSEMBLY	2	698	316.6	426.8	743.4	39.8	3.7	43.6

This chart summarizes the statistics from Informatics inspections on the NASA Ames SWTS project. The statistics are weighted averages, each inspection being weighted by its size, in lines of design or code.

Figure 1
SWTS Inspections Database Summaries

guidelines are 23 hrs/KLOD (Thousand Lines of Design) per person for preparation plus meeting time and the team can expect to find 83 major and 143 minor problems per KLOD. Meeting times and error rates significantly different should be examined to determine their cause. A trend toward increasing error rates may mean that not enough attention is being directed to proper design. A decreasing error rate may mean design is becoming more effective or, when accompanied by decreasing preparation and meeting times, may mean Inspections are becoming less effective.

The statistics are also used to modify the Inspection process itself or its application. At the beginning of the project, the entrance and exit criteria, the checklists, and the methodology and standards are specialized to the project's particular development environment, languages, and review requirements. As statistics are compiled, evaluations of the data may lead to modifications to the entrance criteria to change the level of materials preparation, to the checklists to alter the attention given to certain design or code areas, and to the project standards to remove ambiguity or set new standards as necessary. Removing software components from an Inspection requirement or adding or deleting an Inspection as a quality gate at a particular design stage to more optimally use available time are options made more apparent by the statistics.

DATABASE ANALYSIS

Examination and analysis of the SWTS Inspection database indicate correlations between preparation time, meeting time, inspection rate, and errors detected. These correlations and others allow the overall Inspections procedures to be modified and guidelines established for the optimal conduct of Inspections within a project.

For FORTRAN code Inspections, errors detected are related to inspection rate (LOC inspected per hour), figure 2. Most sessions inspected code at the rate of 100 to 300 LOC per hour and detected between 10 and 80 major errors/KLOC. When the Inspection rate is too rapid, the error detection rate falls gradually. When the Inspection rate is excessively slow, there is a wide range of error densities. For excessively slow Inspection rates, we believe this wide range of error densities results from Inspecting two types of materials: "Difficult Materials" where the materials are complex and require a slower Inspection rate to evaluate but result in a normal to above normal error density; and "Poorly Prepared Materials" which were not ready for Inspection, but were still inspected and thus generated a large number of errors, were difficult to understand, and slow to inspect. The inspection of "Poorly Prepared Materials" represent abnormal situations which the moderator is supposed to prevent prior to scheduling or holding an Inspection. To this end, there are also cut-off limits before and within the Inspection, if the Inspected materials are too hard to understand and/or are producing too many errors, that is, they are probably not ready to be Inspected, the Inspection is stopped and the materials are returned to the author to be properly prepared.

There is a linear correlation between inspection rate and preparation rate (LOC/hr), figure 3. Materials requiring a slower preparation rate also experience a slower Inspection rate, and vice versa. We believe the correlating factor is complexity of materials, more "difficult" code takes more inspector preparation time and more inspection time (lower inspection rate).

Errors Detected vs. Inspection Rate

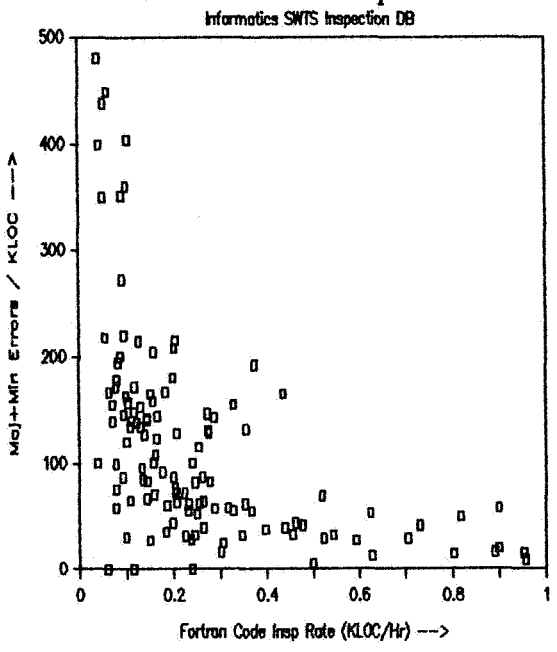


Figure 2

Inspection Rate vs. Preparation Rate

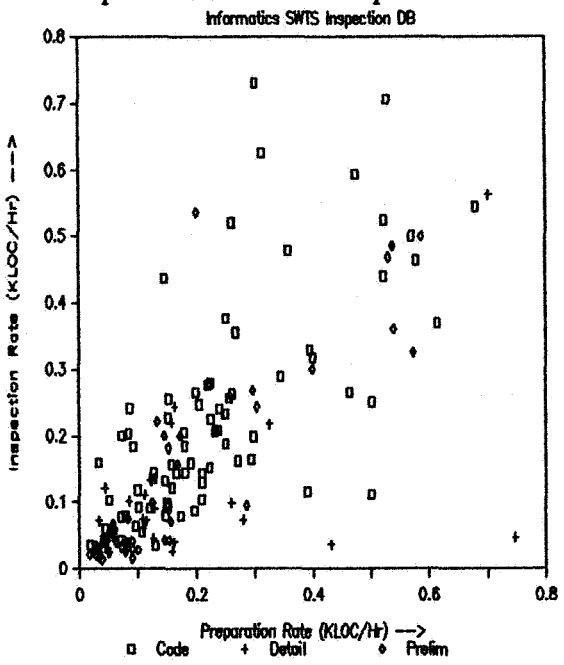


Figure 3

Of any Inspection, we believe the Preliminary Design Inspection is the most critical Inspection to hold, as it helps find modularization errors, data definition errors, and can help to emphasize software re-usability before unit development begins. Based upon major error detection rate and translating preliminary and detailed design lines of design (LOD) to implemented lines of code (LOC), the preliminary design Inspection detects (and removes) a greater number of errors. The translation from lines of design to lines of code is based on a development methodology that requires a preliminary design modularization with logic development where 1 LOD can eventually be coded by 15 to 20 LOC; detailed design logic development is where 1 LOD can be coded by 3 to 10 LOC. Using major errors normalized to estimated implemented LOC, the preliminary design Inspection finds and fixes about 1000 errors per KLOC, the detailed design Inspection locates about 600 errors per KLOC, while the code Inspection is least effective by detecting a mere 20 errors per KLOC. Using the generally accepted cost to repair of an order of magnitude for errors between successive development steps further emphasizes these figures for cost savings purposes: a few ounces of prevention are worth pounds of cure. The SWTS environment uses walkthroughs for reviewing functional requirements specifications; for environments that uniformly use Structured Analysis to generate specifications, the Requirements Specification Inspection would undoubtedly supercede the Preliminary Design Inspection in importance.

Experience in performing Inspections is cumulative and if applied can have an effect on the Inspections process. Over the first two years on the SWTS project, the error rates were widely scattered. In the second year, an examination of the Inspections process resulted in changes in error definition, Inspections procedures, and staff education. Consequently error rates dropped significantly and today remain in a much smaller range.

CONCLUSION

Inspections are not a panacea for Quality Assurance defect removal. They are technical review procedures and may not be appropriate for some situations such

as those needing heavy user interaction (such as user interface definition). They should be used in conjunction with (but probably not as a substitute for) military PDR/CDR large reviews. In appropriate situations, they have been proven to be effective and efficient error detection methods which have extremely important and beneficial "side effects" of accurate planning, scheduling, and tracking for project management and control. The primary effect of Inspections is to move error detection and correction to the earlier (and less costly) development stages. As such, this front-loads the project schedule, but the time is more than recovered during the coding and implementation phases. Consequently, Inspections usage on a project requires proper education, scheduling, and implementation and should not be used on schedule driven projects where the customer understands only two development phases: code and test.

At NASA Ames, based on experience gained using the original IBM model on pilot projects, Inspections have been modified and specialized for numerous projects, development phases, and environments. At Ames, Inspections are expected to play an increasingly major role as a Quality Assurance tool in software development. Some of the directions this can be expected to take are expansion to cover new software languages, incorporation of new structured development methodologies, and modification of the methodologies for the Ames environment based on information gained during Inspections of software developed using those methodologies. Inspections are a significant Quality Assurance tool in their own right and flexible enough to be integrated and implemented with other tools, especially defect prevention, to provide a comprehensive Quality Assurance environment to approach zero defect products.

REFERENCES

1. M.E.Fagan, "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, Vol.15 No.3, 1976
   (This article can be ordered as a reprint, order no. G321-5033)
2. "Inspections in Application Development - Introduction and Implementation Guidelines", Installation Management Manual GC20-2000-0, IBM Corporation, 1977
3. "Guidelines for Software Inspections", NASA Contractor Report 166521, August 1983, NASA Ames Research Center, Moffett Field, Calif. 94035

THE VIEWGRAPH MATERIALS

for the

G. WENNESON PRESENTATION FOLLOW

SOFTWARE INSPECTIONS AT NASA AMES

METRICS FOR

FEEDBACK

AND

MODIFICATION

GREG WENNESON

INFORMATICS GENERAL CORPORATION

WHAT THEY ARE (AND ARE NOT)

INSPECTIONS :

FORMAL REVIEW PROCEDURES
FOR ERROR DETECTION ONLY
DEFINED TEAM MEMBER ROLES
SPECIFICALLY DEFINED TOOLS
HELD AT SELECTED POINTS IN DEVELOPMENT CYCLE
DEFINED INPUT
DEFINED OUTPUT

INSPECTIONS ARE NOT :

DESIGN SESSIONS
WALKTHROUGHS
EVALUATIONS OF THE AUTHOR
RUBBER STAMP PROCEDURES

HISTORY

AT IBM

MIKE FAGIN, PUBLISHED - 1976
ALSO - O.R.KOHLI, R.R.LARSON, R.A.RADICE

FORMAL GUIDELINES - 1977, 1978

PRODUCTIVITY GAIN 23%

ERROR DETECTION 82%

ERROR REDUCTION 38%

AT NASA AMES

PILOT PROJECTS BY INFORMATICS - 1979
(ALSO COMMERCIAL PILOT PROJECTS)

STANDARDIZED WIND TUNNEL SYSTEM (SWTS)

PRODUCTIVITY GAIN 40%*

ERROR DETECTION 90%*

ERROR REDUCTION 40%*

(* - INCLUDES MAJOR METHODOLOGY CHANGES)

NOW USED ON MOST INFORMATICS AMES PROJECTS

INSPECTION COMPONENTS

DEFINED TOOLS

- STANDARDS
- CRITERIA FOR MATERIALS PREPARATION
- CHECKLISTS FOR ERRORS
- EXIT CRITERIA
- WRITTEN RECORDS AND STATISTICS

TEAM MEMBERS

- MODERATOR
- READER
- INSPECTORS
- AUTHOR

INSPECTION PROCESS

- TEAM SELECTION (PLANNING)
- OVERVIEW
- PREPARATION
- INSPECTIONS SESSIONS — DESK INSPECTION
- REWORK
- FOLLOW-UP

PROBLEM AND STATISTICS RECORDING

PROBLEM RECORDING

MODULE INSPECTION PROBLEM REPORT

"GENERAL" PROBLEMS REPORT

PROBLEM STATISTICS

MODULE PROBLEM SUMMARY

MODULE TIME AND DISPOSITION REPORT

INSPECTION STATISTICS

INSPECTOR TIME REPORT

INSPECTION GENERAL SUMMARY

OUTLINE OF REWORK SCHEDULE

G. Wenneson
Informatics General Corp

INSPECTIONS DATA BASE FOR SWTS

- SUMMARIES -

SUMMARY OF INFORMATICS SWTS PROJECT INSPECTIONS STATISTICS

Type of Inspect'n	Lang.	Total Number Held	Total No "Lines" Inspected	DENSITY-OF-PROBLEMS Per Thousand Lines Major	Minor	Total	TIME-PER-PERSON Per Thousand Lines Meet'g	Prep'n	Total
CODE - NON-DESK Only	ALL Lang	94	51186	22.0	59.9	81.9	4.6	4.0	8.7
	FORTRAN	90	49389	22.4	60.4	82.8	4.6	4.1	8.7
	ASSEMBLY	4	1797	10.1	44.5	54.6	5.0	2.6	7.7
CODE - DESK	ALL Lang	47	23206	21.0	51.3	72.3	3.9	0.0	3.9
	FORTRAN	43	21308	19.1	48.1	67.2	3.7	0.0	3.7
	ASSEMBLY	4	1898	42.6	87.6	130.3	6.3	0.0	6.3
DETAILED DESIGN	ALL Lang	44	10349	76.74	144.6	221.3	14.5	9.8	24.3
	FORTRAN	40	9205	83.1	143.4	226.5	14.5	9.2	23.7
	ASSEMBLY	4	1144	25.3	153.9	179.2	14.3	14.4	28.7
PRELIMINARY DESIGN	ALL Lang	43	13268	68.1	107.5	175.7	10.8	5.4	16.1
	FORTRAN	41	12570	54.3	89.8	144.1	9.1	5.5	14.6
	ASSEMBLY	2	698	316.6	426.8	743.4	39.8	3.7	43.6

This chart summarizes the statistics from Informatics inspections on the NASA Ames SWTS project. The statistics are weighted averages, each inspection being weighted by its size, in lines of design or code.

G. Wenneson
Informatics General Corp

STATISTICS USE

AUTHOR

PROBLEM REPORTS
MODULE PROBLEM SUMMARY
PREVIOUS INSPECTION STATISTICS

DESIGN TEAM AND MANAGER

PROBLEM REPORTS
MODULE PROBLEM SUMMARY
OUTLINE OF REWORK SCHEDULE
MODULE TIME AND DISPOSITION
INSPECTION GENERAL SUMMARY
PREVIOUS INSPECTION STATISTICS

PROJECT MANAGER; TEST GROUP; QA GROUP

MODULE PROBLEM SUMMARY
INSPECTION GENERAL SUMMARY
PREVIOUS INSPECTION STATISTICS

SOFTWARE ENGINEERING

MODULE PROBLEM SUMMARY
INSPECTION GENERAL SUMMARY
PREVIOUS INSPECTION STATISTICS

CODE INSPECTION SUMMARIES
NEW FORTRAN CODE, MODIFICATIONS, AND BOTH

SUMMARY OF INFORMATICS SWTS PROJECT INSPECTIONS STATISTICS

Type of Inspect'n	Lang.	Total Number Held	Total No "Lines" Inspected	DENSITY-OF-PROBLEMS Per Thousand Lines Major	Minor	Total	TIME-PER-PERSON Per Thousand Lines Meet'g	Prep'n	Total
CODE - NON-DESK CHECK									
	FORTRAN	90	49389	22.4	60.4	82.8	4.6	4.1	8.7
	/New	46	25981	26.3	68.3	94.6	5.5	4.9	10.3
	/Mods	13	7019	17.2	42.4	59.6	3.0	3.2	6.2
	/Both	31	16389	18.5	55.6	74.1	3.9	3.3	7.2
CODE - DESK CHECK									
	FORTRAN	43	21308	19.1	48.1	67.2	3.7	0.0	3.7
	/New	8	4121	26.3	51.7	78.0	4.9	0.0	4.9
	/Both	25	14453	18.6	50.1	68.7	3.4	0.0	3.4
	/Mods	10	2734	10.6	32.2	42.8	3.8	0.0	3.8

This chart summarizes the statistics from Informatics inspections on the NASA Ames SWTS project. The statistics are weighted averages, each inspection being weighted by its size, in lines of design or code.

G. Wenneson
Informatics General Corp

INSPECTIONS DATA BASE

"MAJOR" PROBLEM DISTRIBUTION, BY PERCENT

PRELIMINARY DESIGN

Category	FORTRAN	ASSEMBLER
SPECIFICATION	10%	13%
CLARIFICATION	17	1
DATA	18	21
LOGIC	21	21
I/F	5	20
LINKAGES	20	
PERFORMANCE	4	3

DETAILED DESIGN

DETAIL	9	29
LOGIC	29	66
DATA	20	1
LINKAGES	22	1
RETURN CODES	5	

CODE

FUNCTIONALITY	9	4
DATA	19	37
CONTROL	18	22
LINKAGES	24	23
READABILITY	17	2
REG. USE		12

PREVIOUS INSPECTIONS EFFECT ON MAJOR ERROR RATES

STAGE OF DEVELOPMENT	NUMBER OF PREVIOUS INSPECTIONS 0	1	2	3
CODE NON-DESK	17.7	30	32.6	38
CODE DESK	15.1	27	30	21
DETAIL DESIGN	95	79	54	-
PRELIM. DESIGN	58	45.6	-	-

Major Errors Per KLOC

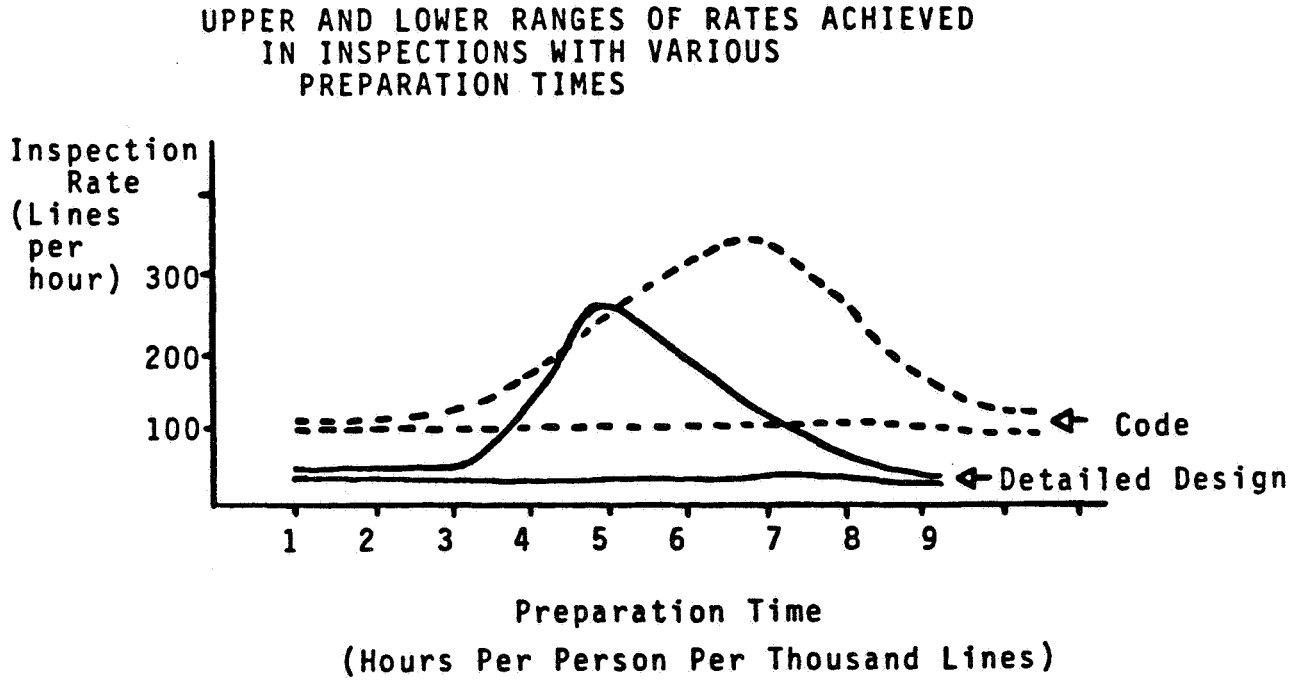
AND ON PREPARATION AND MEETING TIME

STAGE OF DEVELOPMENT	NUMBER OF PREVIOUS INSPECTIONS 0	1	2	3
CODE NON-DESK	8.2	9.2	9.1	10
CODE DESK	4	3.2	3.5	2.5
DETAIL DESIGN	27.7	23.0	9.5	-
PRELIM. DESIGN	14.7	14.4	-	-

HOURS of Preparation plus Meeting time Per KLOC

INSPECTIONS RATE AND PREPARATION TIME RELATIONSHIP

An important area of consideration is the amount of preparation time required in order to allow the participants to proceed at a reasonable rate in the inspection meeting. The graph below, based on the individual inspections to date, suggests that preparation times of 4-7 hours per 1,000 lines may allow the team to proceed at an optimum rate in the meetings. Less preparation time will cause the meeting to slow down because of poor understanding and many questions. More preparation time may have a negative impact on the rate because of over-emphasizing minor problems or discussing the functionality or goals during code or design inspections.



INSPECTIONS AS A PROJECT COORDINATION TOOL

INSPECTIONS CAN INTEGRATE THE FOUR MAJOR PROJECT FACTORS:

PROJECT MANAGEMENT

METHODOLOGY

QUALITY ASSURANCE

STAFF PERFORMANCE

THRU:

REINFORCEMENT OF METHODOLOGY AND STANDARDS
MAJOR MILESTONE TRACKING INFORMATION MATCHING WBS
DETAILED TRACKING AND ESTIMATING INFORMATION MATCHING WBS
DETAILED ERROR AND DESIGN NEEDS AT EACH DEVELOPMENT STAGE
EASY EXTRACTION OF TECHNICAL INFORMATION ABOUT COMPONENTS
INDICATIONS OF TRAINING AREAS NEEDING ATTENTION ACROSS THE PROJECT
INDICATIONS DIRECTLY TO INDIVIDUAL STAFF MEMBERS OF THEIR TRAINING NEEDS

G. Wenneson
Informatics General Corp

ALMOST THE END

CAUTIONS

DOESN'T SUBSTITUTE FOR THINKING
MUST BE SCHEDULED AT BEGINNING - CAN'T BE "TACKED" ON
PARTICIPANTS MUST BE PROPERLY TRAINED
NEED CUSTOMER UNDERSTANDING AND SUPPORT
MANAGEMENT DIRECTION AND SUPPORT CRUCIAL
STATISTICS ARE FOR BETTER SOFTWARE AND MANAGEMENT,
NOT A NUMBERS EXERCISE

WHERE TO GO FROM HERE

EXPAND TO NEW LANGUAGES AND DESIGN TECHNIQUES
EXPAND TO NEW METHODOLOGIES AND SUPPORT TOOLS
FEEDBACK TO CURRENT METHODOLOGIES
EXPAND TO OTHER APPLICABLE COMPANY/CONTRACT AREAS

PANEL #3

SOFTWARE ENVIRONMENTS

C. Gill, Boeing Computer Services
A. Reedy, Planning Research Corporation
L. Baker, TRW Defense Systems Group

A KNOWLEDGE BASED SOFTWARE ENGINEERING ENVIRONMENT TESTBED

Chris Gill
Boeing Computer Services

The Carnegie Group Incorporated (CGI) and the Boeing Computer Services Company (BCS) are jointly developing a knowledge based software engineering environment testbed. The goal of this multi-year experiment is to demonstrate dramatic improvements in software productivity by applying Artificial Intelligence (AI) techniques to the software development process. The resultant environment will provide a framework in which conventional software engineering tools can be integrated with AI based tools to promote software development automation.

Objective

The objectives of the testbed are:

- to demonstrate the integration of multiple techniques for a system that improves both the software development process and the quality of the software being developed;
- to determine, through experimentation, the benefits that may result from AI technology;
- to evaluate alternative functional implementations; and
- to provide a preliminary development facility for building advanced software tools.

The primary emphasis of the testbed is on the transfer of relevant AI technology to the software development process. The primary experiments relate to AI issues, such as scaling up, inference, and knowledge representation.

Approach

The approach being used is two-fold:

- to explore the use of AI tools and techniques for a software engineering environment framework; and
- to explore the use of AI tools and techniques for specific software engineering tools.

The environment will provide functionality for Project Management, Software Development Support and Configuration/Change Management throughout the software lifecycle. For purposes of the experiments, the development environment is considered to have three dimensions: the functional areas mentioned above, the life cycle phases, and a dimension of potential AI techniques. These potential techniques can be grouped into three major categories:

- knowledge representation, which deals with modeling software project concepts and links;
- inference mechanisms, which deal with the ways this knowledge can be used to solve user development problems; and
- knowledge based interface, which deals with intelligent display, explanation, and interaction with the user.

Figure 1 illustrates the three dimensions of the experiment.

Status

We have proceeded in a breadth-first manner, performing experiments in each cell of the matrix in Figure 1 rather than concentrating on any particular cell. During the first year of the project CGI has:

- created a model of software development by representing software activities;
- developed a module representation formalism to specify the behavior and structure of software objects;
- integrated the model with the formalism to identify shared representation and inheritance mechanisms
- demonstrated object programming by writing procedures and applying them to software objects (e.g., propagating changes in a development system);
- used data-directed reasoning to infer the probable cause of bugs by interpreting problem reports;
- used goal-directed reasoning to evaluate the appropriateness of a software configuration; and
- demonstrated knowledge based graphics by converting software primitives to low level graphic primitives.

Plans

Plans for the next phase include completing experiments in the remaining cells of the Figure 1 matrix along with some additional general AI experiments, including:

- use of knowledge based simulations to perform rapid prototyping or to try alternative project schedules;
- use of natural language interfaces for user interaction;
- use of a blackboard architecture to permit "experts" to confer with each other to solve problems; and
- use of distributed processing that would permit separate systems to act upon goals sent to them by others.

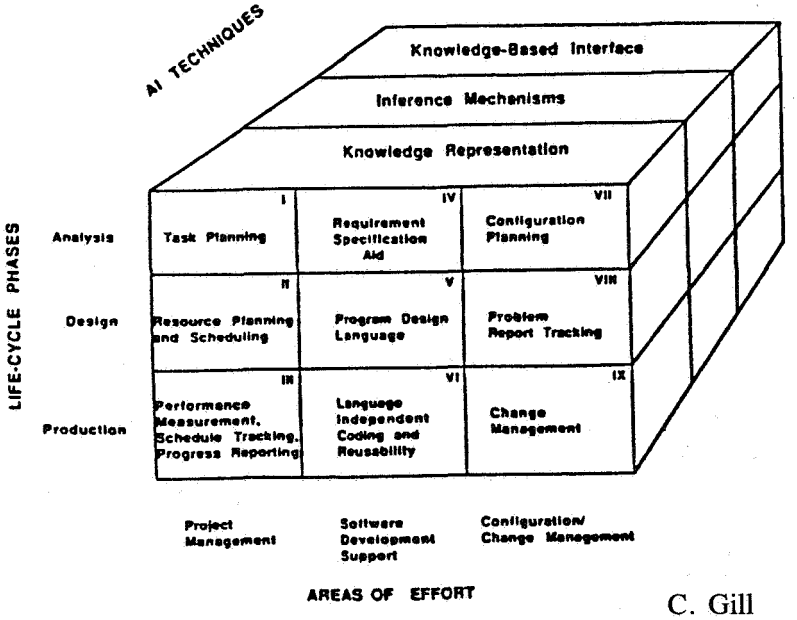


FIGURE 1: ENVIRONMENT FUNCTIONALITY

C. Gill
Boeing Computer Service

THE VIEWGRAPH MATERIALS

for the

C. GILL PRESENTATION FOLLOW

C. Gill
Boeing Computer Service

Boeing Computer Services
Artificial Intelligence Center

A KNOWLEDGE-BASED SOFTWARE ENGINEERING ENVIRONMENT TESTBED

Presentation to

NASA

TENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

Chris Gill

Boeing Computer Services

Boeing Computer Services
Artificial Intelligence Center

Introduction

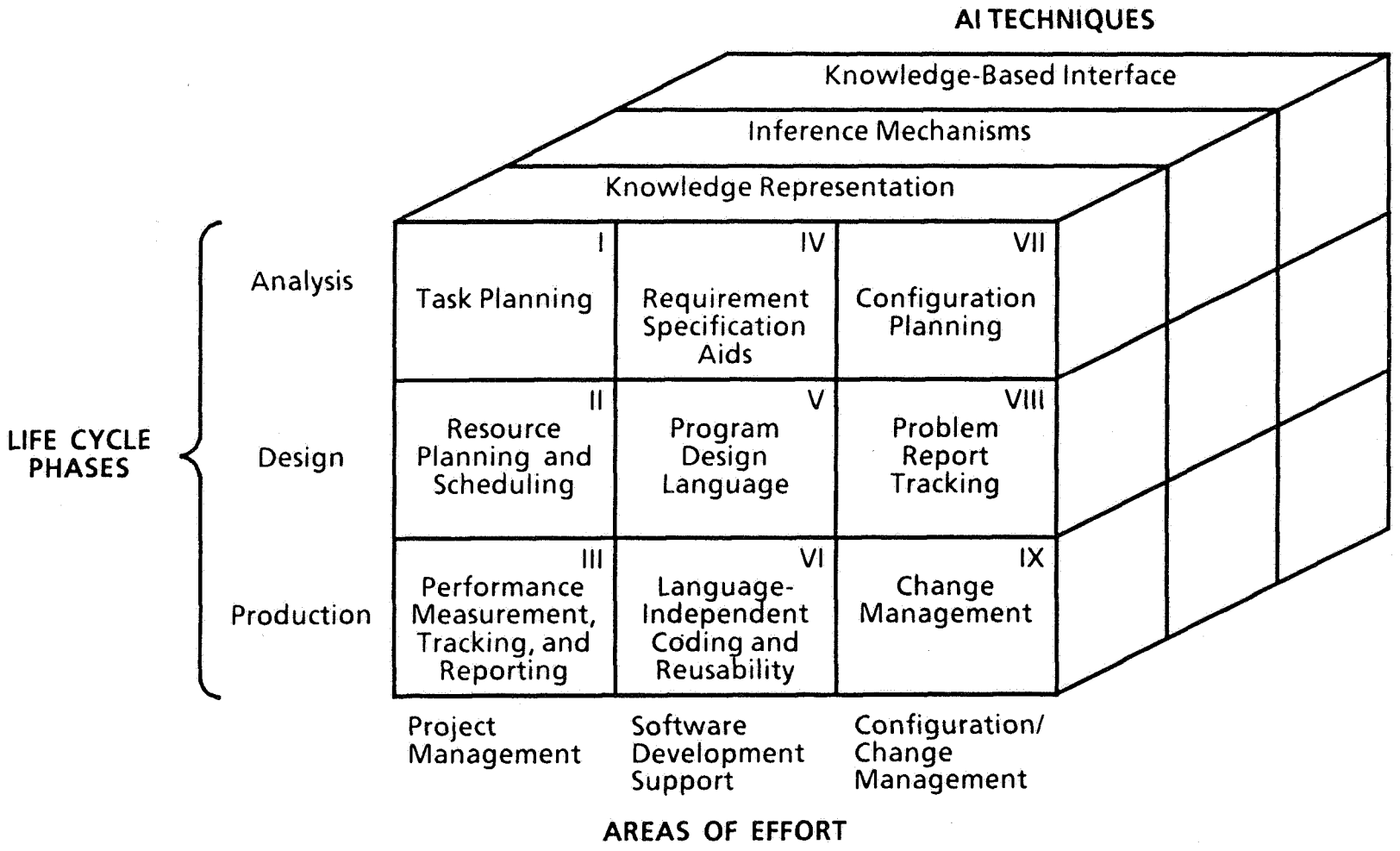
- Need large gains in software productivity
  - AI shows promise
  - Promise has not been demonstrated
- Series of experiments
  - AI applied to Software Engineering
  - Integration of tools
- Joint venture
  - BCS
  - CGI
- Multiyear project
  - First year complete
  - More experimentation needed

Boeing Computer Services
Artificial Intelligence Center

Objective

- Determine benefits of AI applied to Software Engineering
- Demonstrate improvement in software development process
- Demonstrate improvement in software quality
- Develop testbed for experimentation

Approach


AI TECHNIQUES
Knowledge-Based Interface
Inference Mechanisms
Knowledge Representation
LIFE CYCLE PHASES
Analysis
Design
Production
I
Task Planning
IV
Requirement Specification Aids
VII
Configuration Planning
II
Resource Planning and Scheduling
V
Program Design Language
VIII
Problem Report Tracking
III
Performance Measurement, Tracking, and Reporting
VI
Language-Independent Coding and Reusability
IX
Change Management
Project Management
Software Development Support
Configuration/ Change Management
AREAS OF EFFORT

Boeing Computer Services
Artificial Intelligence Center

Status

- Module representation formalism
- Object programming
- Data-directed reasoning
- Goal-directed reasoning
- Knowledge-based graphics

Boeing Computer Services
Artificial Intelligence Center

Plans

- Knowledge-based simulation
- Natural language interface
- Blackboard architecture
- Distributed processing

Boeing Computer Services
Artificial Intelligence Center

Conclusions

- **Tremendous promise**
- **Leverage of integration**
- **Need to use on real projects**

Experience with a Software Engineering Environment Framework

by
R. Blumberg, A. Reedy, and E. Yodis
Planning Research Corporation

1.0 Introduction

This paper describes PRC's experience to date with a software engineering environment framework tool called the Automated Product Control Environment (APCE). The paper presents the goals of the framework design, an overview of the major functions and features of the framework, a summary of APCE use to date, and the results and lessons learned from the implementation and use of the framework. Conclusions are drawn from these results and the framework approach is briefly compared to other software development environment approaches.

2.0 Framework Goals

The APCE was developed to reduce software lifecycle costs. The approach taken was to increase automation of the software lifecycle process and thereby to increase productivity. It was felt that maximum cost reduction could be achieved for the short term by attacking three major problem areas:

- o automation of labor intensive but routine administrative tasks
- o provision of an overall control, coordination, and enforcement framework and information repository for existing tools
- o provision for maximum framework portability, distributability, and data interoperability with the bounds of performance constraints

A distinction was made between tools and the environment. In the PRC view, tools are active elements in the software lifecycle process. They create or modify (document or software) components, test components, or order the execution of groups of tools upon components. The environment or framework, on the other hand, is a more passive element. It provides for overall control, coordination, and enforcement and acts as an information repository. This distinction is important because it serves to separate environment or framework issues from tool issues. PRC wanted to build a framework which could incorporate existing tools. In this way, PRC could build on the excellent work done by others in the tool arena in a timely fashion.

3.0 APCE Overview

The APCE provides automation for:

- o real-time project status tracking and reporting
- o configuration management of software, documentation, and test procedures
- o requirements traceability and change impact traceability
- o test bed generation, component integration, and system integration

A brief overview of how the APCE is organized to support these functions and how the APCE is designed to support portability, distributability, and interoperability is given below.

3.1 Automation and Control

As suggested by Stoneman [1], a database provides the integrating mechanism for the environment framework. The database design incorporates a flexible model of the software development process. Project definition information based on this model is entered into the database during project initialization, and this information is used to control the project and provide the basis for automated tracking and configuration management. The project definition is divided into three components as illustrated in Slide 3 (APCE Entities).

User groups are identified as managers, developers (those who create products), or testers; multiple roles are allowed. The organizational hierarchy is also described so that project problem reports can be automatically forwarded up the chain of command if they are not promptly dealt with. Products, both documents and software, are described in terms of their component structure and are associated with software lifecycle phases which are also entered into the APCE database. Slide 4 (APCE - DOD Documentation and Review Sequence) illustrates the lifecycle phases as specified in Mil-Std 2167.

The levels of integration describe the hierarchy of the test and integration processes that all products (documents or software) must go through. This testing process allows for the enforcement of project standards and quality assurance. The APCE uses the product structure and test procedures developed by the testers to automatically create testing baselines and test harnesses as required.

3.2 Portability, Distributability, and Interoperability

The APCE approach to support for portability, distributability, and interoperability is based on three architectural features:

- o APCE Interface Set (AIS)
- o data-coupled design
- o open system architecture

These features are illustrated in Figure 1 (APCE Static View), which shows the APCE as part of a Software Engineering Environment (SEE).

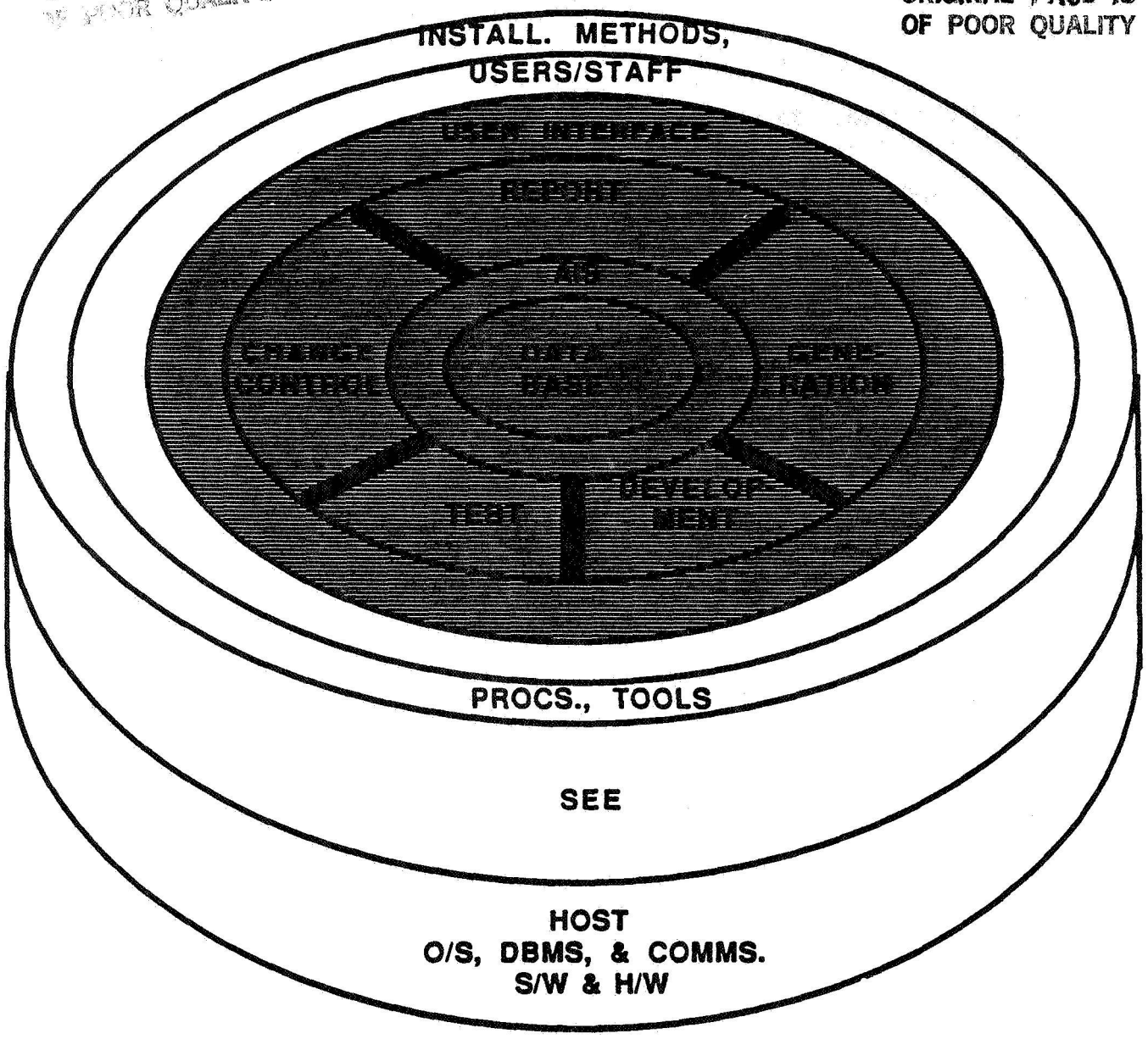
The APCE subsystems and data management capabilities depend on a standard set of interfaces to system services called the AIS. These interfaces define a Stoneman Kernal Ada® Programming Support Environment (KAPSE) like layer for portability purposes. The AIS allows a mapping to existing operating system services. If the needed level of support is not directly available from the host operating system, then an extra layer of software is created to satisfy the requirement. Existing operating system services are not duplicated. The AIS is not based on an implicit model like the Common APSE Interface Set (CAIS) [2].

The data-coupled design provides for both control and distributability. All project information is stored in the framework database. The database controls the activities of the APCE functional subsystems since they do not interface directly but interact through the database. Users do not directly manipulate the database; they affect the database contents indirectly through interaction with the functional subsystems. The database is designed to minimize information exchange, so data is distributable (without replication). The functional subsystems are also distributable since they are controlled by the database contents. The database design is controled by the framework and hidden from the users. Thus, integrity and interoperability of data is ensured.

The open system architecture approach means that the APCE allows the use of existing host tools, including management scheduling and costing tools. The APCE does not interface directly with the tools but rather controls tool invocation and the tool products. Both existing and future tools can be used within the APCE framework without alterations.

4.0 Results

The APCE has been used on a variety of in-house and client projects over the past 21 months. It has been used in-house at PRC to support proposal and document production as well as software development and lifecycle maintenance projects. The framework has also been installed for Army, Navy, and Air Force clients. In one example client installation, APCE features were used to bring a software system under configuration control for a Navy software support activity. The



APCE

INSTALLATION CAPABILITIES

NO INTERFACE ACROSS THIS LINE

FIGURE 1: APCE STATIC VIEW

The full project team for Project 1 consisted of 9 persons, including a manager, 2 computer system scientists, 1 system analyst, 1 analyst, and 4 associate analyst/programmers. Two of the associate analyst/programmers acted as the test team. All of the other team members, except the manager, functioned as APCE developers. The senior staff members were quite experienced with 10 to 15 years experience each. The junior staff members were all new college graduates with no commercial programming experience and no VAX experience. The APCE allowed all personnel to be extremely productive despite their learning curve with a new machine and a new environment.

4.2 Cost/Benefit Analysis

PRC has conducted a cost/benefits analysis of APCE use for one of our clients. This client needed configuration management and lifecycle maintenance control for mission critical software. PRC developed plans for both a manual and a APCE controlled development support facility and plans for transitions to these facilities. A estimation of both the transition costs and the annual recurring resource costs was performed for both facilities. The results of the analysis are given on Slides 6 (Level of Effort Analysis) and 7 (Cumulative Cost Comparison).

The estimated times for transition to both the manual and the APCE controlled facilities were the same (3 months). The activities involved in the transition period involve the establishment and implementation of policies and procedures and, in the case of the APCE controlled facility, the installation of software and training. As shown on Slide 6 (Level of Effort Analysis), the cost for transition in terms of effort was slightly more for the APCE controlled facility. However, the total labor months required for the first year and following years were very much less for the APCE controlled facilities.

Slide 7 (Cumulative Cost Comparison) shows the total cumulative costs of the two facilities projected over a two year period. The larger initial costs for the APCE controlled facility is caused by the APCE licensing fees. The cumulative costs of the manual facility surpass the costs of the APCE controlled facility after seven months (4 months after transition). The cost savings achieved by the APCE facility are due to the increased automation of the control, tracking, and configuration management functions. The estimates did not include cost savings due to increased productivity of developers and testers.

4.3 Portability

The framework has proven very easy to rehost. Part of this ease is due to the design features of the AIS and part is due to rigid enforcement of coding standards for the transportable portions of the APCE. To rehost the APCE on a new machine, all that is necessary is to reimplement the AIS functions. The APCE transportable subsystems have been written in C using coding standards designed to eliminate use of "non-standard" features of the language. The C programming language was orginally

framework is now being used to continue control throughout the maintenance cycle, including the incorporation of module upgrades supplied by other contractors. These various applications of the framework have resulted in rehosting of the APCE to a variety of different hardware configurations. This experience in using the APCE has allowed PRC to collect the data on productivity, transportability, and distributability presented below.

4.1 Productivity

At the National Security Industrial Association (NSIA) DOD/Industry Software Technology for Adaptable, Reliable Systems (STARS) Program Conference in April 1984 [3, pg. L-21], the NSIA Industry Study Task Group reported that the average productivity for U.S. software development projects was 200 lines of code per labor month. This works out to a little over 10 lines per day. On unclassified projects with APCE control, PRC has recorded productivity in excess of 120 lines per day. Slide 5 (Example Projects) gives the productivity figures collected for three PRC in-house projects under APCE control. (Client projects are not far enough along to report meaningful productivity figures.) Productivity in these three projects was an order of magnitude greater that the average reported for industry as a whole.

All of the reported projects used a high level programming language (HOL). Project 1 was the initial development of a software system. This system has been maintained under APCE control. The figures given for Project 1 reflect only the developers' labor and do not count time for the manager or the testers (who basically functioned as Quality Assurance personnel). Productivity during upgrades was equivalent or better than that experienced during the initial development. Further details of Project 1 are given below. Project 2 was an upgrade to an existing system under APCE control. This upgrade included full documentation. Project 3 was a prototyping activity, and is somewhat atypical since only partial documentation (e.g., no formal users manual) was produced. The figures given for Project 2 and Project 3 include the testers' time. These projects were small, so the same personnel functioned as both developers and testers.

Project 1 was a four month project to develop system software in the C programming language. The development host was a VAX 11/780 and the non-APCE tools used are commercially available for the VAX. The project products included: System Engineering Plan, Acceptance Test Plan, Functional Description, Preliminary Design Specification, Detailed Specification (22,000 lines of Ada PDL), Operators Manual, and Users Guide in addition to 58,297 lines of source code. In addition, 660 test procedures were developed and used to test the components of the products. (The test procedure development and test time is not included in the productivity figures given for Project 1.) Some of these test procedures were used to enforce the project specific coding and PDL standards.

chosen because it was available on a wide range of host machines. However, it has caused some problems because there are no standards for C. In the process of transporting, some features of C that were assumed to be commonly implemented turned out to be system specific. A single version of the transportable software is maintained that runs on all supported machines. (Future plans call for conversion to Ada as soon as there are Ada compilers on a sufficiently wide range of machines.)

The APCE is now running on the following machines: VAX 11/780 with VMS, ROLM and Data General with AOS/VS, IBM with MVS, and Intel 310 with XENIX®. Slide 8 (Rehost Efforts) presents a summary of rehosting experiences to date.

4.4 Distributability and Interoperability

The environment data is interoperable because the framework controls the database structure and because the framework controls only the products of tools rather than interfacing directly with the tools. The toolsets available on different hosts may differ, but equivalent functionality is usually available. Filters and standard forms can be used to adjust for differences between specific tools. For example, different editors sometimes embbed control characters in the text. Filters are used at PRC head quarters to move text among the VAX EDT editor, the IBM PC Wordstar editor, and the Macintosh MacWrite editor. A standard, plain text form has been established so that only one new filter needs to be written to introduce another editor.

Project data has been proven to be interoperable between different framework installations. Software and documentation have been routinely developed on one installation and then transferred together with documentation, traceability and configuration management information, and project history information to a different installation on different hardware with no problems. This feature has proven useful in allowing project work to proceed in parallel with the APCE rehost to new hardware. That is, the early phases of a project can begin under APCE control on one machine while the APCE is rehosted to the desired development host. When the rehost is complete, the project can be transfered to its own host.

The framework was designed to function in a distributed, heterogeneous hardware environment. Both the database and the processing may be distributed. Work currently underway will allow distribution of developer processing to IBM PCs and Macintoshes connected to a VAX via a local area network. Future plans call for full distribution of both processing and data.

5.0 Conclusions

The preliminary results presented above provide good evidence that the APCE approach can achieve its goals. The framework increases productivity, allows use of existing tools without modification, and is easy to transport. PRC management has been impressed enough to make the

APCE a company standard. The task of technology insertion into large projects has begun. Because of its flexiblity, the APCE can be introduced into existing projects without undue disruption. Most of the transition problems are in the areas of training. The use of the APCE does involve understanding of some basic concepts. During the next few years, more data will be collected on the benefits of using this type of environment framework.

The APCE framework approach is in contrast with other environment approaches both in the areas of goals and of benefits. Many other recently developed environments, such as the Ada Language System (ALS) [4], have a very different set of goals. One of the goals of the ALS is to provide a minimal set of transportable tools including a retargetable Ada compiler. Much of the effort expended in the ALS development has been to develop tools, especially the Ada compiler. Many of the benefits expected from the ALS are the benefits derived from the use of a standard toolset and command language.

The approach taken by the ALS does not allow the use of non-ALS tools. To work with the ALS, existing tools must be rehosted to the ALS KAPSE and rewritten in Ada, if necessary. The ALS tools are transported by rehosting the ALS KAPSE on new hardware just as the APCE framework is transported by rehosting the AIS on a new operating system. The ALS approach means that there will be significant lead time before the ALS has a reasonably full tool set. Further, features such as full configuration management and project reporting must be added as tools to the ALS. These important productivity tools are not part of the minimal toolset. Important aspects of the ALS approach, such as productivity and portability, have yet to be proven. The problem of distribution was not directly addressed in the first version of the ALS.

The ALS approach may work for organizations such as the U.S. Army that wish to standardize as much as possible on a minimal tool set and a limited selection of standard hardware. However, for a contractor with a wide variety of client and internal standards, methodologies, and hardware, a much more flexible approach is necessary. The APCE framework is an example of a viable alternative approach.

References

[1] Requirements for Ada Programming Support Environments ("Stoneman"), Department of Defense, February, 1980

[2] Proposed MIL-STD-CAIS, Department of Defense, 31 January 1985

[3] Proceedings First DOD/Industry STARS Program Conference, NSIA, 30 April - 2 May 1985, San Diego, CA

[4] Architectural Description of the Ada Language System (ALS), Joint Service Software Engineering Environment (JSSEE) Report No. JSSEE-ARCH-001, 3 December 1984

[5] Architectural Description of the Automated Product Control Environment, Draft, 4 October 1985

THE VIEWGRAPH MATERIALS

for the

R. BLUMBERG PRESENTATION FOLLOW

Experience with a Software Engineering Environment Framework

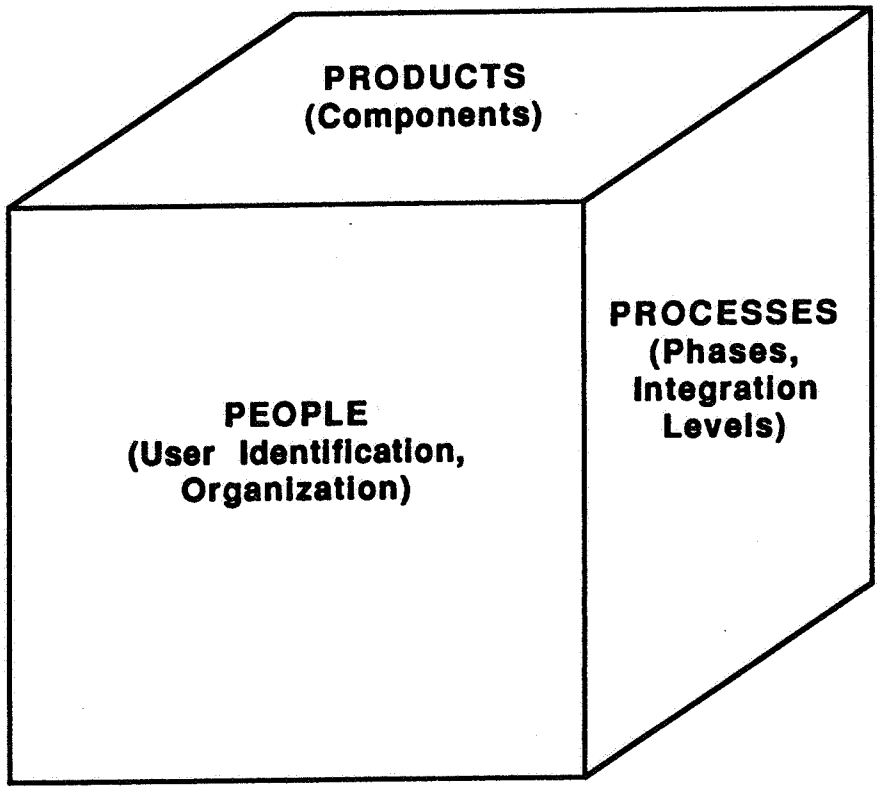
R. Blumberg
A. Reedy
E. Yodis

prc

Automated Product Control Environment

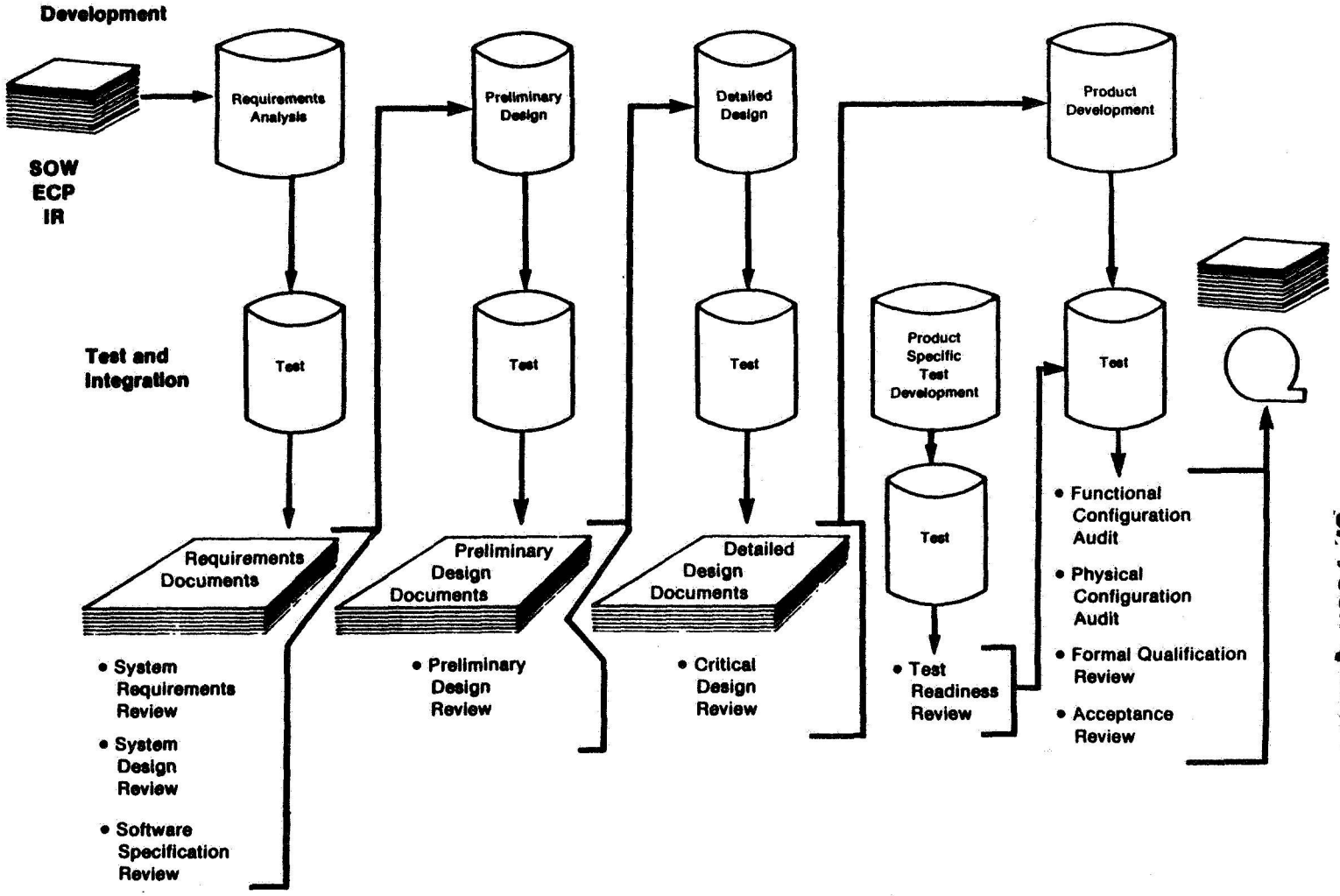
- Goals
  - Environment vs tools
  - Automation
  - Portability, distributability, interoperability
- Functions
  - Project reporting
  - CM
  - Tracking
  - Test and integration

APCE Entities



prc

APCE - DOD Documentation and Review Sequence



ORIGINAL PAGE IS OF POOR QUALITY

prc

Example Projects

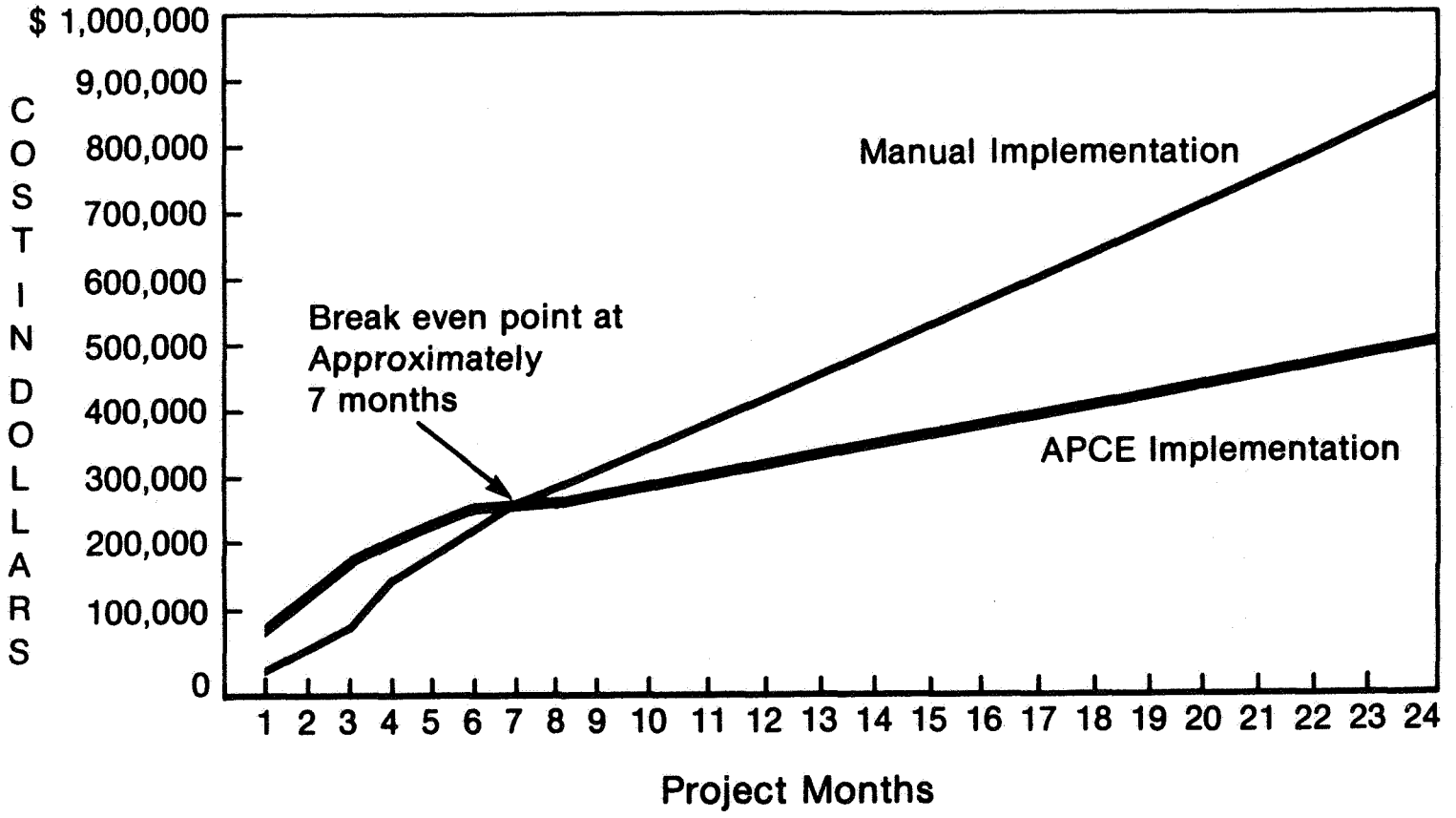
Project	LOC / day	Total LOC	Total personnel	Programming Language
Project 1	121	58,297	6	"C"
Project 2	217	13,024	1.5	"C"
Project 3	384	30,750	2	Pascal "C"

Level of Effort Analysis

First year resource costs	Manual Environment	APCE Environment
Transition activities (months 1 to 3)		
Organization	2.25	3.75
Configuration management	3.75	2.75
QA / testing	4.00	3.75
Development	1.00	1.00
Total transition activities	11.00	11.25
Recurring resource costs (months 4 to 12)	45.00	18.00
Total first year	56.00	29.25
Annual recurring resource costs		
Configuration management	15.00	6.00
QA / test management	21.00	6.00
PSL librarian	12.00	0.00
Coordinator	12.00	0.00
APCE on - site support	0.00	12.00
Total	60.00	24.00

*All values expressed in staff months

Cumulative cost comparison



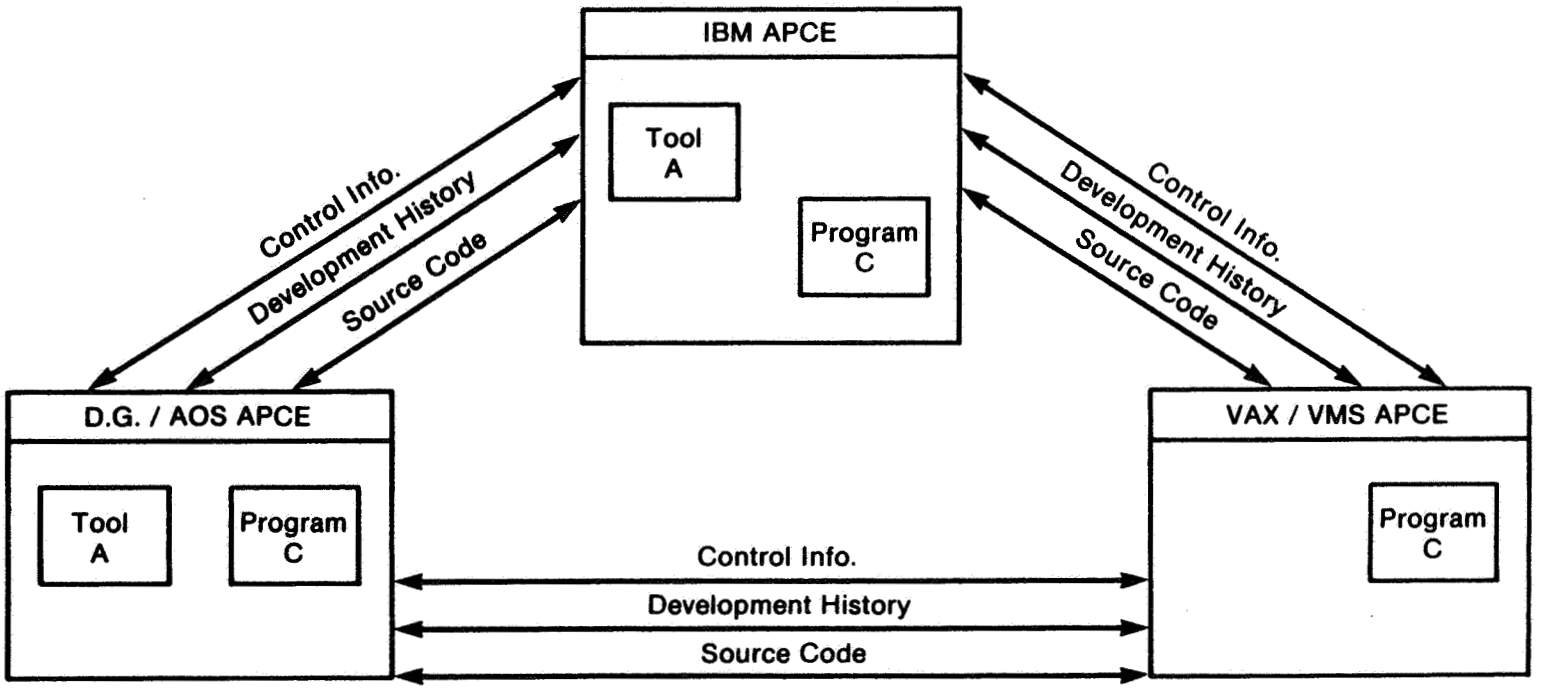
prc

Rehost Efforts

H / W	Calender Time	Team Size
ROLM - DG AOS / VS	1 Month	4 APCE analysts
IBM / MVS	2½ months	5 APCE analysts
Intel 310 XENIX	2 months	1 APCE analyst 2 site personnel

Note: VAX / VMS original host

APCE Interoperability & Portability

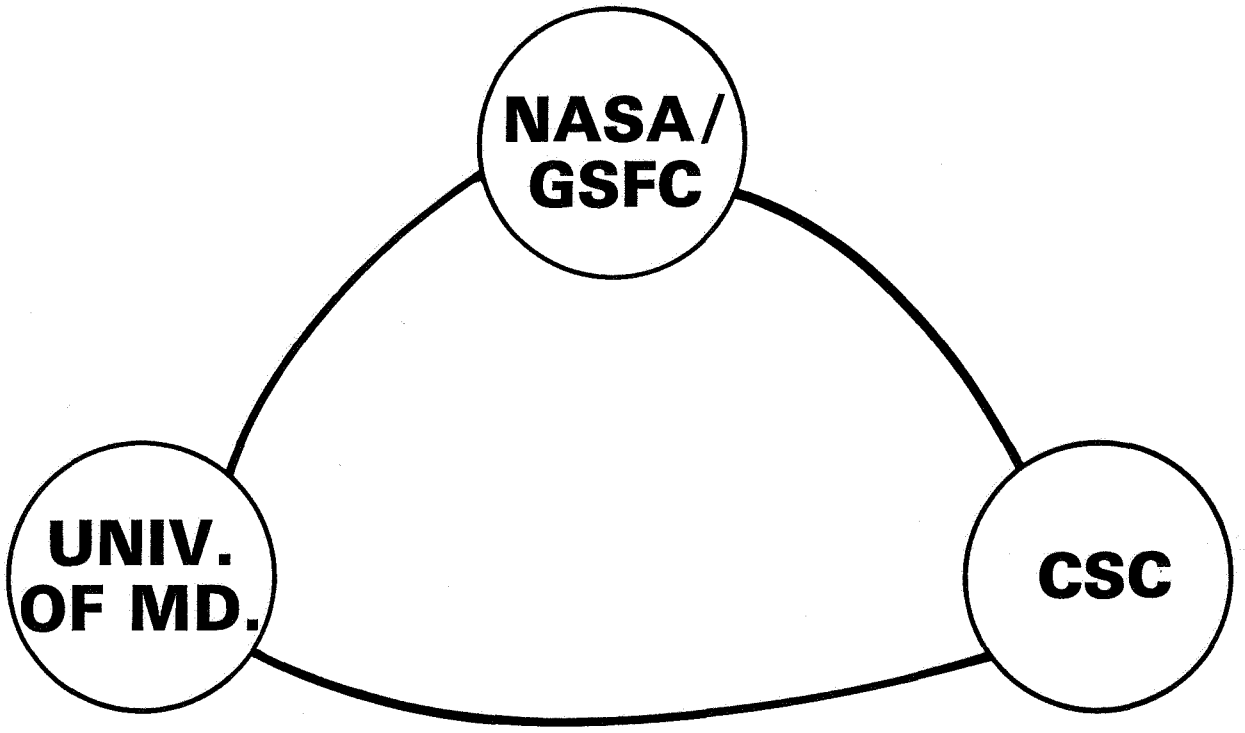


prc

Summary

- Different approach
- Allows use of existing tools
- Good preliminary results
- Acceptance as a corporate standard

SOFTWARE ENGINEERING LABORATORY

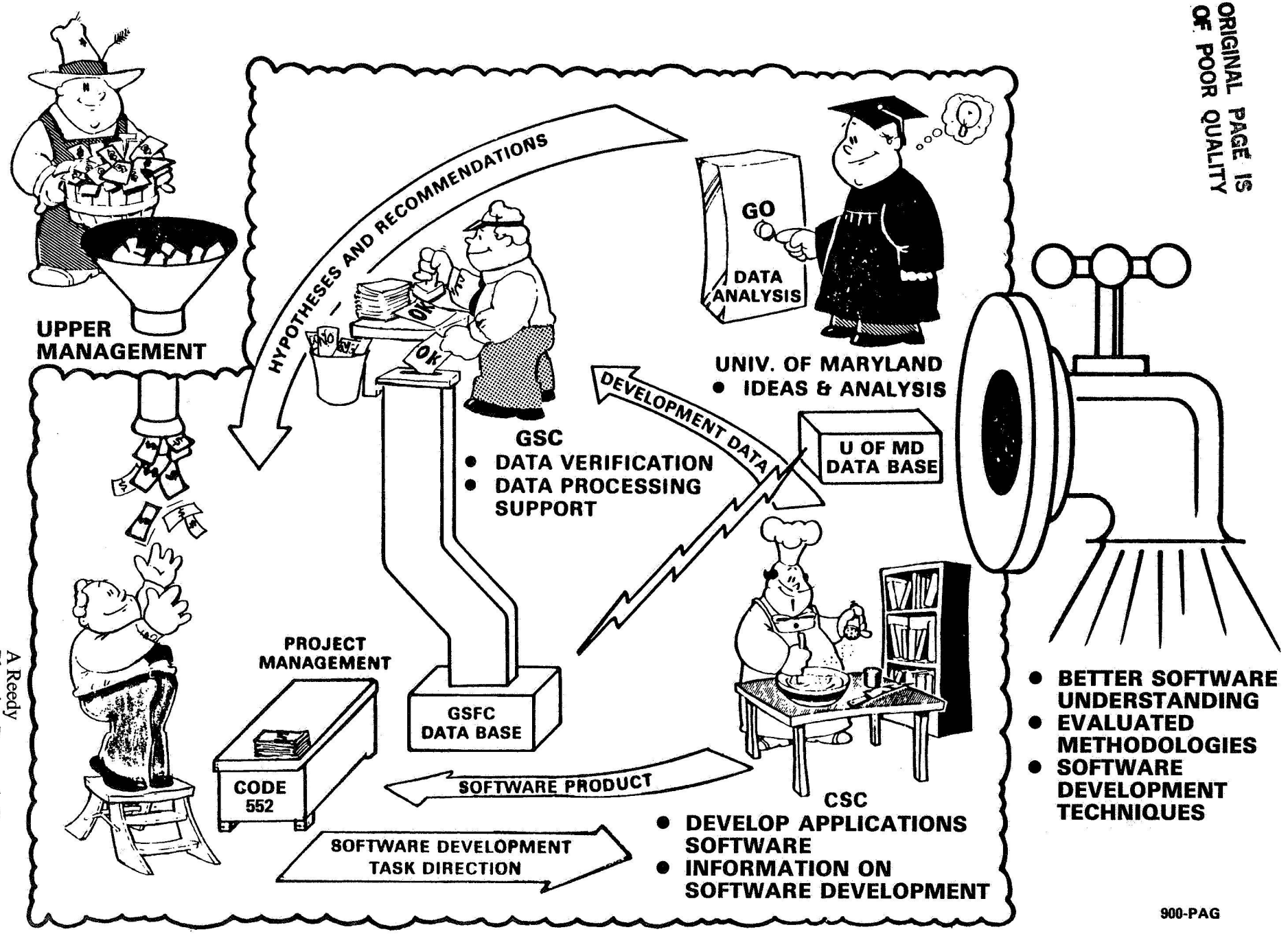


900-PAG-(128*)

SOFTWARE ENGINEERING LABORATORY

- **WHEN: ESTABLISHED IN 1976 BY NASA/GSFC**
- **WHY: TO IMPROVE ITS SOFTWARE DEVELOPMENT PROCESS**
- **HOW: 1) MEASURE SOFTWARE DEVELOPMENT PROCESS**
  2) EVALUATE EXISTING TECHNOLOGY
  3) TRANSFER SUCCESSFUL TECHNOLOGIES INTO THE DEVELOPMENT PROCESS

STRUCTURE OF THE SEL



DEVELOPMENT CHARACTERISTICS

LANGUAGES	FORTRAN (85%) MACROS (15%)
TYPE	SCIENTIFIC GROUND-BASED INTERACTIVE NEAR-REAL-TIME
SIZE	TYPICALLY ~65,000 SLOC (2,000 TO 160,000)
SCHEDULE	16 TO 25 MONTHS (FROM START OF DESIGN TO START OF OPERATIONS)
STAFFING	6 TO 18 PERSONS
COMPUTERS	IBM MAINFRAME (PRIMARILY) VAX-11/780 PDP-11/780

900-PAG-(128*)

SCOPE OF SEL ACTIVITIES (1977—1985)

- COLLECTED DATA FROM MORE THAN 50 PROJECTS
  - OVER 2 MILLION LINES OF CODE PRODUCED
  - OVER 200 DEVELOPERS PARTICIPATED
  - OVER 30,000 FORMS SUBMITTED
- STUDIED ABOUT 50 STATE-OF-THE-ART TECHNOLOGIES
- PRODUCED MANY TOOLS, STANDARDS, AND MODELS FOR USE BY DEVELOPERS
  - RECOMMENDED APPROACH TO SOFTWARE DEVELOPMENT
  - MANAGER'S HANDBOOK FOR SOFTWARE DEVELOPMENT
  - AN APPROACH TO SOFTWARE COST ESTIMATION
  - SOFTWARE TEST AND VERIFICATION PRACTICES

900-PAG-(128*)

One Approach for Evaluating

The Distributed Computing Design System (DCDS) *

- Extended Abstract -

Submitted to:

Tenth Annual Software Engineering Workshop
Scheduled for December 4, 1985

Missile Operations and Data Systems Directorate
Flight Dynamics Division
Goddard Space Flight Center

*This material was presented by L. Baker of TRW

John T. Ellis

TRW Defense Systems Group
213 Wynn Drive
Huntsville, Alabama 35805
(205)830-3326

SEPTEMBER 25, 1985

One Approach For Evaluating the Distributed Computing Design System

DCDS provides an integrated environment to support the life cycle of developing real-time distributed computing systems. The primary focus of DCDS is to significantly increase system reliability and software development productivity, and to minimize schedule and cost risk. DCDS consists of integrated methodologies, languages, and tools to support the life cycle of developing distributed software and systems. Smooth and well-defined transistions from phase to phase, language to language, and tool to tool provide a unique and unified environment. An approach to evaluating DCDS highlights its benefits.

1. DCDS OVERVIEW

Distributed solutions to complex systems require sophisticated tools and techniques for the specification and development of distributed software. In response to this need, TRW has developed the Distributed Computing Design System (DCDS) to provide an integrated environment for the specification and life-cycle development of software and systems, with an emphasis on the development of real-time distributed software. The primary focus of DCDS is to significantly increase system reliability and software development productivity, through the use of disciplined techniques and automated tools. To minimize schedule and cost risk, DCDS offers management visibility into the development process. The development of DCDS is sponsored by the Ballistic Missile Defense Advanced Technology Center (BMDATC).

As illustrated in Figure 1, DCDS consists of integrated methodologies, integrated languages, and an integrated tool set. Following the five methodologies, the user can produce specifications for system requirements, software requirements, distributed architectural designs, detailed module designs, and tests. The five languages support the specific concepts for each of the methodologies, and provide the medium for expressing the requirements, designs, and tests. All five languages use the same constructs and syntax. DCDS formal languages, as opposed to natural languages such as English, can be used without ambiguity - all components of the language are explicitly defined.

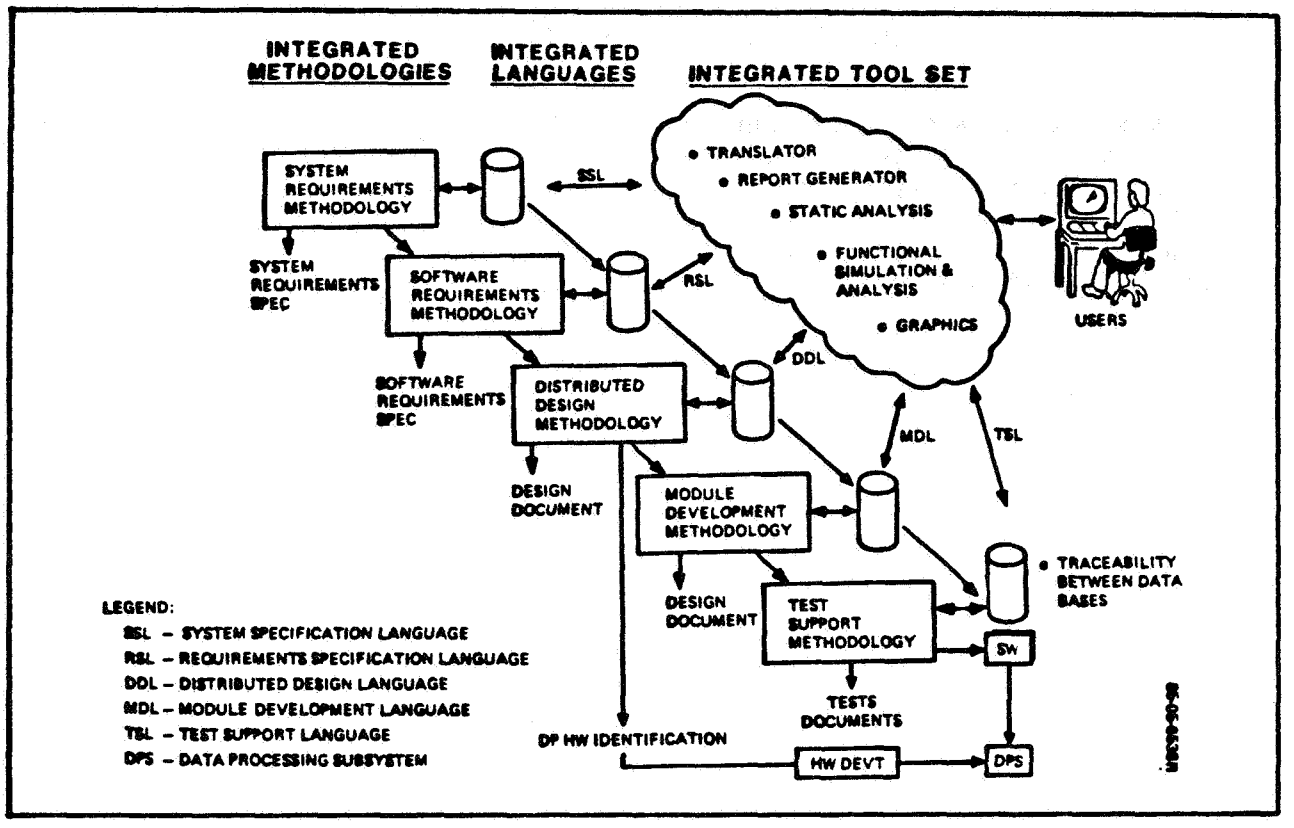


Figure 1. The DCDS Unified Environment

As shown in Figure 1, the user has access to a variety of tools to incrementally define the specification contents, and to check them for completeness and consistency. For each methodology, the tools maintain a data base to store the specification contents. The data base maintains the specification information in a support suitable for automated and thorough analysis. DCDS tools can also support simulation and various types of analyses.

Data extraction tools are used to generate readable listings according to user-defined formats. The listings can be used as working-level documentation, briefing charts, or incorporated into formal specifications. The data base from one methodology is used as a source to initialize the data bases in downstream methodologies, permitting automated traceability between specifications.

THE FIVE DCDS METHODOLOGIES

1. System Requirements Engineering Methodology (SYSREM) for defining and specifying system requirements, with an emphasis on the data processing subsystem.

2. Software Requirements Engineering Methodology (SREM) for defining system software requirements, with an emphasis on stimulus-response behavior.

3. Distributed Design Methodology (DDM) for developing a top-level architectural design for the system software, including distributed design, process design, and task design.

4. Module Development Methodology (MDM) for investigating and selecting algorithms, defining detailed design, and producing units of tested code.

5. Test Support Methodology (TSM) for defining test plans and procedures against requirements, producing an integrated tested system, and recording test results.

THE FIVE DCDS LANGUAGES

1. System Specification Language (SSL) for specifying structured sequences of functions to be performed by the system, inputs/outputs between functions, performance indices for functions, and allocations of functions to subsystems.

2. Requirements Statement Language (RSL) for describing a stimulus-response structure of inputs, outputs, processing, and performance of a DP subsystem in a form which assures unambiguous specifications of explicit, testable software requirements.

3. Distributed Design Language (DDL) for describing the distributed hardware architectures of processing nodes and interconnections, the software architecture, the allocation of processing and data to nodes, and the communication topology.

4. Module Development Language (MDL) for recording detailed designs and algorithms considered and selected for the design.

5. Test Support Language (TSL) for recording tests, their relationship to the requirements, test procedures, and test results.

Figure 2. DCDS Methodologies and Languages

DCDS is used to produce units of tested software, and to identify the data processing hardware. Tools are available to aid in the software process construction activities. The final output (Figure 1) from DCDS is the integrated and tested Data Processing Subsystem.

The DCDS methodologies and languages are defined in Figure 2. Within each methodology, individual steps are provided and are explicit and observable. Acitivites are defined and must be completed prior to each of the major reviews duirng the development life cycle. Well-defined interfaces between the life-cycle phases allow a unified approach for using DCDS. DCDS also provides measurable intermediate milestones for management visibility between the major review points.

DCDS provides a unique and proven capability. First, DCDS is the only integrated environment which addresses the entire life cycle of distributed software development. The techniques are independent of the implementation language, and can be applied effectively to development activities or used as a verification and validation tool. Second, DCDS concepts are based on proven technology - the early results, oriented for software requirements, have been validated, improved, and now extended to support the complete system development life cycle. DCDS is the result of 12 years of research and development, as discussed in IEEE COMPUTER magazine.*

2. DCDS EVALUATION

To gain a better perspective on DCDS and its characteristics, DCDS was compared against three other commerically available products. These three products provide methodologies and/or tools for developing specifications and software. To allow an objective and multi-factored comparative evaluation of the different methodologies and tools, TRW prepared a list of evaluation criteria partitioned into three classes: (1) factors lending credibility to the product, (2) costs of acquiring and using the product, and (3) benefits of the product.

*M. Alford, "SREM At the Age of Eight", IEEE COMPUTER, April 1985, pp. 36-46.

The individual criteria from each of the three classes was assigned a value weight of "high", "medium", and "low". A score of "better", "acceptable", or "deficient" was used to evaluate each product against each evaluation criteria. An explanation of each evaluation criteria and the rationale for each individual score against each product is available.

The results of the evaluation are summarized in Figure 3. Since the evaluation was not performed by an independent organization, the other three products shall remain nameless. However, they do represent well-known products. All the products support an overall acceptable rating, and have been used successfully in major applications. DCDS received an overall higher rating within this evaluation process due to the following discriminating factors:

- Automated traceability across life-cycle phases
- Automated analysis tools
- Documentation support capabilities
- Relatively low cost to acquire and use the product

It is anticipated that the evaluation approach and criteria as outlined in this report could be used by an independent agency for a more in-depth analysis and evaluation of various methodologies and tools. The author wishes to acknowledge Mack Alford and Bob Loshbough of TRW for their extensive technical contribution to the author's summation of DCDS and its evaluation.

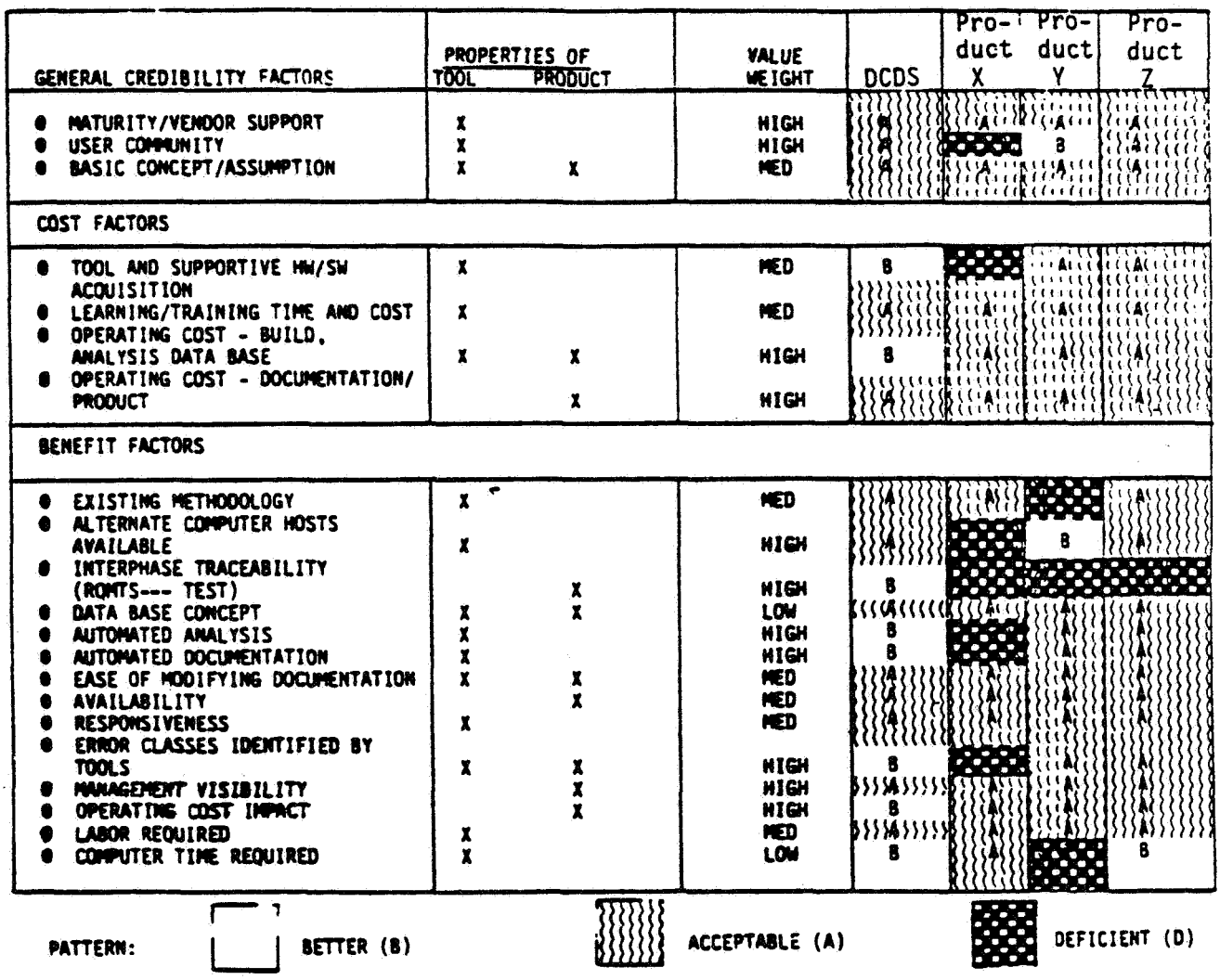


Figure 3. Evaluation Results

THE VIEWGRAPH MATERIALS

for the

J. ELLIS PRESENTATION FOLLOW

TRW Defense
Systems Group
Huntsville Laboratory

TRW

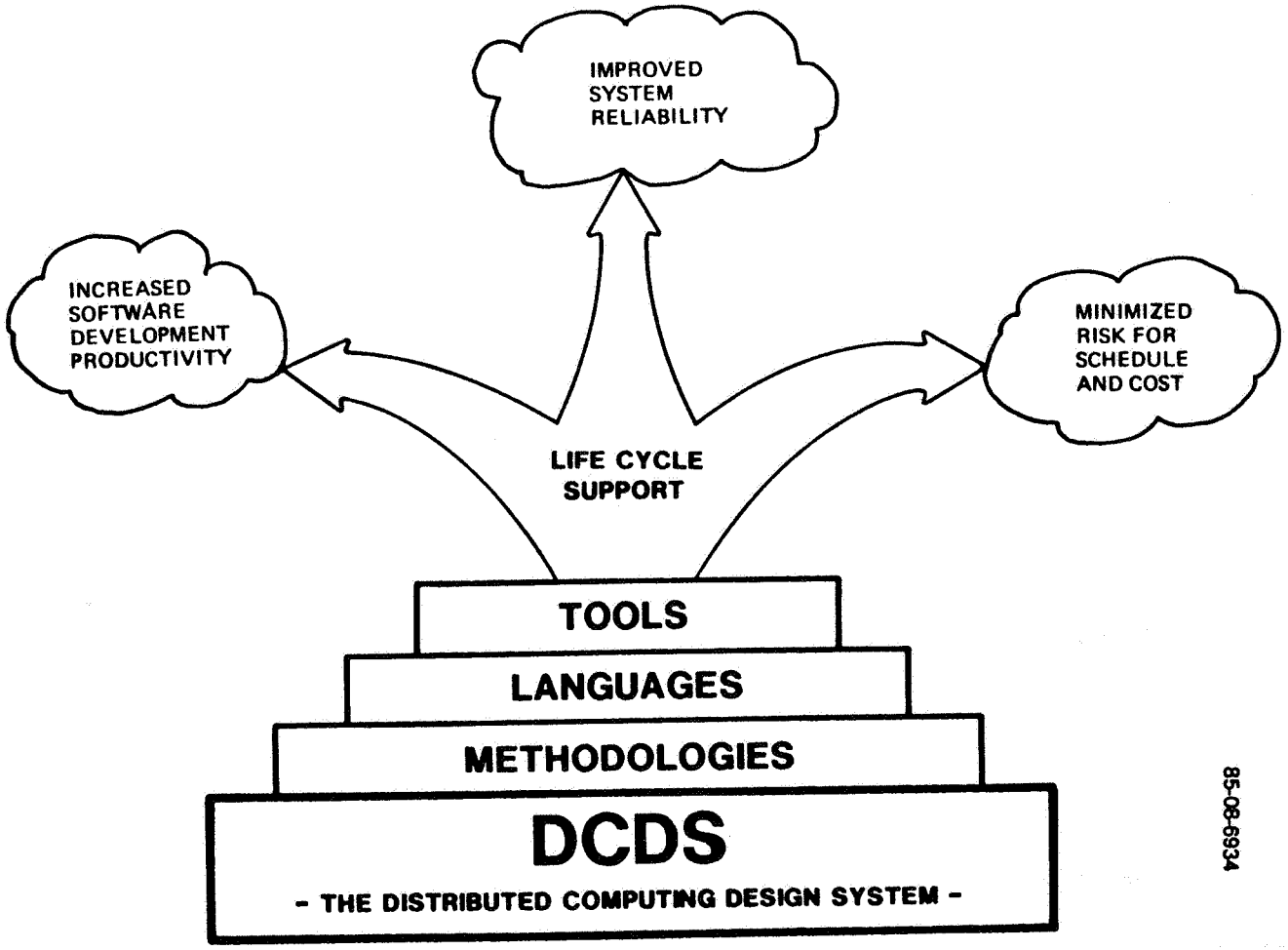
TRW85-169

APPROACHES FOR EVALUATING THE DISTRIBUTED COMPUTING DESIGN SYSTEM (DCDS)

4 December 1985

J. Ellis
TRW

THE DISTRIBUTED COMPUTING DESIGN SYSTEM



85-08-6934

DCDS OBJECTIVES (1981)

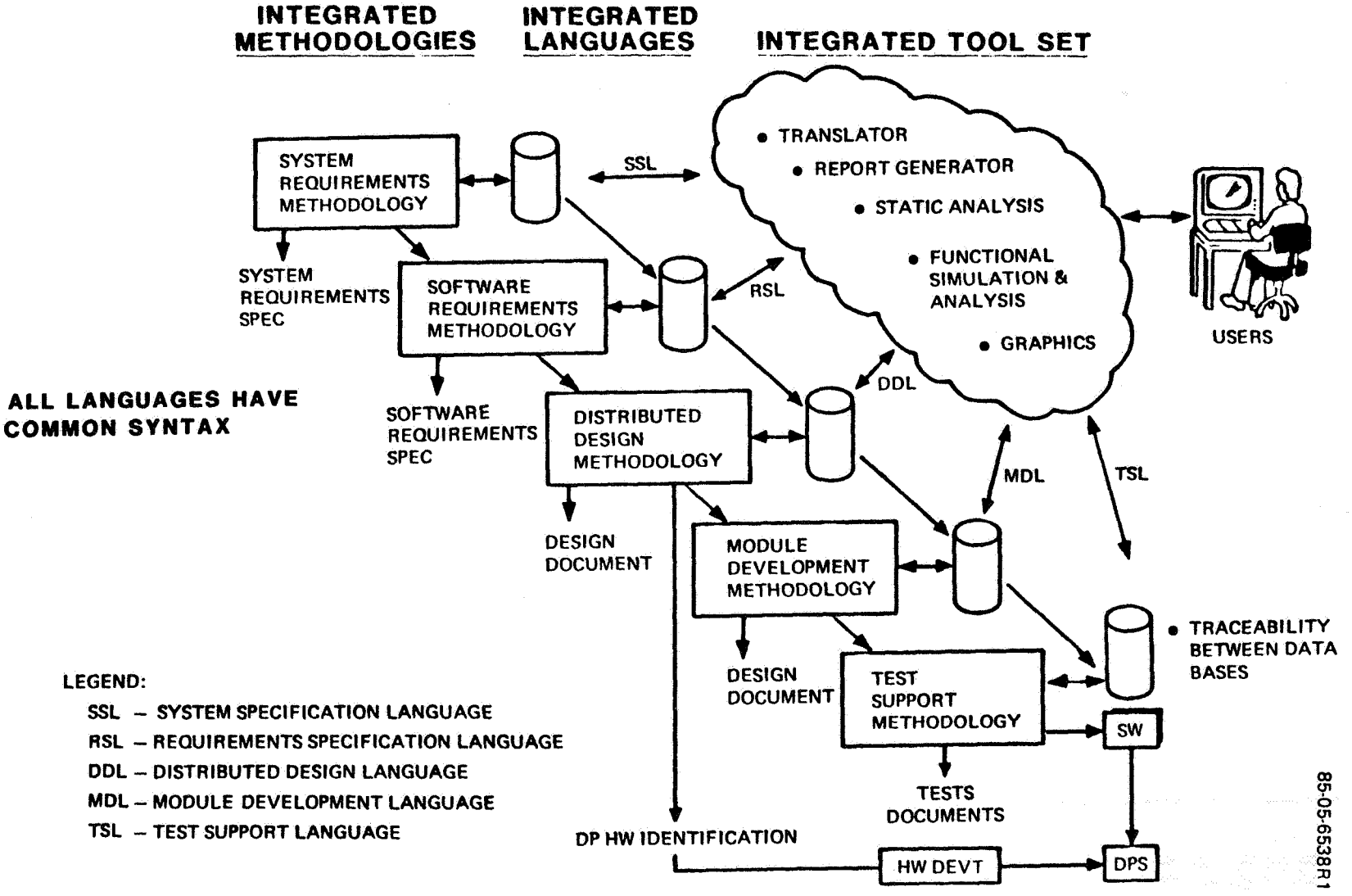
OBJECTIVES

- A SECOND GENERATION SOFTWARE DEVELOPMENT ENVIRONMENT WITH AN INTEGRATED METHODOLOGY
- BASED ON A REQUIREMENTS-DRIVEN METHODOLOGY
  - INTEGRATED MODEL ACROSS ALL DEVELOPMENT PHASES
  - INTEGRATED LANGUAGE FOR REQUIREMENTS, DESIGN, TEST, AND MANAGEMENT
  - AUTOMATED TOOLS EXPLOIT LANGUAGE
    - -- DESIGN ANALYSIS AND SYNTHESIS
    - -- EARLY AND MORE COMPREHENSIVE ERROR DETECTION
  - INTEGRATED METHODOLOGY FOR ALL PHASES
- PRODUCTIVITY INCREASE (200 PERCENT TO 400 PERCENT)
- QUALITY INCREASE (DEFECTS REDUCED TO 1 DEFECT PER 1000 LINES OF CODE)

DCDS SUPPORTING GOALS

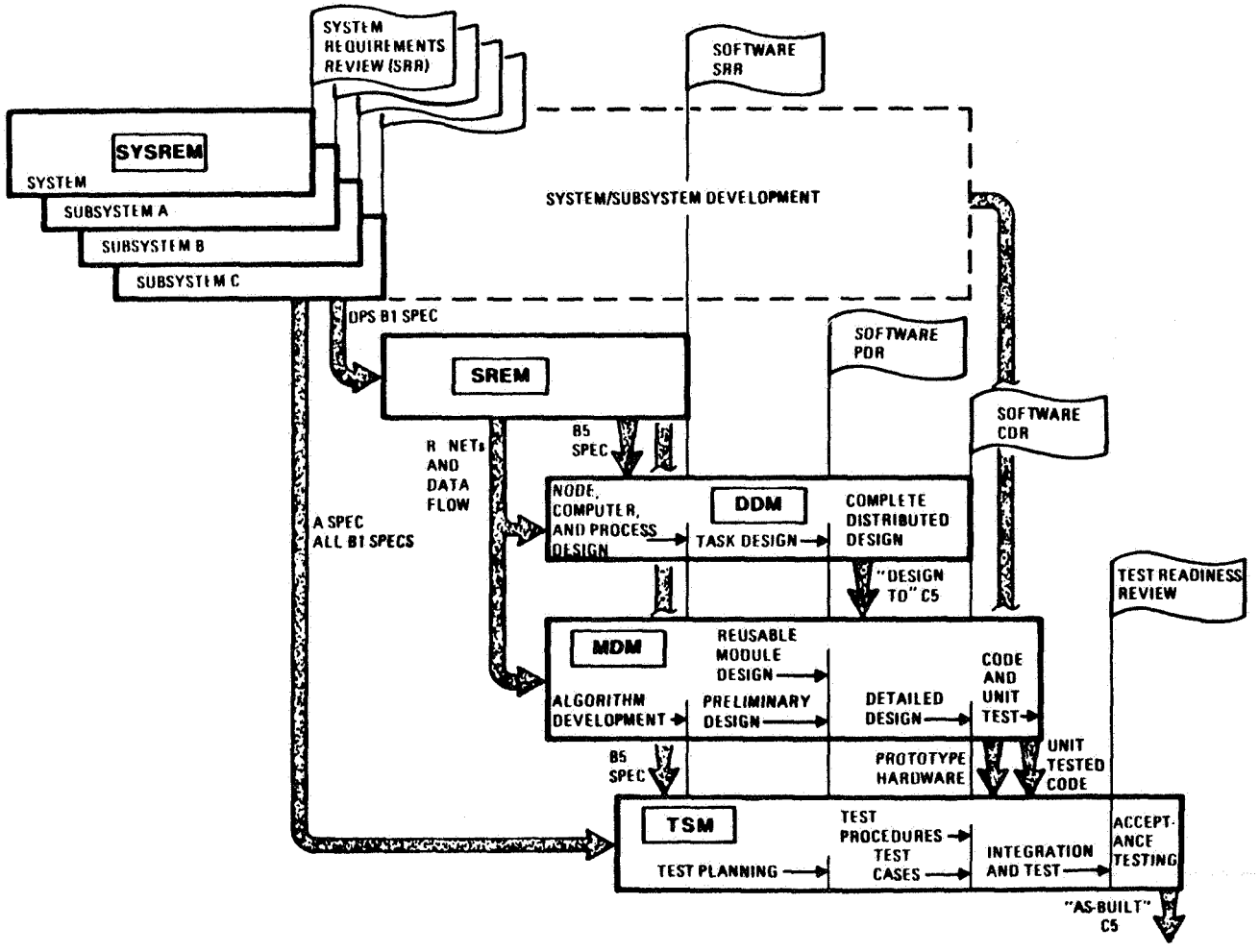
- INTEGRATION OF EFFORTS BETWEEN METHODOLOGIES
- BETTER UNDERSTANDING OF THE PROCESS AND PRODUCTS
  - SYSTEMATIC SEQUENCE OF STEPS AND DECISIONS
  - IMPROVED REPRESENTATION OF THE PRODUCTS IN DATA BASES
  - SEPARATION OF CONCERNS
  - ADDED DISCIPLINE
  - MANAGEMENT VISIBILITY
- EARLY REQUIREMENTS EMPHASIS
  - DISCIPLINED STRUCTURE APPROACH
  - PORTION OF EACH DATA BASE USED TO INITIATE DOWNSTREAM DATA BASES
  - ACHIEVES REAL PAYOFF IN REDUCTION OF DOWNSTREAM ERRORS
- AUTOMATED TOOL SUPPORT
  - MENU-DRIVEN SYSTEM
  - DATA BASE CONSISTENCY COMPLETENESS CHECKING
  - ELECTRONIC FORMS ENTRY
  - DATA FLOW ANALYZER
  - CLUSTERING ANALYZER
  - QUERY SYSTEM (ALSO SUPPORTS DOCUMENTATION)
  - INTERACTIVE GRAPHICS AND PLOTTING SUPPORTS STRUCTURES
  - AUTOMATED UNIT DEVELOPMENT FOLDER
  - PROCESS CONSTRUCTION SYSTEM
- STRONG TRACEABILITY
  - REQUIREMENTS DRIVEN
  - DIRECT TRACEABILITY ACHIEVABLE BETWEEN ELEMENTS IN EACH DATA BASE
  - EASES MODIFICATIONS

DCDS UNIFIED ENVIRONMENT



TRW Defense
Systems Group
Huntsville Laboratory

THE INTEGRATED DCDS METHODOLOGIES SUPPORT THE ENTIRE SOFTWARE DEVELOPMENT CYCLE



HOW DO I EVALUATE

METHODOLOGY/ENVIRONMENT?

QUALITATIVE APPROACH

GENERAL CREDIBILITY FACTORS	PROPERTIES OF TOOL	PROPERTIES OF PRODUCT	VALUE WEIGHT	DCDS	SREM	X	Y	Z
• MATURITY/VENDOR SUPPORT	X		HIGH	A	A	A	A	A
• USER COMMUNITY	X		HIGH	A	A	A	B	A
• BASIC CONCEPT/ASSUMPTION	X	X	MED	A	A	A	A	A
COST FACTORS								
• TOOL AND SUPPORTIVE HW/SW ACQUISITION	X		MED	B	B	A	A	A
• LEARNING/TRAINING TIME AND COST	X		MED	A	A	A	A	A
• OPERATING COST - BUILD, ANALYSIS DATA BASE	X	X	HIGH	B	B	A	A	A
• OPERATING COST - DOCUMENTATION/ PRODUCT		X	HIGH	A	A	A	A	A
BENEFIT FACTORS								
• EXISTING METHODOLOGY	X		MED	A	A	A	A	A
• ALTERNATE COMPUTER HOSTS AVAILABLE	X		HIGH	A	A	A	B	A
• INTERPHASE TRACEABILITY (RQMTS--- TEST)		X	HIGH	B	A	A	A	A
• DATA BASE CONCEPT	X	X	LOW	A	A	A	A	A
• AUTOMATED ANALYSIS	X		HIGH	B	B	A	A	A
• AUTOMATED DOCUMENTATION	X		HIGH	B	B	A	A	A
• EASE OF MODIFYING DOCUMENTATION	X	X	MED	A	A	A	A	A
• AVAILABILITY		X	MED	A	A	A	A	A
• RESPONSIVENESS	X		MED	A	A	A	A	A
• ERROR CLASSES IDENTIFIED BY TOOLS	X	X	HIGH	B	B	A	A	A
• MANAGEMENT VISIBILITY		X	HIGH	A	A	A	A	A
• OPERATING COST IMPACT		X	HIGH	B	B	A	A	A
• LABOR REQUIRED	X		MED	A	A	A	A	A
• COMPUTER TIME REQUIRED	X		LOW	B	B	A	D	B

PATTERN: BETTER (B) ACCEPTABLE (A) DEFICIENT (D)

85-12-7675

QUANTITATIVE-MULTIPLE PROJECT TYPES

METHODOLOGY	SOFTWARE CATEGORY: SIMPLE																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	20	21	21	19	19	24	21	22	20	21	24	20	21	20	16	23	22	21
B	33	34	34	37	34	37	40	37	33	40	37	33	40	33	25	36	37	34
C	23	23	23	20	20	26	23	23	23	23	26	23	23	23	16	23	23	23
D	28	28	28	26	26	30	28	28	28	28	30	28	28	28	22	31	28	27
E	27	27	27	27	25	31	30	31	27	30	31	27	30	27	19	30	31	27
F*	40	39	39	40	40	40	41	39	40	41	40	40	41	40	29	42	39	38
G	22	19	19	20	20	24	23	24	22	23	24	22	23	22	13	26	24	22
H**	42	40	40	41	41	45	44	45	42	44	45	42	44	42	27	47	45	42
I	24	21	21	21	18	26	24	26	24	24	26	24	24	24	17	25	26	22
J	27	33	33	33	30	33	33	33	27	33	33	27	33	27	19	30	33	27
K	35	40	40	38	37	43	41	41	35	41	43	35	41	35	27	40	41	37
L	38	35	35	38	35	40	41	38	38	41	40	38	41	38	26	39	38	36

* SREM
** DCDS

85-12-7676

QUANTITATIVE-MULTIPLE PROJECT TYPES (CONTINUED)

METHODOLOGY	SOFTWARE CATEGORY: MEDIUM																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	1	1	1	-1	0	2	-1	1	1	-1	2	1	-1	1	1	1	1	2
B	14	14	14	17	15	15	18	16	14	18	15	14	18	14	10	14	16	15
C	4	3	3	0	1	4	1	2	4	1	4	4	1	4	1	1	2	4
D	9	8	8	6	7	8	6	7	9	6	8	9	6	9	7	9	7	8
E	8	7	7	7	6	9	8	10	8	8	9	8	8	8	4	8	10	8
F*	21	19	19	20	21	18	19	18	21	19	18	21	19	21	14	20	18	19
G	3	-1	-1	0	1	2	1	3	3	1	2	3	1	3	-2	4	3	3
H**	23	20	20	21	22	23	22	24	23	22	23	23	22	23	12	25	24	23
I	5	1	1	1	1	4	2	5	5	2	4	5	2	5	2	3	5	3
J	8	13	13	13	11	11	11	12	8	11	11	8	11	8	4	8	12	8
K	16	20	20	18	18	21	19	20	16	19	21	16	19	16	12	18	20	18
L	19	15	15	18	16	18	19	17	19	19	18	19	19	19	11	17	17	17

* SREM
** DCDS

QUANTITATIVE-MULTIPLE PROJECT TYPES (CONTINUED)

METHODOLOGY	SOFTWARE CATEGORY: REAL-TIME																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	-18	-19	-19	-21	-19	-20	-23	-20	-18	-23	-20	-18	-23	-18	-14	-21	-20	-17
B	-5	-6	-6	-3	-4	-7	-4	-5	-5	-4	-7	-5	-4	-5	-5	-8	-5	-4
C	-15	-17	-17	-20	-18	-18	-21	-19	-15	-21	-18	-15	-21	-15	-14	-21	-19	-15
D	-10	-12	-12	-14	-12	-14	-16	-14	-10	-16	-14	-10	-16	-10	-8	-13	-14	-11
E	-11	-13	-13	-13	-13	-13	-14	-11	-11	-14	-13	-11	-14	-11	-11	-14	-11	-11
F *	2	-1	-1	0	2	-4	-3	-3	2	-3	-4	2	-3	2	-1	-2	-3	0
G	-16	-21	-21	-20	-18	-20	-21	-18	-16	-21	-20	-16	-21	-16	-17	-18	-18	-16
H **	4	0	0	1	3	1	0	3	4	0	1	4	0	4	-3	3	3	4
I	-14	-19	-19	-19	-20	-18	-20	-16	-14	-20	-18	-14	-20	-14	-13	-19	-16	-16
J	-11	-7	-7	-7	-8	-11	-11	-9	-11	-11	-11	-11	-11	-11	-11	-14	-9	-11
K	-3	0	0	-2	-1	-1	-3	-1	-3	-3	-1	-3	-3	-3	-3	-4	-1	-1
L	0	-5	-5	-2	-3	-4	-3	-4	0	-3	-4	0	-3	0	-4	-5	-4	-2

85-12-7677

* SREM
** DCDS

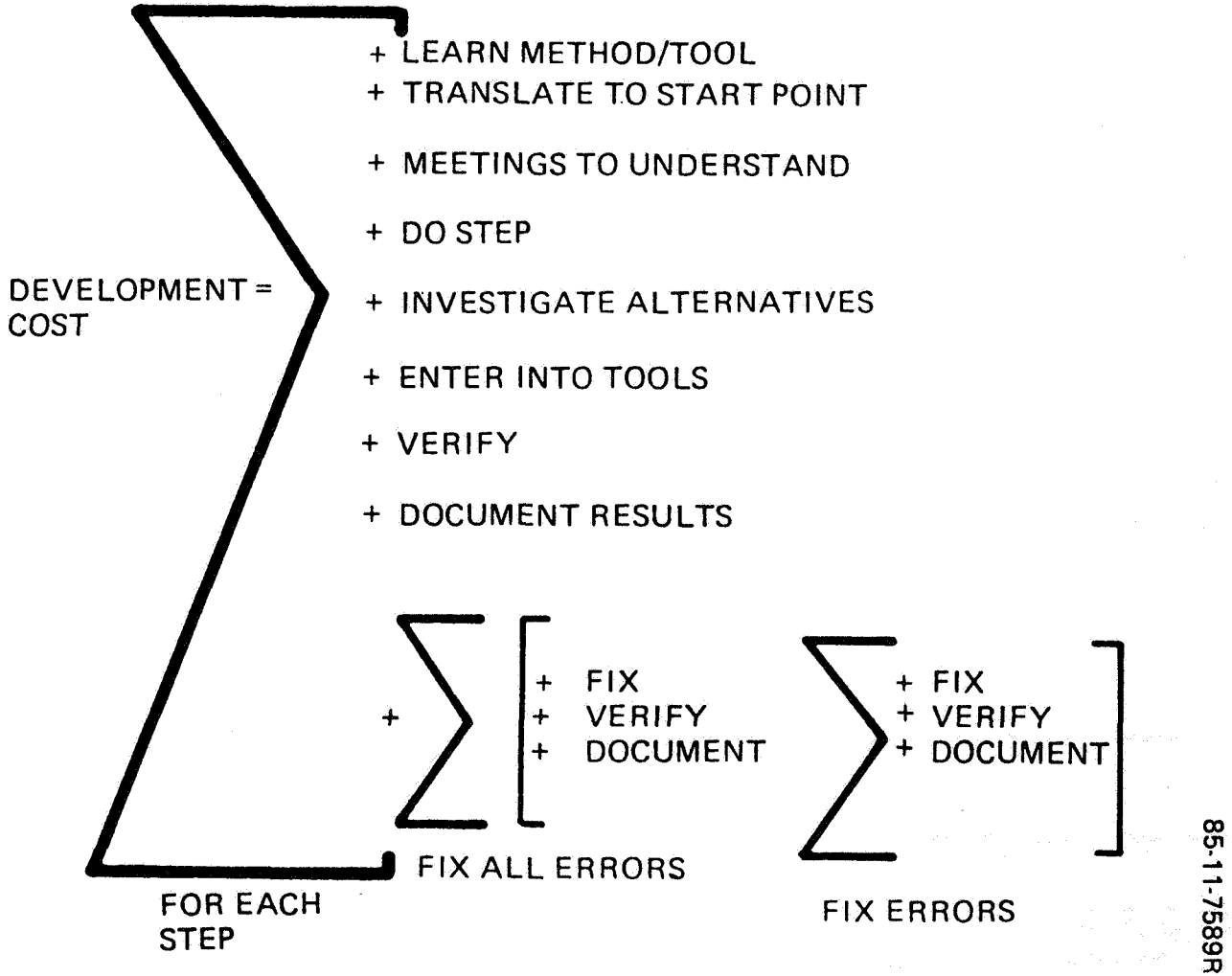
QUANTITATIVE-MULTIPLE PROJECT TYPES (CONTINUED)

METHODOLOGY	SOFTWARE CATEGORY: MAN-RATED																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	-37	-30	-30	-41	-38	-42	-45	-41	-37	-45	-42	-37	-45	-37	-29	-43	-41	-36
B	-24	-26	-26	-23	-23	-29	-26	-26	-24	-26	-29	-24	-26	-24	-20	-30	-26	-23
C	-34	-37	-37	-40	-37	-40	-43	-40	-34	-43	-40	-34	-43	-34	-29	-43	-40	-34
D	-29	-32	-32	-34	-31	-36	-38	-35	-29	-38	-36	-29	-38	-29	-23	-35	-35	-30
E	-30	-33	-33	-33	-32	-35	-36	-32	-30	-36	-35	-30	-36	-30	-26	-36	-32	-30
F *	-17	-21	-21	-20	-17	-26	-25	-24	-17	-25	-26	-17	-25	-17	-16	-24	-24	-19
G	-35	-41	-41	-40	-37	-42	-43	-39	-35	-43	-42	-35	-43	-35	-32	-40	-39	-35
H **	-15	-20	-20	-19	-16	-21	-22	-18	-15	-22	-21	-15	-22	-15	-18	-19	-18	-15
I	-33	-39	-39	-39	-39	-40	-42	-37	-33	-42	-40	-33	-42	-33	-28	-41	-37	-35
J	-30	-27	-27	-27	-27	-33	-33	-30	-30	-33	-33	-30	-33	-30	-26	-36	-30	-30
K	-22	-20	-20	-22	-20	-23	-25	-22	-22	-25	-23	-22	-25	-22	-18	-26	-22	-20
L	-19	-25	-25	-22	-22	-26	-25	-25	-19	-25	-26	-19	-25	-19	-19	-27	-25	-21

85-12-7679

* SREM
** DCDS

COST MODEL APPROACH FOR EVALUATING DCDS



85-11-7589R

TRW Defense
Systems Group
Huntsville Laboratory

COST MODEL-SAMPLE CALCULATIONS

COST MODEL

DEVELOPMENT = COST

- + LEARN METHOD/TOOL
- + TRANSLATE TO START POINT
- + MEETINGS TO UNDERSTAND
- + DO STEP
- + INVESTIGATE ALTERNATIVES
- + ENTER INTO TOOLS
- + VERIFY
- + DOCUMENT RESULTS

FOR EACH STEP

+ Σ [+ FIX + VERIFY + DOCUMENT]

FIX ALL ERRORS

Σ [+ FIX + VERIFY + DOCUMENT]

FIX ERRORS

EXAMPLE CALCULATIONS*

	MANUAL METHOD	GOOD MANUAL VERIFICATION	METHODOLOGY AND AUTOMATED TOOLS (DCDS)
+ LEARN METHOD/TOOL	-	-	-
+ TRANSLATE TO START POINT	15	15	5
+ MEETINGS TO UNDERSTAND	10	10	5
+ DO STEP	30	30	25 (15?)
+ INVESTIGATE ALTERNATIVES	10	10	10
+ ENTER INTO TOOLS	0	0	5
+ VERIFY	5	25	5
+ DOCUMENT RESULTS	30	30	5
	100%	120%	60% (50%)
FIX ERRORS	200%	100%	20%
	300%	220%	80%

*Estimates

FULL COST VERSUS SIZE

SIZE	SMALL	MEDIUM	LARGE
COST	$0.5M	$5M	$40M
NOMINAL REQUIREMENTS (3.6%)	$18K	$180K	$1.8M
REAL REQUIREMENTS			
• MANUAL (300%)	$54K	$540K	$5.4M
• GOOD MANUAL VERIFICATION (220%)	$40K	$400K	$4.0M
• DCDS LABOR (80%)	$14K	$140K	$1.4M
+			
(LEARN METHODOLOGY)	$20K	$50K	$700K
+			
(TOOL COST (μVAX)	$20K	$25K	$50K

CONCLUSIONS

- QUALITATIVE EVALUATION APPROACHES DON'T ADDRESS THE "REAL" BOTTOM LINE COST
- (DITTO QUANTITATIVE APPROACHES BASED ON THEM)
- DATA IS NOT BEING KEPT ON THE RIGHT COSTS TO SUPPORT EVALUATION
- SOME TRENDS ARE CLEAR
  - INTEGRATED METHODS/TOOLS REDUCE INTERPHASE COSTS/ ERRORS
  - VERIFICATION REDUCES ERRORS
  - AUTOMATED VERIFICATION REDUCES COSTS AND ERRORS
  - AUTOMATED DOCUMENTATION REDUCES COST
  - LEARNING AND TOOL COSTS ARE NON-LINEAR

PANEL #4

EXPERIMENTS WITH ADA

D. Roy, Century Computing Inc.
M. McClimens, Mitre Corporation
W. Agresti, Computer Sciences Corporation

Daniel M Roy
Century Computing, Inc.

Abstract

A 1200 line Ada source code project simulating the most basic functions of an operations control center was developed for code 511. We selected George Cherry's Process Abstraction Methodology for Embedded Large Applications (PAMELA) and DEC's Ada Compilation System (ACS) under VAX/VMS to build the software from requirements to acceptance test. The system runs faster than its FORTRAN implementation and was produced on schedule and under budget with an overall productivity in excess of 30 lines of Ada source code per day.

Author current address:
Century Computing Incorporated,
8101 Sandy Spring Rd.
Laurel, Md. 20707
(301) 953 3330

1 BACKGROUND

The Multi-satellite Operations Control Center branch (MSOCC), code 511, has embarked on an effort to improve productivity in the development and maintenance of Operations Control Center (OCC) systems. This productivity effort is addressing a range of issues from equipment and facilities improvements to the development and acquisition of tools and the training of personnel.

Century Computing's previous work on MSOCC's productivity improvement program, identified the Ada language as a promising technology, and recommended evaluating Ada on a small "pilot project" related to MSOCC applications [Century-84].

2 PURPOSE OF THE STUDY

The objective of the study was to evaluate the applicability of Ada and its development environment for MSOCC. Metrics were identified for this evaluation, along with an approach to collecting the data required for these metrics. The evaluation was based on using Ada to re-develop from scratch a small scale, real-time project related to MSOCC applications: an Application Processor (AP) benchmark system.

3 DESCRIPTION OF THE AP BENCHMARK SYSTEM

An AP is a computer that performs the functions required by a satellite operations control center. The AP Benchmark system was previously developed to simulate the characteristics of a typical MSOCC's AP software system [CSC/SD-83]. Like most AP software, the Benchmark was developed in FORTRAN with some supporting assembly language.

The AP Benchmark software simulates the following AP functions:

- Reads a telemetry data stream from tape - meters the frequency of tape reads to simulate various data rates.
- Decommutates the telemetry data.
- Performs some limit checking on the data.
- Displays some of the telemetry data on CRT screens.
- Simulates the history and attitude data recording processes.
- Simulates strip chart recorders and associated functions.
- Gathers statistics on the above process and generates reports.

4 DESCRIPTION OF THE ADA PILOT PROJECT

The pilot project began with a reverse engineering phase to construct requirements from the existing FORTRAN code. Then, a staged approach was used to develop the software, using Ada for all project phases:

- We used Ada as a Data Definition Language to produce a data dictionary during the requirements analysis phase. A special package, the "TBD" package (fig. 1) helped in the top down design of the data structure.

- We used Ada as a Program Specification Language very early in the project and easily prototyped the data flow. The Process Abstraction Methodology tools [Cherry-84] (see appendix B) produced a tasking model that worked at first try (fig. 2a and b). The preliminary and detailed design templates we created (fig. 3a and b) proved to be very useful for enforcing good practices.

- We used Ada as a Program Design Language [IEEE-990] (fig. 4) and refined the PDL into detailed Ada code in the usual staged manner. The DCL tools and templates for Ada construct, developed at the onset of the project, had a dramatic impact on productivity and code consistency.

- We enjoyed the elegance of Ada as an implementation language and used most of its features (attributes, generics, exception handlers, etc.)

- Full assessment of the DEC ACS tools was beyond the scope of this study, but we appreciated the built-in configuration control tool, the automatic recompilation system and the symbolic debugger [DEC-85].

The total re-development approach we followed (from requirements to final tests) led us to believe that we could produce a still more efficient design. Actually, the PAMELA methodology design rules detected several extraneous tasks in the current AP benchmark model, but we decided to respect the existing global structure as the model was built to represent the typical CPU load of an actual OCC.

```
Package TBD is          --| Decision deferral package --*
--| Raises:
--|     None
--| Overview:   --| Purpose:
--|     This is an improvement over Intermetrics' TBD package and IEEE 990
--|     recommendations about decision deferral techniques.
--| Effects:    --| Description:
--|     The distinction is clarified between types, variables and values.
--|     The naming is more consistent (enum_i, component_i ...) and more
--|             readable (scalar_variable intead of scalarValue)
--|     There are more definitions (enum_type, record_type)
--|     Better compatibility with BYRON (or search utility processing)
--| Requires:   --| Assumptions:
--|     Please only "WITH" this package.  By systematically specifying
--|     "TBD.x" items, it is easier to assess the stage of development of
--|     a compilation unit.
--| Notes:
--|    Change log:
--| Daniel Roy   9-AUG-1985     Baseline
--|                                                                      --|
    subtype scalar_type is integer range integer'first .. integer'last; --|
    scalar_variable : scalar_type;                                      --|

    type access_type is access integer;                                 --|
    access_variable : access_type;                                      --|
                                                                        --|
    type record_type is record                                          --|
        component_1 : integer := 0;                                     --|
        component_2 : integer := 0;                                     --|
        component_i : integer := 0;                                     --|
        component_p : integer := 0;                                     --|
        component_n : integer := 0;                                     --|
    end record;                                                         --|
    record_variable : record_type;                                      --|
                                                                        --|
--      Inspired by IBM PDL stuff
    Condition,CD : Boolean := true;                                     --|
                                                                        --|
--      Queues services
    type queue_type is array (array_index_type) of integer;             --|
    type queue_ptr_type is access queue_type;                           --|
--
end TBD;        --| --*
```

Fig. 1: Excerpt from the TBD package

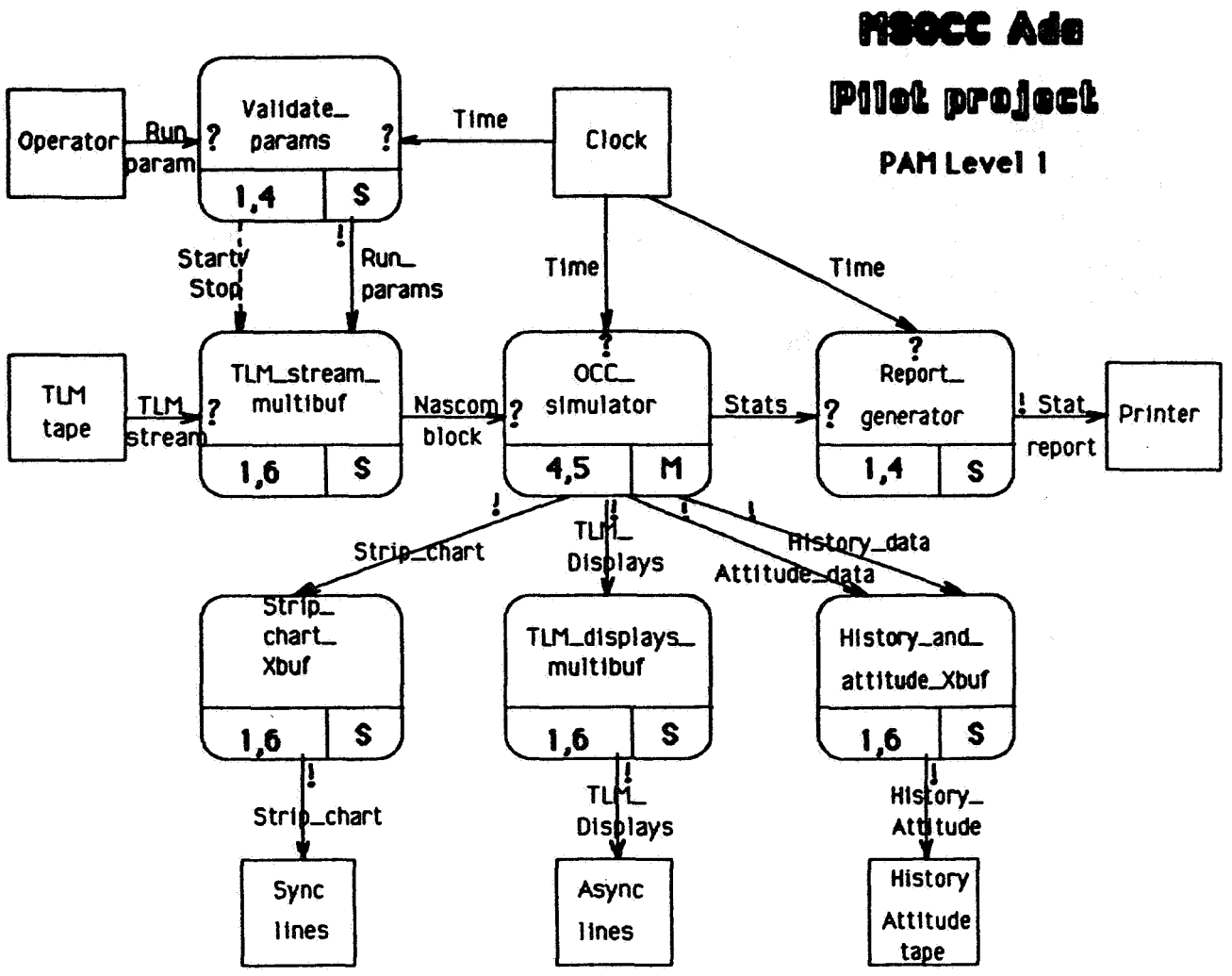


Figure 2a: PAM decomposition level 1

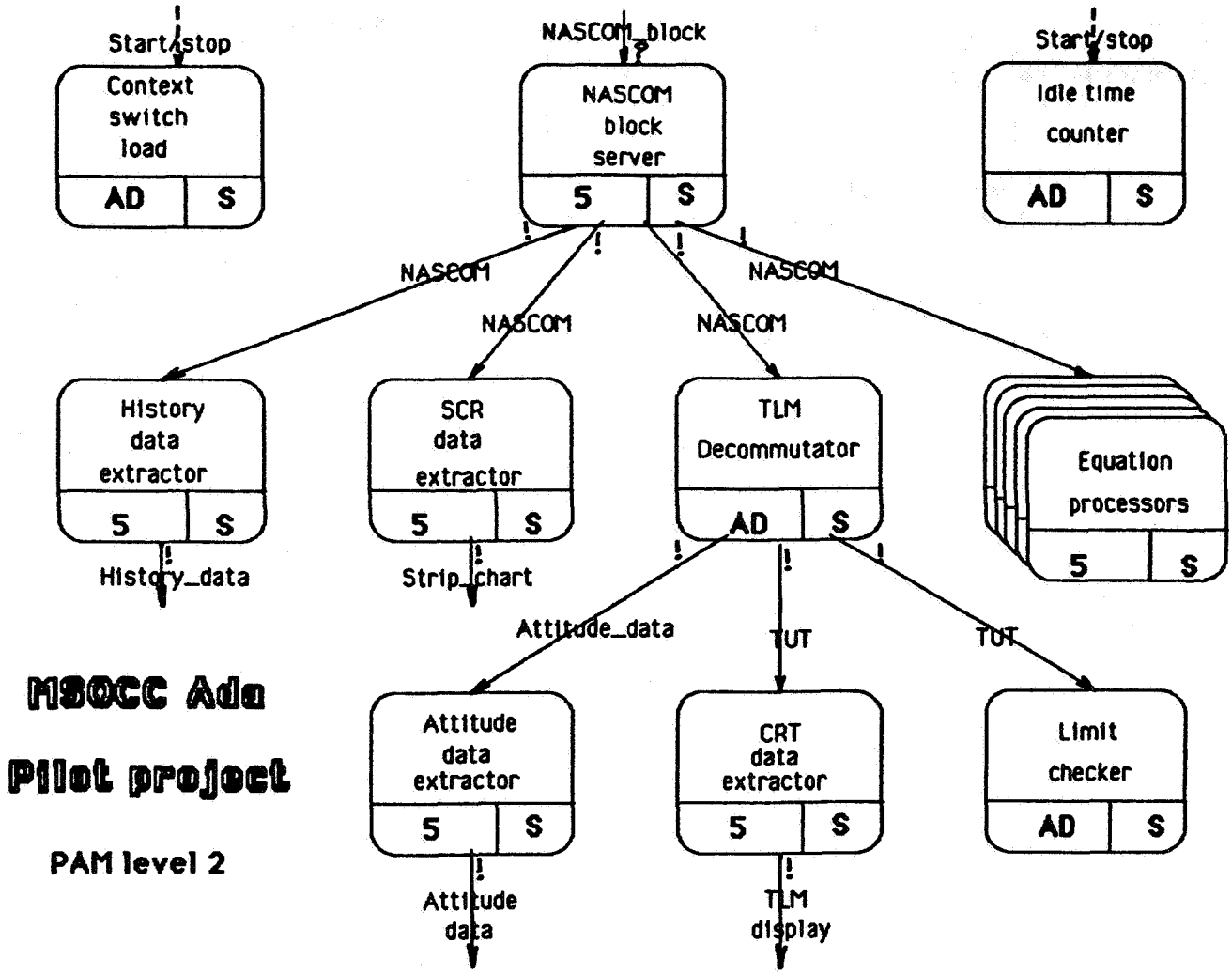


Figure 2b: PAM decomposition level 2

```
--
    procedure P (       --| synopsis --*
        param_1 : IN some_type := some_constant ; --| description --*
        param_n : OUT some_type                   --| description --*
        ) ;                              --| --*
```

Fig. 3a: Preliminary design template for procedure (proc spec)

```
    separate ( )        --| --*
    procedure body P (  --| Short synopsis.  Must be the same as in body. --*
        param_1 : IN some_type := some_constant ; --| description --*
        param_n : OUT some_type                   --| description --*
        ) is                             --| --*
--|
-- ****** Cut and paste from specification.  Use Gold D for rest of DOC. ******
--
-- Packages

-- types

-- subtypes

-- records

-- variables

-- functions

-- procedures

-- separate clauses

    begin       --| --*
        null;
    end P ;     --| --*
```

Fig. 3b: Detailed design template for a procedure (proc body)

```
package body user_interface is          --| Isolate user interface --*
--
    function inquire_int (              --| Emulate DCL verb for integers --*
        prompt : string         --| --*
        ) return inquired_var_type is   --| --*
        inquired_var : inquired_var_type ;      --* The variable we'll return
    --
    begin       --| inquire_int --*
        --* Displays "prompt (min..max): "
        for try in 1.. max_nr_errors loop       --* until good value or else
    --
            begin       --* <<exception_block>>
                --* Get unconstrained value
                --* Validate and translate unconstrained value
                return inquired_var ;   --| --*
        --
            exception   --* recoverable exception when invalid input
                when data_error | constraint_error =>   --*
                    --* display "try again" message
            --| end exception   --*
            --
            end ;       --* <<exception_block>>
        end loop;       --* until good value or else
    --
        exception       --* catch all handler
        --
            when others =>      --*
                raise;  --*
    end inquire_int ;   --| --*
```

Fig. 4: PDL extracted from code by PDL tool

5 RESULTS SUMMARY

Some of the objectives of the evaluation were to determine what is required to train software engineers to use Ada, to define adequate metrics to measure productivity and quality gains and to assess the current Ada development environment.

5.1 Training

We found that Ada is sufficiently complex that we kept learning throughout the pilot project, and even beyond. We also found that none of the standard training devices (seminars, books, computer aided instruction) could alone address the broad range of issues that really are at the heart of the problem:

In the Ada era, a comprehensive education in the software engineering principles that form the basis of the Ada culture must replace ad-hoc training in the syntactic recipes of a language.

That is why we recommend a variety of continuous education measures in our report: Assuming adequate familiarization with modern software engineering practices, at least 4 person-week is the minimum minimorum training time. This time includes teaching a methodology adapted to Ada and 50% hands on experiments under the supervision of an expert.

5.2 Metrics And Data Collection Approach

After a review of established research in the areas of metrics and data collection, a brief paper outlining the metrics approach was issued. The metrics work of the NASA Software Engineering Laboratory was the key input [McGarry-82].

Simple DCL tools were built to gather the metrics data and comprehensive logs of errors, problems and interesting solutions were maintained on-line and are part of the deliverables.

5.3 Productivity

Our productivity during the seven weeks coding period averaged 32 lines of Ada source code (LOC) per day and nearly 130 lines of text (LOT) per day (includes embedded documentation, comments and blank lines). We experienced a low point of 10 LOC per day at the beginning of the coding phase, and reached a peak of 90 LOC and 370 LOT per day during the final week (fig. 5). Averaged over the whole 18 weeks of development (including reverse engineering with DeMarco before PAM, tools development, two seminars, compilers installation, etc.) productivity still remains above 13 LOC and 50 LOT per day.

Although formal verification techniques were not employed, intense validation testing discovered two errors, both due to subtle differences between our implementation and its FORTRAN precursor. A detailed log of all the problems we had at various phases of the implementation was kept on-line.

Those productivity and quality results are interesting data points, but they must be taken with the following caveat:

- We were re-implementing a working system.
- Our deliverables did not include all standard documentation.
- We did not produce a performance prediction study.
- We did not perform a deadlock avoidance study.
- Unit testing was not up to the standards we would have applied to an operational system.
- We sometimes abandoned early our search for better solutions.
- When a problem arose we did not always research why.
- More than 90% of the code was written by a single individual.

On the other hand, we wrote much more scaffolding and experimental ("throw away") software than a normal project would require.

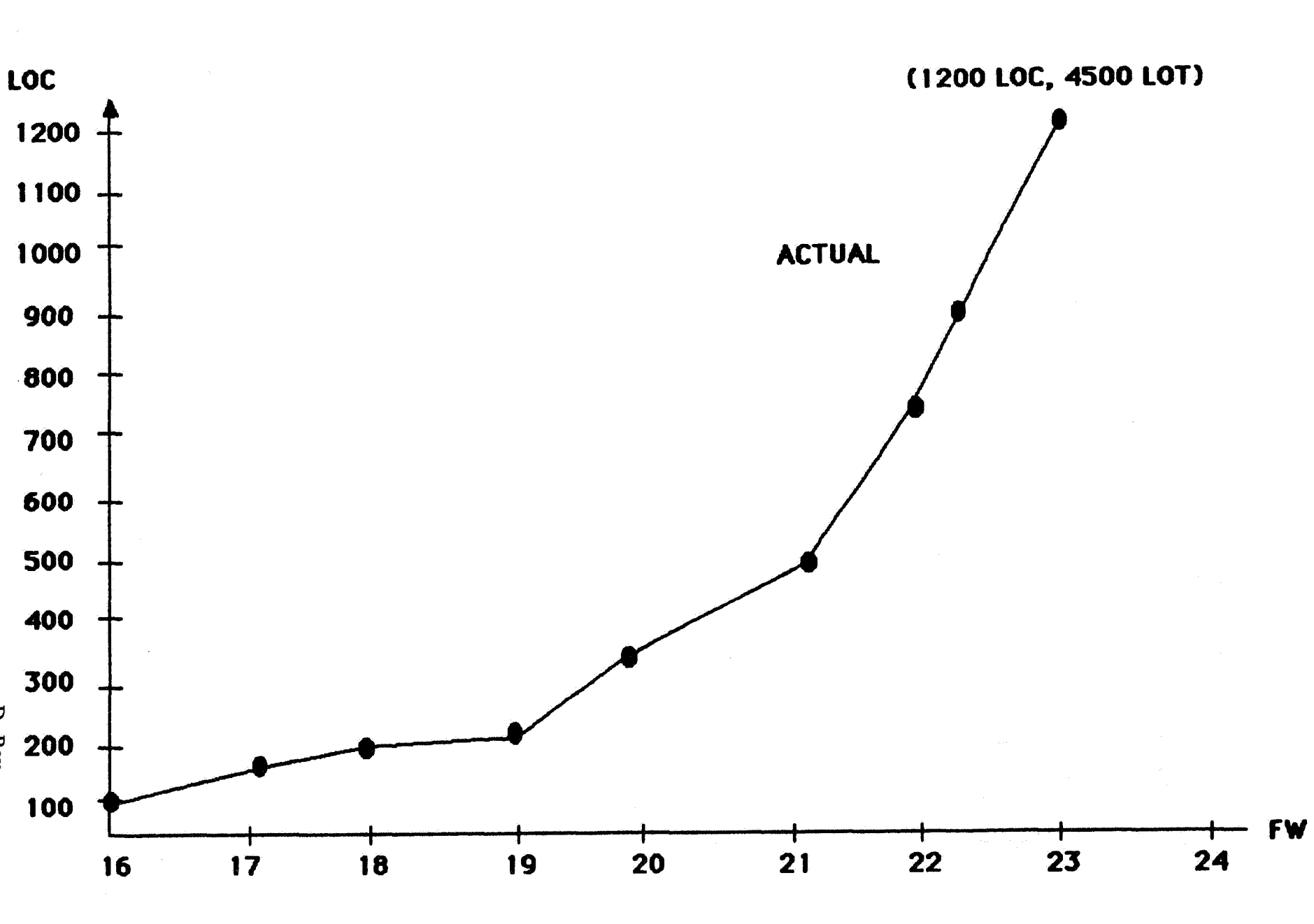

ADA PILOT PROJECT LINES OF SOURCE CODE
LOC
(1200 LOC, 4500 LOT)
ACTUAL
1200
1100
1000
900
800
700
600
500
400
300
200
100
16
17
18
19
20
21
22
23
24
FW

Figure 5

5.4 Compilers Experience

We first used Century's NYU Courant Institute Ada interpreter on our VAX 11/750 for training and tools development. We quickly became frustrated with this system.

Thanks to NASA's cooperation, we got some exposure to the Telesoft compilers and the DEC Ada Compilation System (ACS).

We then installed Softech's Ada Language System (ALS) on another NASA VAX. Our conclusion was that the current performance problems of the ALS made it unsuitable in light of our schedule constraints.

In the end we were granted access to code 520's test version of DEC's Ada Compilation System (ACS) under VMS 4.1 which we used to develop most of the pilot project. It is clear to us that the ACS made the timely completion of our project possible and that, in general, the quality of the development environment significantly impacts software development productivity.

As delivered, the Ada pilot project features about the same number of statements as its FORTRAN precursor (about 1200) but is larger in the number of lines of text (4,500 vs 2,000). Image sizes are comparable (about 170 kbytes for Ada vs about 200 kbytes for FORTRAN).

Even though it is difficult to compare run time performance on the very different computer environments we used, **our preliminary results seem to indicate that the Ada code runs faster than its FORTRAN counterpart.** We suspect that our good results may be due to the fact that some data elements could be directly addressed in Ada and not in FORTRAN. Nevertheless, this is a completely unexpected result that is even contrary to popular belief. We think it speaks for the high quality of DEC's ACS and the adequacy of the chosen methodology (the Process Abstraction Methodology for Embedded Large Applications).

6 CONCLUSIONS

Ada is clearly a step forward in the software industry's search for a better programming language for real-time and embedded systems. Ada also represents significant advancements in the field of practical programming language development.

Furthermore, the Ada Programming Support Environment (APSE) and the Software Technology for Adaptable Reliable Systems (STARS) initiative will support the language with an impressive set of evolving tools.

But even with these features, it is possible to develop poor software in Ada. In fact, packaging, generics, multitasking and, above all, representation clauses (that allow direct access to the hardware!) will have to be closely controlled by competent project managers because these features are powerful, hence dangerous. Moreover, those powerful features provide another dimension of design decision. We

feel that a methodology that helps the software engineer allocate function and data structures to packages and tasks is necessary.

Ada should prove to be an excellent tool in the hands of competent and properly trained software developers. It will not be a panacea, compensating for inadequate methods or training, but it will be beneficial if properly applied.

In that context, we make the following predictions relative to the future of Ada:

1. The momentum of the Department of Defense will make Ada a reality. The last time that DoD backed a language (COBOL), the language became, and still is, the most popular in the world.

2. **There will be major false starts in the use of Ada, especially when the aerospace contractors tackle large projects with newly trained programmers.** Ada itself will become the focus of these projects, leaving the target application in second place.

3. The "reality" of Ada will be delayed due to the immaturity of the compiler technology, expense of computer resources, and the training problem.

4. There will be major difficulties at both ends of the programmer competency scale. Many of the brightest programmers will tend to produce overly complex designs, using every possible feature of the language; the application itself becoming a side issue. Many of the less competent programmers will never really understand the Ada technology.

5. Programmer productivity will decrease (relative to conventional languages) before it eventually increases.

6. Universities will eventually produce proficient Ada software engineers, using the language as a basis for teaching all the traditional computer science courses. (This day is getting near. We recently polled area universities and found Ada present in every computer science curriculum.)

7 A FINAL NOTE

In July 1985, following the recommendation of the APSE Beta Test Site Team headed by Dr. McKay (University of Houston at Clear Lake), NASA officially adopted Ada as the language of choice for all flight software of the space station program.

APPENDIX A

BIBLIOGRAPHY

[Century-84] Century Computing Inc., "Software Tools and Methodology Study for NASA MSOCC", Laurel, Md., June 1984.

[Cherry-84] George W. Cherry, "Advanced Software Engineering with Ada", Seminar notes , 1984.

[Cherry-85] George W. Cherry, "The PAMELA (TM) Methodology, A Process-oriented Software Development Method for Ada.", To be published.

[CSC/SD-83] Computer Science Corporation, "Gamma Ray Observatory Era Application Processor Benchmark User's Guide", Update 1, Doc. No. CSC/SD-83/6101UDI, January 1984.

[DEC-85] Digital Equipment Corporation, "Developing Ada Programs On VAX VMS", February 1985.

[IEEE-990] IEEE working group on Ada PDL (990), "Ada PDL draft recommended practice", 5 March 1985.

[McGarry-82] Frank McGarry et al., "Guide to Data Collection", SEL-81-101, NASA GSFC, August 1982.

[Methodman-82] Peter Freeman, Anthony Wasserman, "Software Development Methodologies and Ada", National Technical Information Service, ADA 123-710, November 1982.

APPENDIX B

THE PROCESS ABSTRACTION METHODOLOGY

"The Process Abstraction Methodology for Embedded Large Applications (PAMELA or PAM for short) is a real-time software development method which takes full advantage of Ada's features of type abstraction, process abstraction, exception handling, top-down separate compilation, and bottom-up separate compilation.

Because the PAMELA method recognizes that abstract processes as well as abstract data types are ideal modules for programming in the large, the method is process-oriented as well as object-oriented.

The method is primarily a top-down, outside-in method; but it allows and encourages the bottom-up generation or incorporation of software components (library units).

The PAMELA method contains guidelines to ensure that program units are reusable or portable or both reusable and portable. It also contains guidelines to ensure superior real-time performance (for example, guidelines to ensure that the minimum number of necessary tasks are defined)." [Cherry-85]

"The process abstraction methodology (PAM) is based on the concept of a hierarchical structure of processes. The process as a data transforming element and data flow as a connection link between processes are central concepts in this method." [Cherry-84]

At first glance, the PAMELA methodology "process graphs" (fig. 2a and 2b) look very much like DeMarco's Data Flow Diagrams. The major difference however, is that in any data driven methodology, there is no apparent synchronization between the processes nor any explicit representation of the synchronization between the flow of data and the processes. In a process graph, the processes communicate by the Ada rendez-vous mechanism. Because the concepts of data flow and task to task synchronization are part of the semantics of the Ada rendez-vous, PAM's process graphs overcome one of the major limitations of data flow diagrams for real-time applications. This makes PAMELA applicable to the requirements analysis phase. Most importantly, PAMELA defines a limited number of "process idioms" and provides rules for their use. These rules guide the analyst in a very smooth transition between requirements analysis and preliminary design. It is this author's personal style to indicate the applied rules by their

number on the process graph. For instance, the symbols [1,6 | S] at the bottom of the TLM_stream_multibuf box in fig. 2a, indicate that this Single thread process (S), results from a user's requirement to provide an asynchronous interface (rule 1) of an application independent and hardware dependent nature (rule 6). The "?" and "!" show which process requested or originated the data flow, a control information vital to real-time applications (but specifically forbidden on DeMarco's DFDs).

During the preliminary design phase, the hierarchy of process graphs is mapped to Ada constructs such as abstract data types (type definition, procedures and functions), packages and tasks specification objects by a small set of simple rules. These rules encourage the re-use of library units. To simplify, multiple thread processes are mapped to packages. These packages encapsulate the single thread processes mapped to Ada tasks. "The leaves of the tree of this hierarchical structure are the procedures and functions invoked by the single thread processes." [Cherry-85]

In the detailed design phase, Ada PDL is entered in the preliminary design object bodies. This PDL is then refined into Ada code.

We found that PAMELA builds on proven modern software engineering techniques (DeMarco, Parnas, Hoare, Myers) to provide a very smooth transition between all software development phases; a quality deemed fundamental in the methodman document [Methodman-82]. Furthermore, "PAMELA uses all of Ada's advanced features (generics, packages, tasks, exceptions, and both forms of separate compilation) wisely and effectively. PAM adds a welcome limitation, form, and rationale to the use of Ada's many features which, without a suitable design and programming discipline, can and likely will be used in bizarre, ineffective, and inefficient ways." [Cherry-84]

THE VIEWGRAPH MATERIALS

of the

D. ROY PRESENTATION FOLLOW

Century Computing

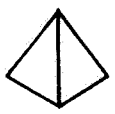
EVALUATION OF ADA

FOR MSOCC

DANIEL ROY

CENTURY COMPUTING, INCORPORATED

953-3330



Century Computing

ORIGINS OF CODE 511's ADA STUDY

- CENTURY'S "SOFTWARE TOOLS AND METHODOLOGY" STUDY FOR MSOCC IN 1984

- ADA IDENTIFIED AS A PROMISING TECHNOLOGY

- NEED TO EVALUATE ADA FOR MSOCC
  - ASSESS THE ADA LANGUAGE
  - DEMONSTRATE ITS USE ON A SMALL PILOT PROJECT

REQUIREMENTS ANALYSIS

PILOT PROJECT: REQUIREMENTS ANALYSIS

- WE PERFORMED A STANDARD STRUCTURED ANALYSIS FIRST
  - USING TEXT EDITOR TEMPLATES
  - AND EXISTING VMS TOOLS (SEARCH, RUNOFF, EDT)
- USING THE TOOLS, WE PRODUCED
  - A FULL DATA DICTIONARY
  - ALL MINI SPECIFICATIONS
- WE DOCUMENTED THE PROBLEMS OF SA WITH REALTIME APPLICATIONS

OPCON

OPCON is the benchmark software's operator interface (>OPCON-val-op-int). It also controls the initial activation and the shutdown of the system's other tasks.

SPECIFICATION

```
Level-1-single-tasks is (       EVEPRT, -- Events printer
                                TIMLOD) -- CPU time loader

Begin

    1. Prompt operator for Run-params

    2. Activate OCC simulator           -- >OPCON-ver-OCC-act

    3. for task in Level-1-single-tasks

        1. Activate task                -- >OPCON-ver-st-act


    4. end loop

    5. for i = 1 to IDLE-number-tasks

        1. Activate IDLE-i              -- >OPCON-ver-idle-act


    6. end loop

    7. delay req-run-time               -- >OPCON-ver-run-time

    8. Shutdown all activated tasks

    9. delay 1 second           -- See note 2 >OPCON-ver-shut-time

   10. Print stat-report (PRTRPT) -- >OPCON-val-stat-rep

end
```

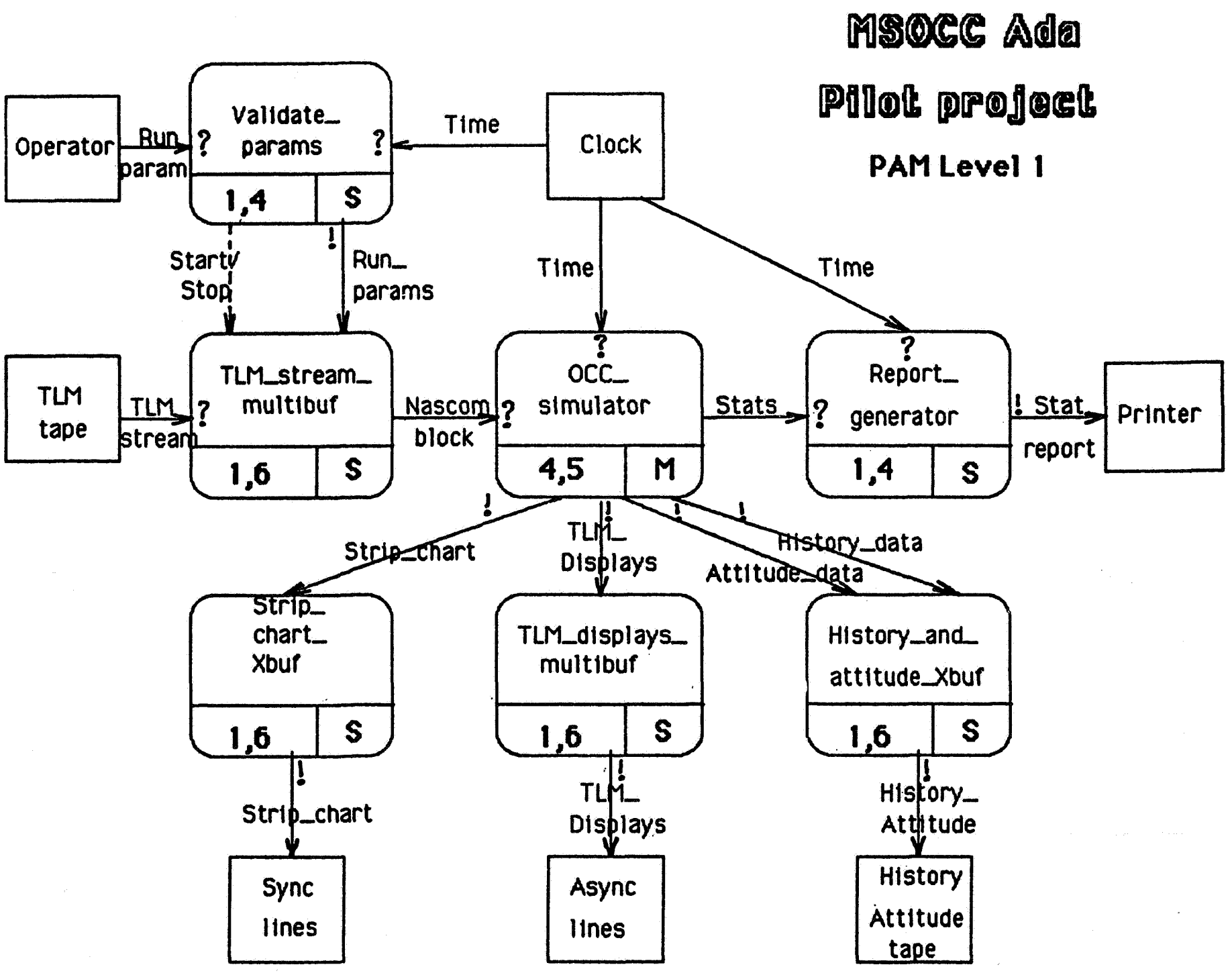
Fig 4-3: Minispec example built with the tools

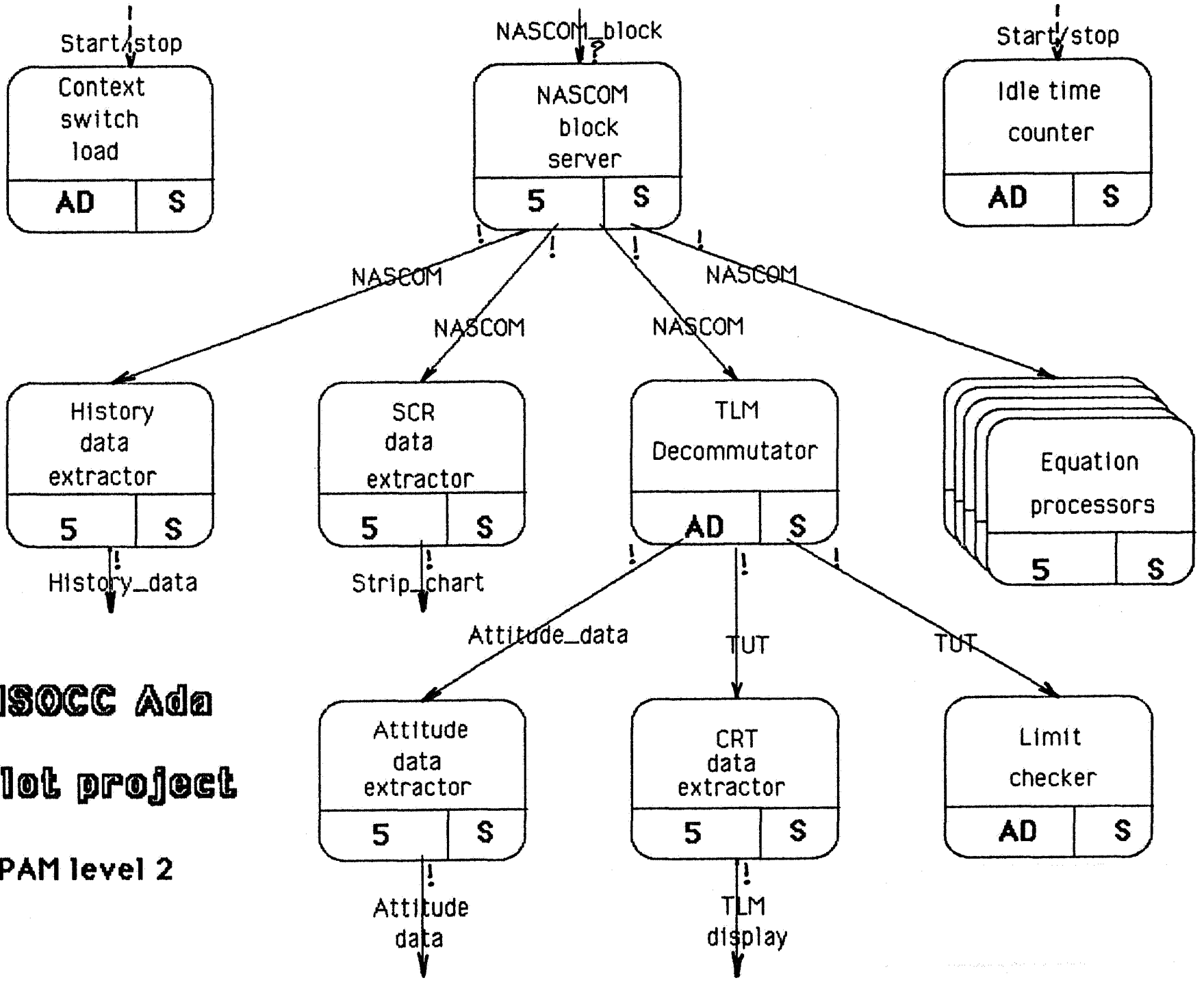
Century Computing

THE PROCESS ABSTRACTION METHODOLOGY

FOR EMBEDDED LARGE APPLICATIONS

(PAMELA OR PAM)


MSOCC Ada
Pilot project
PAM Level 1
Operator
Run param
Validate_ params
1,4
S
Time
Clock
Start/ Stop
Run_ params
Time
Time
TLM tape
TLM stream
TLM_stream_ multibuf
1,6
S
Nascom block
OCC_ simulator
4,5
M
Stats
Report_ generator
1,4
S
Stat report
Printer
Strip_chart
TLM_ Displays
History_data
Attitude_data
Strip_ chart_ Xbuf
1,6
S
TLM_displays_ multibuf
1,6
S
History_and_ attitude_Xbuf
1,6
S
Strip_chart
TLM_ Displays
History_ Attitude
Sync lines
Async lines
History Attitude tape



MSOCC Ada

Pilot project

PAM level 2

Century Computing

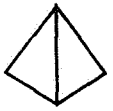
PRELIMINARY DESIGN

BARON preliminary design help

```
GOLB B => BARON TBD package              GOLD C => --| (doc), --* (PDL)
GOLD D => Bring in DOC template          GOLD E => Task entry
GOLD F => Function                       GOLD H => This text
GOLD P => Package                        GOLD S => Procedure
GOLD T => Task                           GOLD W => Bring WITH$EBP file in
GOLD X => Exception

GOLD > => half tab adjust right (*)      GOLD < => half tab adjust left (*)
GOLD TAB => half tab                     GOLD DEL => delete half tab (**)

(*) Must select range first like you would for tab adjust (control T)
(**) Careful, really does "delete" 4 times.

                BE SHORT IN PRELIMINARY DESIGN DOCUMENTATION

Algorithm:
        Can be ref to textbook and other biblio.
Effects:        --| mini-spec:
        Describes module functional requirements (more detailed than overview).
Errors:
        Describes error messages issued by module.
Modifies:       --| Side effects:
        Lists non-local variables modified (x.all. Access values, Global var).
Notes:
        User oriented description of dependencies, limitations, version
        number, status (prel des, code, etc.).  Limit change log to
        package level.
Overview:       --| Purpose:
        Describes module usage in very general terms.
Raises:
        Lists the exceptions that can be raised and not handled by module.
Requires:       --| Assumptions:
        Warns designer and user about limitations of implementation.
Synchronization:
        Describes synchronization requirements, tasks termination conditions,
        rendezvous time-outs, deadlocks prevention and other tasking reqs.
Tuning:         --| Performances:
        Specify timing and performance requirements.  Addresses performance
        issues that user can control.
```

Fig. 4-10: Preliminary design tool help

```
Package TBD is          --| Decision deferral package --*
--| Raises:
--|     None
--| Overview:   --| Purpose:
--|     This is an improvement over Intermetrics' TBD package and IEEE 990
--|     recommendations about decision deferral techniques.
--| Effects:    --| Description:
--|     The distinction is clarified between types, variables and values.
--|     The naming is more consistent (enum_i, component_i ...) and more
--|             readable (scalar_variable intead of scalarValue)
--|     There are more definitions (enum_type, record_type)
--|     Better compatibility with BYRON (or search utility processing)
--| Requires:   --| Assumptions:
--|     Please only "WITH" this package.  By systematically specifying
--|     "TBD.x" items, it is easier to assess the stage of development of
--|     a compilation unit.
--| Notes:
--|    Change log:
--| Daniel Roy   9-AUG-1985     Baseline
--|                                                                    --|
--      Constants
    some_constant : constant := 1;                                     --|
    positive_constant : constant := 10;                                --|
    negative_constant : constant := -10;                               --|
    real_constant : constant := 1.0;                                   --|

--      Defer decision about type (real),(discrete(enum,integer)), subtype
--      (natural,defined subtypes), range etc... that belong to detail design
    subtype some_type is integer range integer'first .. integer'last;  --|
    subtype scalar_type is integer range integer'first .. integer'last; --|

--      Distinguishes between type, variable and value (enum_1).
--      By convention (consistent with math notation) n is last.
--      Should be Enumeration_... all over for consistency.
--      But this is so much more comfortable.
    type enum_type is (enum_1, enum_2, enum_i, enum_p, enum_n);        --|
    enum_variable : enum_type := enum_1;                               --|
                                                                       --|

--      Keep consistency with enum_type
    type record_type is record                                         --|
        component_1 : integer := 0;                                    --|
        component_2 : integer := 0;                                    --|
        component_i : integer := 0;                                    --|
        component_p : integer := 0;                                    --|
        component_n : integer := 0;                                    --|
    end record;                                                        --|
    record_variable : record_type;                                     --|
                                                                       --|

--      Inspired by IBM PDL stuff
    Condition,CD : Boolean := true;                                    --|
                                                                       --|

--      Queues services
    type queue_type is array (array_index_type) of integer;            --|
    type queue_ptr_type is access queue_type;                          --|
--
end TBD;        --| --*
```



Century Computing

DETAILED DESIGN

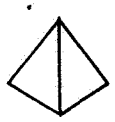
```
--
    procedure P (       --| synopsis --*
        param_1 : IN OUT some_type := some_constant ;   --| description --*
        param_n : IN OUT some_type              --| description --*
        ) ;                                 --| --*
```

Fig. 4-7: Preliminary design template for procedure (proc spec)

```
    separate ( )        --| --*
    procedure body P (  --| synopsis.  Must be the same as in body. --*
        param_1 : IN OUT some_type := some_constant ;   --| description --*
        param_n : IN OUT some_type              --| description --*
        ) is                                --| --*
--|
-- ****** Cut and paste from specification.  Use Gold D for rest of DOC. ******
--
-- Packages

-- types

-- subtypes

-- constants

-- records

-- variables

-- functions

-- procedures

-- separate clauses

    begin       --| --*
        null;
    end P ;     --| --*
```

Fig. 4-8: Detailed design template for a procedure (proc body)

```
separate (mbuf) --| --*
task body P is  --| processing task --*
--|
    procedure process_block (   --| Do something useful --*
        inp_ptr : IN data_ptr_type;     --| for input blocks --*
        outp_ptr : IN data_ptr_type     --| for output block --*
        ) ;                             --| --*
--
    procedure put_blocks (      --| Dump block queue --*
        Queue : IN out_Q_type   --| Where all output blocks are queued --*
        ) ;                             --| --*
--
begin   --| P --*
--
<<exception_block>>     --*
    begin       --* for recoverable exceptions
    --
    << till_EOF >>      --| loop until all input tasks are terminated --*
        while TBD.CD loop       --* Verification:
        << build_out_Q >>       --| loop until EOF or output queue full --*
            while TBD.condition loop    --* Verification:
                --* get in_ptr (RV with I tasks)
                process_block (in_ptr, out_ptr);        --*
                --* build queue
            end loop;   --* build_out_Q
            --
            put_blocks (out_queue);     --* watch EOF case
        end loop;       --* till_EOF
        --
            exception   --| --*
                when others     =>      --| --*
        --
        --| end exception;      --*
        --
    end ;       --* <<exception_block>>
--
    exception   --| --*
        when others     =>      --| --*
--
--| end exception;      --*
--
end P ; --| --*
```



Century Computing

CODE AND TEST

BARON code help

Gold A Access type	Gold M Modulo statement
Gold B Block statement (range,rename)	Gold N NEW (instantiations/access/tasks)
Gold C Case statement	Gold P Package use examples
Gold D Bring in doc template	Gold R Record (variable clause)
Gold E Entry statement	Gold S Procedure (declaration and code)
Gold F Function (declaration and code)	Gold T Tasks (select,terminate)
Gold G Generics (overloading)	Gold U Predefined attributes
Gold H This HELP menu	Gold W ?
Gold I IF-THEN-ELSE statement	Gold X Exception (raise)
Gold L Loop statements	
GOLD > => half tab adjust right (*)	GOLD < => half tab adjust left (*)
GOLD TAB => half tab	GOLD DEL => delete half tab (**)

(*) Must select range first like you would for tab adjust (control T)
(**) Careful, really does "delete" 4 times.

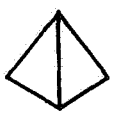
Fig. 4-15: Code and unit test tools built-in help

```
--
<<label>>       --*
    select      --*
        --* task.entry (params);
    or | else   --*
        --* delay (time_out) | any_other_statement
    end select; --* <<label>>
```

Fig. 4-20a: Entry call template copied in program

```
Selective entry call (no more that 2 alternatives !)
<<TLM_in>>      --*  calls TLM_stream_multibuf.do_you_have_a_block ?
    select      --*
        TLM_stream_multibuf.do_you_have_a_block (nascom_block_Xbuff);
    else        --*
        --* increment TLM_stream_multibuf overrun
        TLM_stream_multibuf_stat.increment (overrun);
    end select; --* <<TLM_in>>

Selective WAIT (any number of alternatives)
<<scr_loop>>    --* Accept and send block
    loop        --*
        select  --*
            accept here_is_a_block (    --| Accept NASCOM block --*
                nascom_block_Xbuff : IN nascom_block_Xbuff_type --| --*
                ) do    --| --*
                local_block := nascom_block_Xbuff ;
            end here_is_a_block ;       --| --*
            --* calls strip_chart_multibuf.here_is_a_set !
            put_line ("SCR_data_extractor saw a block");
        or      --*
            terminate;  -- could be delay for time-out
        end select;     --*
    end loop;   --* scr_loop
```

Fig. 4-20b: The examples buffer for task entries



Century Computing

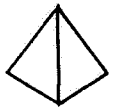
METRICS

Century Computing

METRICS

- SOFTWARE ENGINEERING LABORATORY WORK WAS AT THE BASIS OF OUR METRICS EFFORT

- DEVELOPED SIMPLE TOOLS IN DCL (LOGGER, LOC COUNTER)
- DEVELOPED A REFINED WBS
- KEPT ON LINE SEVERAL FILES DOCUMENTING OUR EFFORT
  - WEEKLY REPORTS
  - PROBLEMS.ADA (INCLUDING COMMENTS .LIS, ETC...)
  - GREAT.ADA (FOR NICE FEATURES OF THE LANGUAGE AND COMPILER)



Century Computing

RESULTS

	Hours	%
Training	253	22.9
Requirements	105	9.5
Design	93	8.4
Code/test	335	30.3
Tools dev	319	28.9

Fig. 4-17: Development data

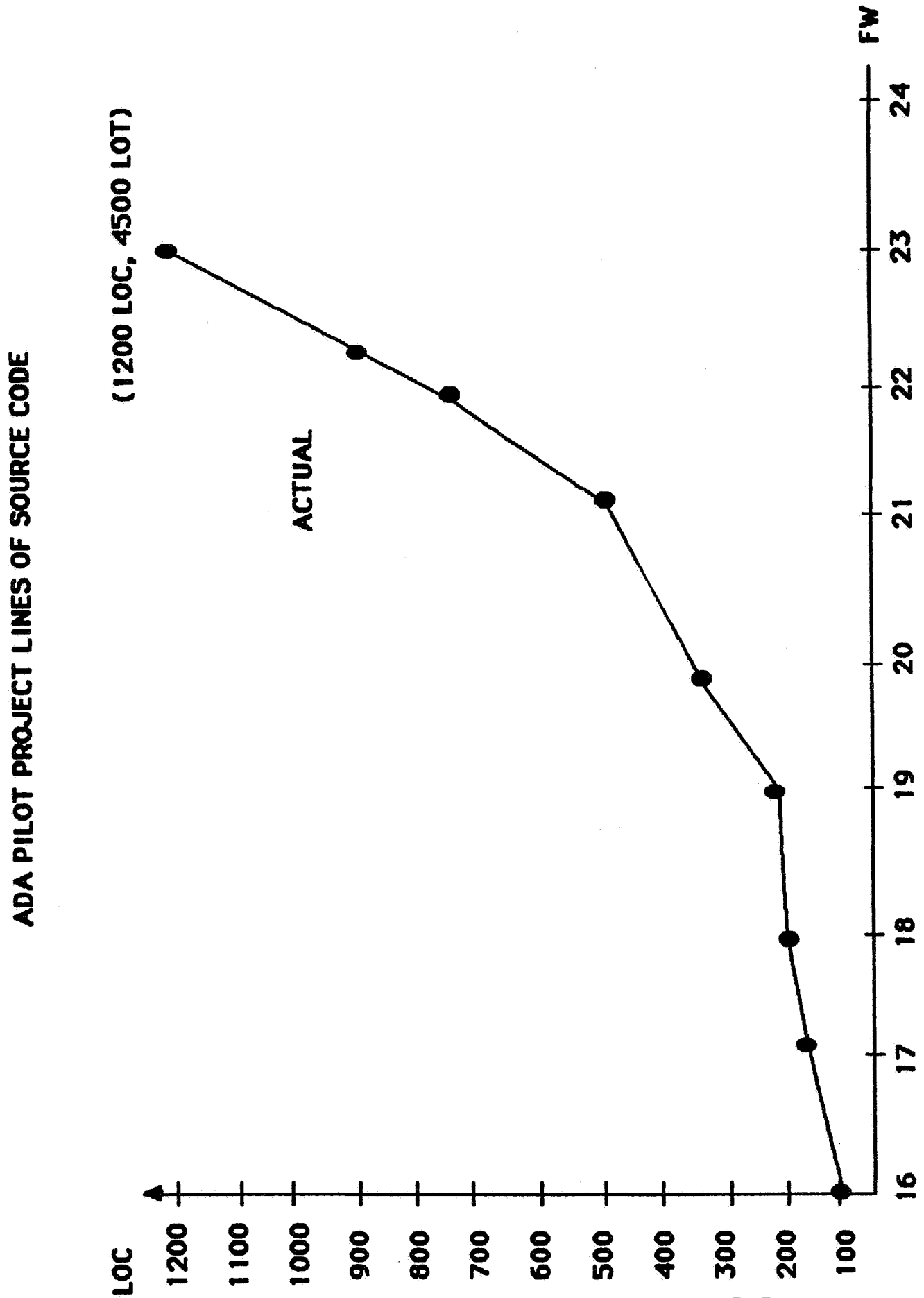

ADA PILOT PROJECT LINES OF SOURCE CODE
(1200 LOC, 4500 LOT)
ACTUAL
LOC
1200
1100
1000
900
800
700
600
500
400
300
200
100
16
17
18
19
20
21
22
23
24
FW

Figure 2-1

Century Computing

CONCLUSIONS

- ADA VERSATILITY (PSL, DDL, PDL, IL)
- ADA COMPLEXITY (NEED FOR A METHODOLOGY AND TOOLS)
- START TRAINING NOW (SE FIRST, ADA SECOND)
- PILOT PROJECT IS THE WAY TO GO (NO SCHEDULE PRESSURE)
- TASKING WORKED WELL FOR US (TASK TYPES, I/O CONCURRENCY)
- PAMELA WORKED VERY WELL FOR US (PRODUCED EFFICIENT DESIGN)
- DEC ACS IS A SUPERB IMPLEMENTATION

OBSERVATIONS FROM A PROTOTYPE IMPLEMENTATION OF THE COMMON APSE INTERFACE SET (CAIS)

Mike McClimens, Rebecca Bowerman, Chuck Howell,
Helen Gill, and Robbie Hutchison
MITRE Corporation

EXECUTIVE SUMMARY

This paper presents an overview of the Common Ada Programming Support Environment (APSE) Interface Set (CAIS), its purpose, and its history. The paper describes an internal research and development effort at the Mitre Corporation to implement a prototype version of the current CAIS specification and to rehost existing Ada software development tools onto the CAIS prototype. Based on this effort, observations are made on the maturity and functionality of the CAIS. These observations support the Government's current policy of publicizing the CAIS specification as a baseline for public review in support of its evolution into standard which can be mandated for use as Ada is today.

CAIS HISTORY

The Ada programming language was developed by the United States Government to promote the maintainability, portability, and reusability of software. Although no special software tools are required to use the Ada language, a collection of portable and modern tools is expected to enhance the benefits of using Ada. The term Ada Programming Support Environment (APSE) is used to refer to the support (e.g., software tools, interfaces) available for the development and maintenance of Ada application software throughout its life cycle. The Common APSE Interface Set (CAIS) is the interface between Ada tools and host system services, which is being standardized to promote portability of tools among APSEs.

In 1980, the DoD sponsored two efforts to develop APSEs: the Ada Language System (ALS) contracted to Softech by the Army and the Ada Integrated Environment (AIE) contracted to Intermetrics by the Air Force. The DoD also funded publication of the document, Requirements for Ada Programming Support Environments, nicknamed "Stoneman". It is the Stoneman document that first defined layers within an Ada Programming Support Environment. The Ada Joint Program Office (AJPO) was formed in late 1980 to serve as the principle DoD agent for the coordination of all DoD Ada efforts.

Multiple DoD-sponsored APSEs threatened to undermine the Ada program's goal of commonality. In late 1981/early 1982 AJPO established the

Kernel APSE Interface Team (KIT) as a tri-service organization chaired by the Navy. The KIT was supported by an associated group consisting of members from industry and academia, called the KIT Industry and Academia (KITIA). The charter of the KIT and KITIA was to define the capabilities that comprise the Kernal APSE layer (KAPSE) and its interface to dependent APSE tools. The interface between the KAPSE and dependent APSE tools became called the Common APSE Interface Set and a subgroup of the KIT/KITIA called the CAIS Working Group was formed to define a standard for this set of interfaces.

The CAIS has been an evolving concept. It began as a bridge between the Army and Air Force APSEs but has become a more generalized operating system interface. However, issues such as interoperability, configuration management, and distributed environments have not yet been addressed. Significant changes have appeared with each iteration of the CAIS specification up to the submittal in January 1985 of CAIS Version 1 as a proposed Military Standard (MIL-STD-CAIS).

In response to concern from the Ada community that the CAIS, as defined in Version 1, is too premature for standardization, a policy statement was released along with the proposed MIL-STD-CAIS directing that use of the CAIS be confined to prototyping efforts. The policy clearly states that the CAIS should not at this time be imposed on development or maintenance projects where the primary purpose is other than experimentation with the CAIS.

Further refinement of the CAIS is planned, but a contract to produce Version 2 of the CAIS specification has not yet been competed. Potential future applications of the CAIS include several major government projects (e.g., STARS and the NASA Space Station).

CAIS OVERVIEW

The CAIS is a set of Ada package specifications that serve as calls to system services. The implementation of these packages may differ between systems while the package specifications remain the same. These package specifications then become a system independent interface between software development tools and the host operating systems. The CAIS is composed of four major sections: a generalized node model, support for process management, an extended input/output interface, and an abstraction for the processing of lists.

The generalized node model is by far the most significant part of the CAIS. Processes, structures, and files may all be represented as nodes. Among other features, the node model provides a replacement for the host file system. As such it contains enough functionality to support the needs of tools rehosted from a wide range of file systems. The node model is a hierarchical tree augmented by secondary relationships between nodes. Attributes may be assigned to any node or relationship in the tree. The attribute and relationship facilities provide a powerful mechanism for organizing and manipulating interrelated sets of nodes. The node model also provides support for mandatory (secret, etc.) and discretionary access control (read only, etc.).

Process support and an extended set of I/O interfaces are integrated with the node model. Process support is not extensive but does include the facilities to spawn and invoke processes or jobs and facilities for communication of parameters and results between processes. The I/O interfaces, on the other hand, are quite voluminous. Although they constitute more of the specification than the node model, the I/O interfaces largely duplicate the I/O support provided in Ada. In addition to integrating I/O with the node model, CAIS I/O tightens some of the system dependencies left in Ada and defines standard interfaces for devices such as scroll terminals, page terminals, and tapes.

The CAIS defines an abstract data type for processing lists. CAIS Lists may be any heterogeneous grouping of integers, strings, identifiers, sublist, or floating point items. Items may be named or unnamed. Lists are used throughout CAIS for the representation of data such as attributes and parameter lists, and they provide a powerful abstraction for tool writers in general.

MITRE'S PROTOTYPE CAIS

Under a three staff year (Oct 84 to 85) internal research and development effort, MITRE Corporation has implemented a large subset of the CAIS specification and has exercised both rehosted and newly-written tools on this prototype. The MITRE prototype includes the node model, the list utilities, Text_Io, Direct_Io, and Sequential_Io. Parts of the process model and scroll_terminal have also been implemented in support of a line editor and a menu manager rehosted from other systems. In the next year the prototype will be completed, additional tools will be rehosted, the CAIS will be rehosted to a second system, and an analysis of distributing the CAIS will be undertaken. The prototype CAIS was developed using the Verdix Ada compiler running under Ultrix on a DEC VAX 11/750. Of the two tools rehosted to the prototype, one was originally developed using the Data General Ada compiler, and the other, using the Telesoft compiler.

The objective of MITRE's prototype development was to submit the CAIS specification to the rigor of implementation and actual use. It was believed that implementation of a prototype would test the implementability of the CAIS specification, would identify the level of support that CAIS provided to existing tools, and would result in practical input to CAIS designers, DoD policy makers, and program managers. The primary focus was on evaluating the CAIS functionality and not on developing an efficient implementation.

The consensus from this study is that the CAIS, for the most part, is internally consistent and provides a good foundation for continued work in standardized operating system interfaces for Ada programming support environments. The next version of the CAIS must, however, be considerably more complete in its specification. Table 1 lists the specific observations made as a result of the prototype

Section	Item	Scale	Scope
3.1.1	The conceptual model is consistent, except for the I/O packages.	N/A	N/A
3.1.2	Some of the semantics are ambiguous.	Major	Semantics
3.1.3	Redundant capabilities and alternate interfaces need tightening.	Medium	Both
3.1.4	The nesting of packages within the package CAIS is not explicitly required.	Minor	N/A
3.1.5	The use of limited private types implies a need for additional facilities.	Minor	N/A
3.1.6	The error handling model in the specification is insufficient.	Major	Both
3.1.7	Parameter modes and positions are sometimes inconsistent.	Minor	Interface
3.1.8	The use of functions versus procedures should be consistent.	Minor	Interface
3.2.1	Multiple definitions of subtype names exist.	Minor	Interface
3.2.2	Inconsistent descriptions of access synchronization constraints are given.	Minor	N/A
3.2.3	Unnecessary complexity is introduced with the predefined relation 'User.	Minor	Semantics
3.2.4	The description of implied behavior of open nodes is good but needs to be more explicit.	Medium	Semantics
3.2.5	Boundary conditions are undefined.	Medium	Semantics
3.2.6	Capabilities for node iterators are limited.	Medium	Both
3.2.7	Definition of node iterator contents is ambiguous.	Medium	Semantics
3.2.8	Pathnames are inaccessible from node iterators.	Minor	Both

Section	Item	Scale	Scope
3.3.2	Ability to specify initial values for path attributes is missing.	Minor	Both
3.3.3	Error in sample implementation of additional interface for Structural_Nodes.Create_Node.	Minor	N/A
3.4.1	Treatment of files departs from the node model.	Major	Both
3.4.2	Consequences are implied by a common file type.	Medium	Both
3.4.3	Initialization semantics are incomplete.	Medium	Semantics
3.4.4	Mode and Intent are coupled.	Minor	Both
3.4.5	Additional semantics are needed for multiple access methods that interact.	Medium	Semantics
3.4.7	Import_Export of files is under-specified.	Medium	Both
3.4.8	Semantics of attribute values are conflicting.	Minor	Semantics
3.4.9	Interfaces diverge from Ada IO.	Minor	Interface
3.5.1	Clarification of dependent processes is needed.	Minor	Semantics
3.5.2	Support for process groups is needed.	Medium	Both
3.5.3	Proliferation of process husks is implied by the interfaces.	Minor	Semantics
3.5.4	Disposition of handles following process termination needs to be clarified and restricted.	Medium	Semantics
3.5.5	Parameter passing and inter-tool communication need to be re-evaluated.	Major	Both

Section	Item	Scale	Scope
3.5.6	Response is undefined when attempting to spawn a process that requires locked file nodes.	Minor	Semantics
3.5.7	Clarification of IO_Units and IO_Count with respect to meaning of Get and Put operations is needed.	Minor	Semantics
3.6.1	The use of predefined attributes should be clarified.	Medium	Semantics
3.6.2	Attribute values should not be restricted to List_Type.	Medium	Both
3.6.5	The order of Key and Relationship parameters should be reversed.	Minor	Interface
3.7.1	Enclosing string items in quotes decreases readability and is unnecessary.	Minor	Semantics
3.7.2	List_Utilities should present a textual rather than a typed interface.	Medium	Both
3.7.3	Token_Type should include all list items, not just identifiers.	Minor	Both
3.7.5	The Position parameter should never be required for operations on named lists.	Minor	Interface
3.7.6	Nested packages names conflict with Item_Kind enumerals.	Minor	Interface
4.3	Handling of control characters remains poorly defined.	Medium	Semantics
4.4	The Scroll_Terminal package provides improvements over Ada IO packages.	N/A	N/A

implementation. Many of these comments reflect ambiguities in the text. Some major refinement of exception handling, input/output, and the list utilities is recommended. Other comments reflect specific technical areas and may be addressed by simple modification or addition to existing interfaces. While the required changes certainly appear to be within the scope of the planned upgrade, Version 2.0 of the CAIS will likely contain significant changes to the operational interfaces for tools. The most difficult problems to evaluate are the ambiguous areas of the specification which may simply disappear or which may result in considerable conflict depending upon the nature of the resolution that is adopted.

MAJOR OBSERVATIONS AND RECOMMENDATIONS

The results of MITRE's prototype implementation of the Common APSE Interface Set support the Government's current policy for promulgating the CAIS. The CAIS provides a relatively consistent set of interfaces which address portability issues, but it is not refined to the degree that it can be mandated as a standard. The non-binding Military Standard CAIS issued 31 January 1985 publicizes the direction that the CAIS is taking. It can be used as guidance for current development efforts and provides a baseline for public critique.

An upgrade of the current definition of CAIS is planned. The new document, CAIS Version 2.0 will be an input to the Software Technology for Adaptable Reliable Systems(STARS) Software Engineering Environment program. It is intended that CAIS Version 2.0 have the quality and acceptance required of a true military standard. To achieve this quality, the upgrade will have to add rigorous precision to the current document, will have to refine several existing technical areas, and will have to include technical areas previously postponed.

CAIS Version 2.0 should be expected to contain major refinements and additions to the current document. The MITRE prototype effort has found five major issues that must be addressed in the next revision of the current document:

* The current document is ambiguous and imprecise--more rigor and precision is required.

* The List_Utilities abstraction can be made simpler, more complete, and more consistent.

* A central model is required for CAIS exception facilities.

* The CAIS IO model is not uniform-- it is inconsistent with Ada and with the CAIS node model

* The CAIS does not adequately address interactions between itself and the host operating system.

RESOLUTION OF AMBIGUITIES

The precision with which the CAIS is specified in the current document leaves many issues open to the interpretation of the implementor. The semantics of many routines are not specified in detail; implications of alternate interfaces and suggested implementations are not addressed in text; broad statements are made in introductory sections and then are not reflected in discussions of specific routines; information on specific topics (such as predefined attributes) is dispersed throughout the document; and interactions among routines are not qualified. Together these deficiencies result in confusing the intentions of the CAIS and in giving an impression that the CAIS is not completely thought out. Unless corrected, they will make implementation of the CAIS difficult and standardization across CAIS implementations improbable. Clarification of the specification is also necessary to achieve the widespread acceptance necessary for adoption of CAIS as a standard.

LIST UTILITIES REFINEMENT

During the most recent revision of the CAIS document, the List_Utilities package underwent significant modification. Further refinement is necessary. The List_Utilities package provides an abstraction that is used throughout the CAIS. Our recommendation is that the definition of Token_Type be expanded so that it can represent any of the list items currently supported (lists, integers, floating points, strings, and identifiers). This will allow the removal of redundant subprograms, will provide a more consistent interface, and will provide more functionality with less complexity. Enhancements to List_Utilities may allow the CAIS features that rely on List_Utilities to also be enhanced.

CENTRAL EXCEPTION MODEL

The treatment of exceptions in the current document is inadequate. The Ada specifications do not correspond to the text, and the text references exceptions by unqualified names. The same exception name is used to refer to several different error conditions. Thus it is difficult to determine the complete set of CAIS exceptions and their relationships. It appears that exceptions were considered only on a procedure-by-procedure basis. A CAIS user will expect a single exception model that is consistent across the entire CAIS. We have proposed a candidate set of exceptions that addresses the entire CAIS and that reduces the instances of exceptions with multiple meanings. The method of exception handling in the Ada I/O packages could be adopted as a model for coordinating exceptions across several packages, or all exceptions could be declared in the package CAIS. However, the CAIS must evolve to one, consistent, well-engineered model for exception handling.

CLARIFICATION OF THE I/O MODEL

The co-existence of both node handles and file handles makes the CAIS file nodes inconsistent with either process or structural nodes. The entire treatment of I/O facilities in CAIS suffers from its unclear relationship with Ada I/O facilities. Large sections of the CAIS I/O packages currently refer to Ada I/O packages without addressing specific effects of differences. While Ada defines distinct file types for Text_Io, Direct_Io, and Sequential_Io, the CAIS defines a single file type and indicates that operations from different I/O modes may be intermixed. However, many implications arising from this capability are not adequately addressed. The description of CAIS I/O would be greatly improved by discussing its intended compatibilities and differences with Ada I/O.

CAIS AND THE HOST OPERATING SYSTEM

For an indefinite time, CAIS environments will be required to co-exist with the environment of the host operating system. It is unreasonable that all host facilities be converted to interface with a newly installed CAIS. Military Standard CAIS simply does not address issues related to this co-existence. Even the procedures for importing and exporting files between the two systems disregard important properties of host files and of CAIS files. Methods need to be established for reporting host errors, activating host processes, and making the contents of file nodes available to non-CAIS programs. Unless standards are established to integrate the host and CAIS environments, users of each CAIS will develop their own methods, and portability across CAIS implementations will be impacted.

THE VIEWGRAPH MATERIALS

for the

M. McCLIMENS PRESENTATION FOLLOW

MITRE CAIS PROTOTYPE

MITRE Corporation,
W33 Ada Software Eng.
IR&D

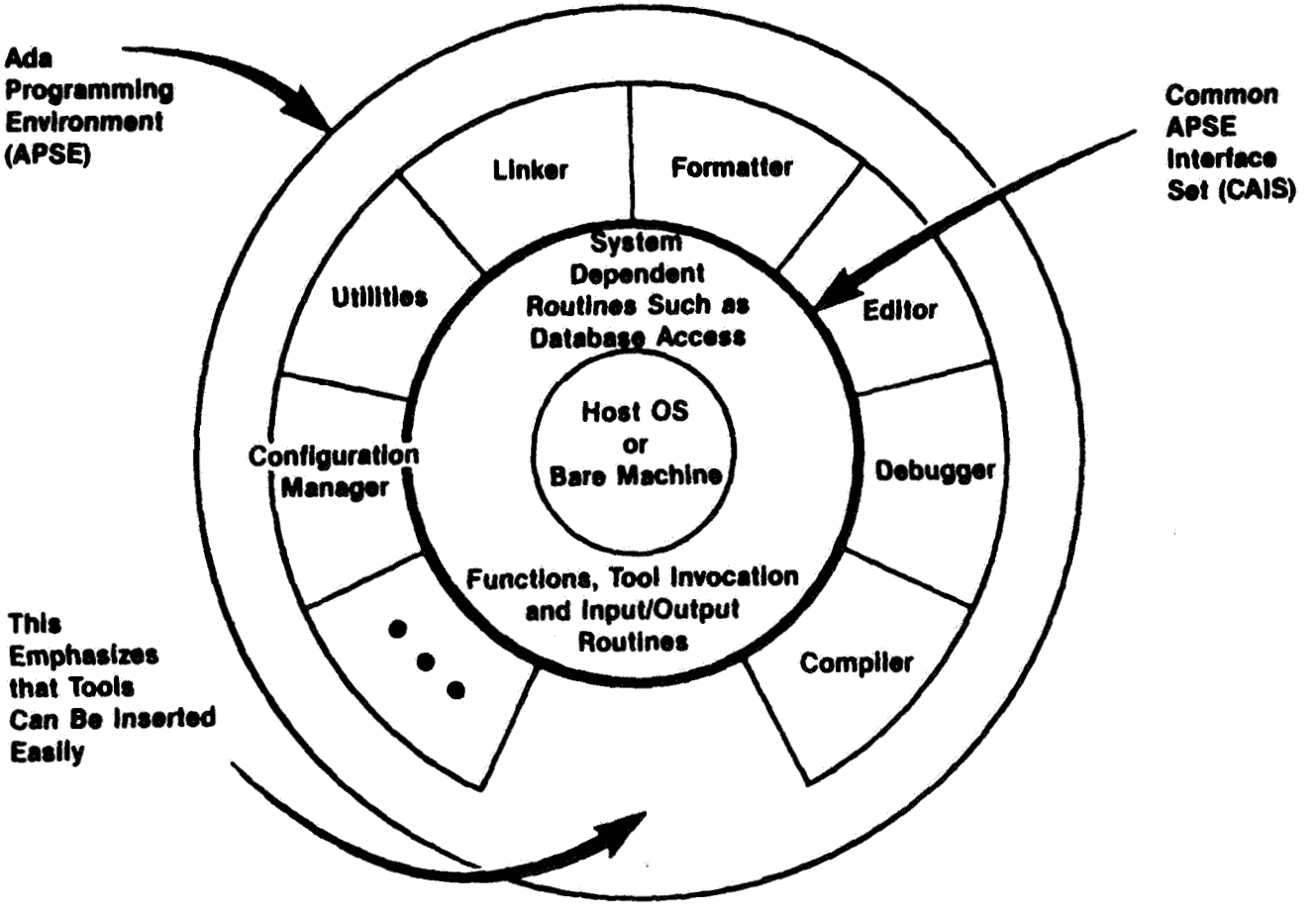
M. McClimens
MITRE Corp.

Background

- Air Force Was Lead Service For Ada PSE (APSE) Development: The Ada Integrated Environment (AIE)
- Army Was To Develop A Few Critical Tools And Take Lead In Education
  - Compiler And Tools Intended To Act As A Bridge Between Existing Environments And The AIE: The Ada Language System (ALS)
- Ultimate Goal Was The Development Of A DoD Standard APSE To Complement Ada

M. McClimens
MITRE Corp.

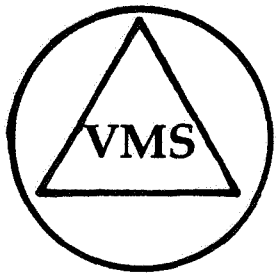
APSE STANDARDIZATION

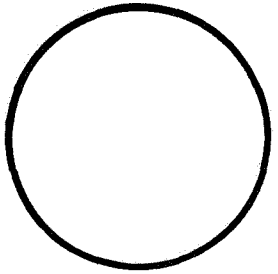


M. McClimens
MITRE Corp.

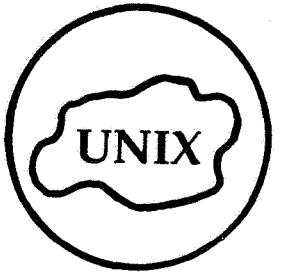
CAIS Goals

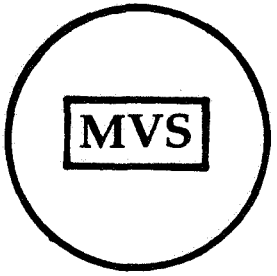
- Portability
- Uniformity
- Interoperability

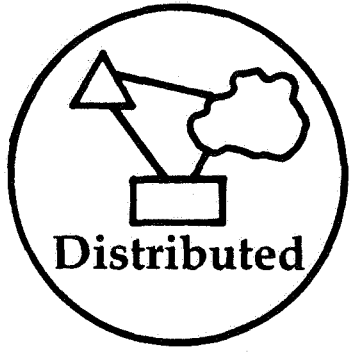


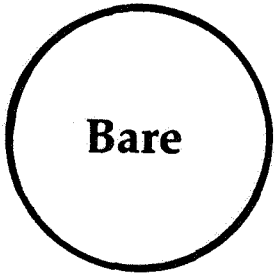


Standard CAIS









Background

- Commercial Compilers Available On A Variety Of Systems
- WIS SDME
- NASA Space Station SEE
- STARS Software Engineering Environment (SEE)
- Several European APSE Efforts

M. McClimens
MITRE Corp.

Background

- CAIS Working Group (CAISWG) Formed To Define Interfaces To Act As A "Bridge" Between ALS And AIE KAPSEs

- Goal Was To Support Tools Written In Ada That Use CAIS Services, Promoting Portability Between The ALS and AIE

M. McClimens
MITRE Corp.

Overview of The CAIS

- CAIS Is Defined As A Set Of Ada Package Specifications And A Description Of Associated Semantics

- Underlying Model Is A Directed Graph With Attributes
  - Nodes Can Be Files, Processes, Or "Directories"
  - Graph Nodes And Edges Have Attributes

M. McClimens
MITRE Corp.

What is in the CAIS

- Node Management
  - Expanded File System
- Process Management
  - Spawn/Invoke Abort/Suspend/Resume
- I/O
  - Text, Direct, Seq., Devices (Scroll, Page, ...)
  - Import/Export
- List_Utilities
  - Abstract Data Type
  - Heterogeneous list of items

M. McClimens
MITRE Corp.

What is not in the CAIS

- Support for Concurrency
- Memory Management
- Interrupts

→ Ada

- Scheduling
- Paging/Segmentation
- Low Level I/O

→ Operating System

- DBMS

M. McClimens
MITRE Corp.

Background

- CAIS Has Evolved To A General System-Independent Operating System Interface

- Current Status of CAIS
  - Controversy Over Perceived Premature Standardization
  - Version 1 Is A Draft MIL-STD
  - Explicitly for Prototyping Only
  - Version 2 Contract Is Planned
    - Funded Design Of CAIS

M. McClimens
MITRE Corp.

Objectives of MITRE FY85 Work

- Select Subset Of CAIS And Implement Prototype
- Port Ada Tools To The CAIS Prototype
- Evaluate Implementability Of Specification
- Evaluate How Well The Interfaces Support Tools
- Provide Practical Input To CAIS Designers

M. McClimens
MITRE Corp.

Technical Approach

- Ada/Ed Philosophy; Efficiency Is NOT A Priority
- Top Priority Is To Implement Subset Of CAIS
- Secondary Storage Is Used To Store Node Information
- Host (UNIX) Dependencies Are Isolated
- Using Verdix Ada Development System On ULTRIX

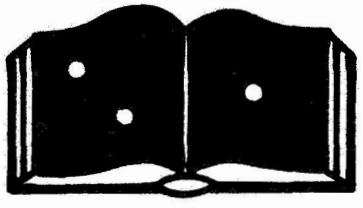
General Comments

- Learning Curve For Using CAIS Will Be SIGNIFICANT
- Overall "Conceptual Consistency" Among Parts Is Good
- Implementation Is Uncovering Problems That Would Be Very Hard To Find By Inspection Of Specification
- CAIS Is A Good Vehicle For Continued Work In Standardized OS Interfaces And PSE/SEEs

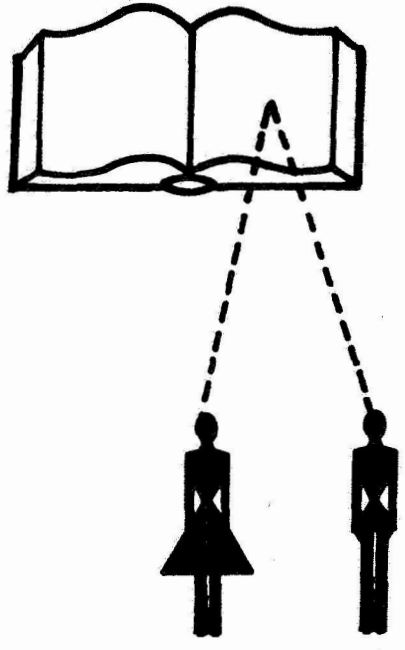
Resolution of Ambiguities

☛ **MIL-STD-CAIS needs to be Written with More Rigor**

Completeness



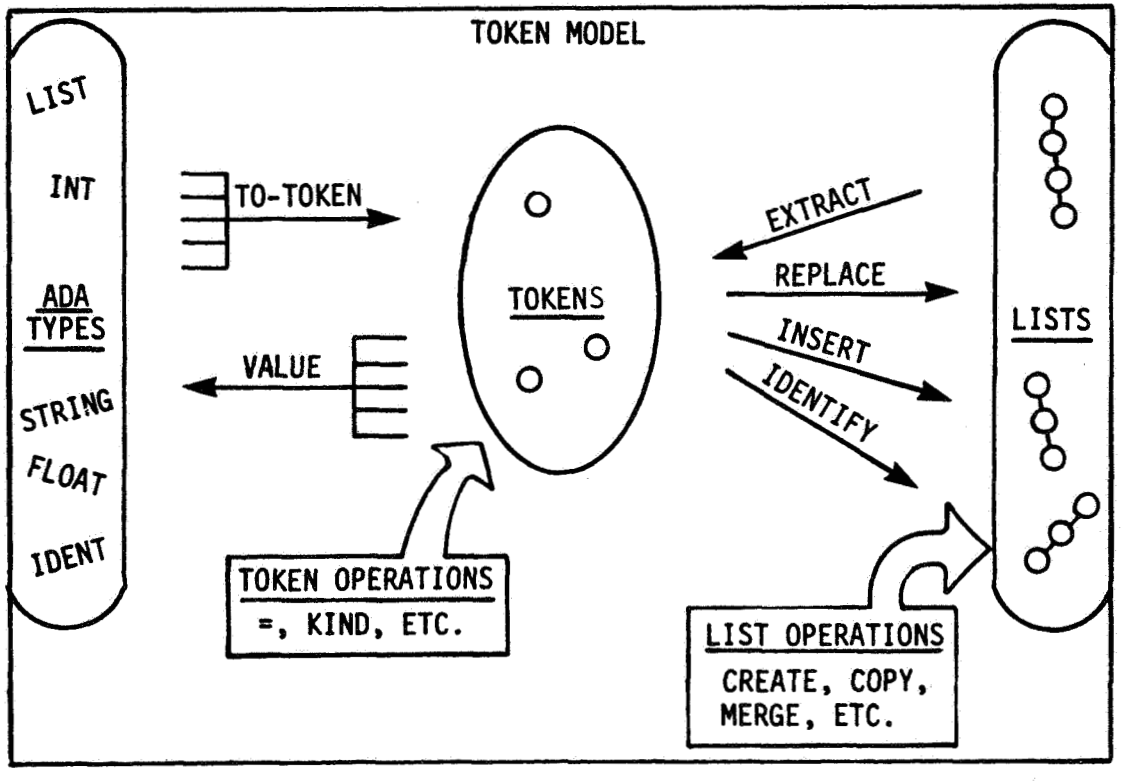
Precision



M. McClimens
MITRE Corp.
24 of 29

List_Utilities Refinement

- The Current Model is Cumbersome and Inconsistent
- List_Types should Represent Lists of Tokens



M. McClimens
MITRE Corp.

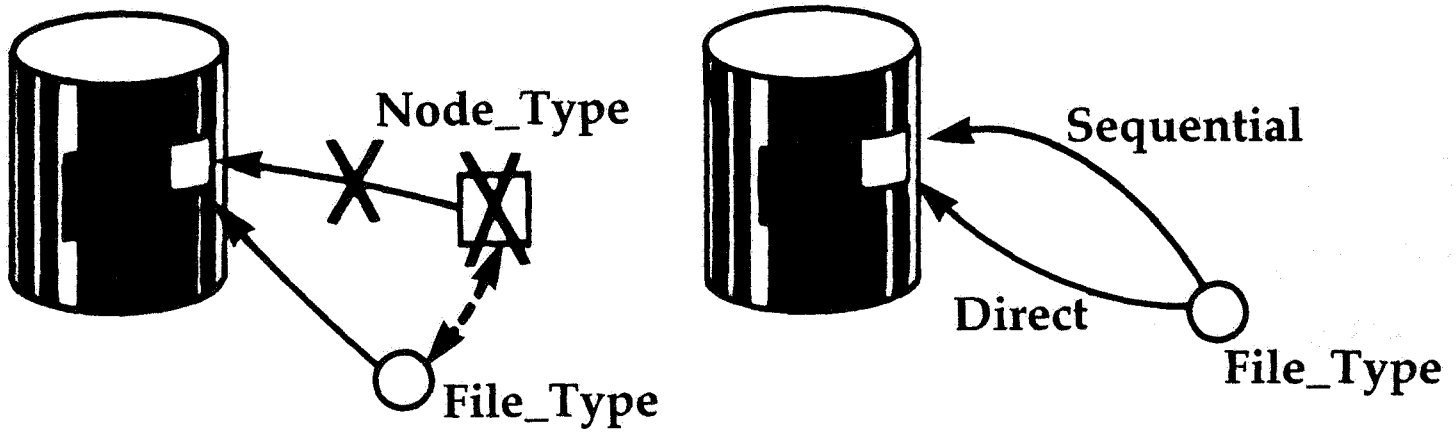
Central Exception Model

- Currently, Addressed Procedure-By-Procedure
  - Affects Clarity of MIL-STD-CAIS
  - Affects User's Interface to CAIS

- The Set of CAIS Exceptions should be Centralized
  - Declared Together
  - Renamed in Each Package where they are used
  - Names Refined to Establish a Clear Meaning

Clarification of I/O Model

- File_Type Differs from Cais's Node_Type
  - Affects Access Control
- File_Type Differs from Ada's multiple File_Type
  - Affects Mixed Operations



CAIS and the Host Operating System

- All Host Facilities won't be compatible with CAIS
- Access to Nodes and to Host Files must be Addressed
  - Expose / Escape Sequence for Host File Names

M. McClimens
MITRE Corp.

Objectives for Rest Of FY86

- Extend Prototype
- Continue To Port Ada Tools to CAIS
- Port Prototype To VMS, Sun Workstations
- Port CAIS/UNIX Tools To CAIS/VMS
- Add Support For Distributed Processing
- Make The Source Widely Available

MEASURING ADA* AS A SOFTWARE DEVELOPMENT TECHNOLOGY IN THE SOFTWARE ENGINEERING LABORATORY (SEL)**

William W. Agresti***
Computer Sciences Corporation
and the SEL Staff

ABSTRACT

An experiment is in progress to measure the effectiveness of Ada in the National Aeronautics and Space Administration/ Goddard Space Flight Center flight dynamics software development environment. The experiment features the parallel development of software in FORTRAN and Ada. The experiment organization, objectives, and status are discussed. Experiences with an Ada training program and data from the development of a 5700-line Ada training exercise are reported.

INTRODUCTION

An experiment is underway to assess the effectiveness of Ada for flight dynamics software development. This paper is an interim report on the experiment, discussing the objectives, organization, preliminary results, and plans for completion.

---

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

**Proceedings, Tenth Annual Software Engineering Workshop, National Aeronautics and Space Administration, Goddard Space Flight Center, December 1985.

***Author's Address: Computer Sciences Corporation, System Sciences Division, 8728 Colesville Road, Silver Spring, Maryland 20910.

The Ada experiment is planned and administered by the Software Engineering Laboratory (SEL) of the National Aeronautics and Space Administration's Goddard Space Flight Center (NASA/GSFC). NASA/GSFC and Computer Sciences Corporation (CSC) are cosponsors of the experiment. Personnel from all three SEL participating organizations (NASA/GSFC, CSC, and the University of Maryland) support the experiment.

TECHNOLOGY ASSESSMENT IN THE SEL

There is a great deal of optimism concerning Ada's potential effect on software development. The SEL seeks to establish an empirical basis for understanding Ada's effectiveness in a particular environment--namely flight dynamics software development at NASA/GSFC. Figure 2* shows some of the characteristics of this development environment. (Reference 1 contains a more detailed description.)

As Figure 2 implies, in seeking to understand the effectiveness of Ada, the SEL is approaching this task as it has addressed the assessment of other software technologies. Some methods that have been demonstrated to be effective in other environments have not been effective in the SEL environment. The SEL is therefore cautious about expecting that reported experiences with Ada will obtain in the SEL environment. Instead, the SEL seeks to conduct an assessment of Ada in its own environment.

The assessment methods used by the SEL have included controlled experiments, case studies, and analytical investigations. The Ada assessment is referred to as an experiment, although it is clearly not a controlled experiment. Identifying this effort as an experiment follows the general use

*All figures are grouped together at the end of the paper.

of the word to denote "any action or process undertaken to discover something" (Reference 2). As the later discussion will make clear, the Ada experiment is a highly instrumental case study of an Ada implementation in parallel with a FORTRAN implementation, with both systems developed in response to the same requirements.

OBJECTIVES

The primary objective of the experiment (Figure 3) is to determine the cost-effectiveness of Ada and its effect on the flight dynamics environment. A related objective is to assess various methodologies that are related to the use of Ada. An initial set of such methodologies includes object-oriented design (Reference 3), the process abstraction method (Reference 4), and the composite specification model (Reference 5). Additional methodologies will be identified as the experiment continues.

Reusability is an important tactic for cost-effective software development, both in a general sense and in the SEL environment. Ada was designed (in part) to facilitate reusability. This experiment seeks to develop approaches for reusability when Ada is the implementation language.

The Space Station is a program of great size, complexity, and significance to NASA. Ada has been recommended as the language to be used for the development of new software for the Space Station. An objective of the Ada experiment is to develop measures that may assist in planning for the large-scale use of Ada in the Space Station program. Examples of such measures are those that relate to size, productivity, or reliability in an Ada implementation.

Because the experiment is not completed, these objectives have not yet been met. However, experiences thus far will contribute to addressing the objective of understanding the effect of Ada.

EXPERIMENT PLANNING

The experiment consists of the parallel development, in FORTRAN and Ada, of the attitude dynamics simulator for the Gamma Ray Observatory (GRO) (Figure 5); which is scheduled to be deployed in May 1988. It is worth noting that the dynamics simulator is part of the standard complement of ground support software planned for the GRO mission. The simulator would routinely be developed in FORTRAN alone; because of the experiment, it is being developed in Ada as well.

When completed, the system is expected to comprise 40,000 source lines of (FORTRAN) code, requiring 18 to 24 months to develop on a VAX-11/780 computer. Each team was staffed initially with seven personnel from NASA/GSFC and CSC. Each development project is expected to require 8 to 10 staff-years of effort.

Three teams have a role in the experiment (Figure 6): the Ada development team; the FORTRAN development team; and an experiment study team consisting of NASA/GSFC, CSC, and University of Maryland personnel. The study team is responsible for planning the experiment, collecting data from the development teams, and evaluating the progress and results of the experiment. The study team will also be able to compare the software products generated by each team.

The profiles of the development teams (Figure 7) reveal that the Ada team on average is familiar with more programming languages and is more experienced than the FORTRAN team.

However, the Ada team is less experienced with dynamics simulators, the application area of interest.

Striking differences exist in the relationships of the teams to their development tasks (Figure 8). The FORTRAN team is able to reuse some design and code from related systems. The Ada team is charged with starting fresh to design a system that can take advantage of Ada-related design approaches. For the Ada team, both the development environment and the language are new.

Figure 9 shows the timeline for the Ada experiment with the activities of the three teams during the expected 2-year duration of the experiment. The timeline shows the FORTRAN team to be slightly more than one development phase ahead of the Ada team. The shift is due to the training in Ada required by the Ada team at the start of the project. The FORTRAN team, by contrast, was able to start immediately with the requirements analysis activity--the first phase in the development process.

The study team is collecting data on both development teams. Figure 10 shows the range of resource, project, and product data collected. Wherever possible, routine SEL forms were used. However, special Ada versions of two forms--the component origination form and the change report form--were developed. The new component form allows the identification of an Ada component as a package, task, generic, or subprogram and further recognizes that a component can be a specification or body. The new change form adds a section to identify separately any Ada-related errors.

TRAINING APPROACHES

A major portion of the experiment thus far has been the Ada training program, which was planned by the study team, in

particular by the University of Maryland personnel. The principal training resources (Figure 12) were as follows:

- Ada language reference manual (LRM) (Reference 6)
- Ada textbook (Reference 3)
- Ada videotapes (Reference 7)

The 27 videotapes were viewed by the team over a 1-week period. A University of Maryland graduate student, experienced in Ada, was available to direct the training--that is, to plan the schedule of tape viewing, answer questions about Ada material, stop the tapes to clarify the material, lead the discussion between tapes, and assign reading and small coding assignments. Two sets of diskettes for use on personal computers were available to the team to supplement the videotaped instructions. Lectures on Ada-related design methods--the state-machine abstraction and process abstraction method (Reference 4)--were presented to the team.

A principal component of the Ada training program was the design and implementation in Ada of a practice problem. The purpose of this training exercise was to enable the team to apply what it had been taught about Ada and to begin working together as a team.

Figure 13 shows the coverage of topics by the training elements. The textbook and the training exercise covered all three training topics: the Ada language itself, software engineering with Ada, and Ada-related design methods.

Experience with Ada training led to several recommendations for future sessions (Figure 14). Consistent with several other published recommendations (e.g., Reference 3), the appropriate emphasis should be on software engineering with Ada and not simply the language syntax and semantics. The methods and resources used in training the Ada team--videotapes, class discussion, and a practice problem--were

effective. Additional hands-on experience with the Ada compiler (in addition to work on the practice problem) is also beneficial.

Two months of full-time training are recommended for each staff member. After this period, the staff member would be able to join a development team and begin contributing. Ideally, this first assignment as a developer should be carefully chosen and closely monitored by a more senior developer. Reference 8 contains a more thorough assessment of Ada training methods and more detailed recommendations for the design of future Ada training programs.

DATA FROM THE ADA TRAINING EXERCISE

The training exercise (or practice problem) emerged as the single most valuable element of Ada training. It also provided the study team with an opportunity to practice monitoring a small Ada project.

The exercise was to design and develop an electronic message system (EMS) that allows users to send and receive electronic mail and to manage groups of users (Figure 16). EMS has been used as a student programming project at the University of Maryland, where it was implemented in the SIMPL language, requiring typically 1000 to 2000 lines of code.

For the Ada team, EMS was a chance to practice object-oriented design as well as to experiment with Ada. The study team could try out the data collection system and begin measuring a small Ada development.

The completed EMS system in Ada comprised 5730 lines of code (Figure 17), much larger than the student projects in SIMPL. An analysis is currently underway to compare the functionality of the Ada and SIMPL versions. It is already clear that

the Ada version has a much more extensive user interface and help facility. Also, the 5730 source lines contained only 1402 executable statements. The drop from source lines to executable statements is more severe than in SEL FORTRAN systems, where reductions of only 2 to 1 are typical.

Developing EMS required 1906 staff-hours (including 570 hours of training). A productivity/cost measure frequently used in the SEL is the number of hours per thousand executable statements. Figure 17 shows the cost of EMS development to be greater than the average cost of developing FORTRAN systems. Of course, the EMS example in Ada represents only a single data point whereas the FORTRAN cost data are taken from hundreds of FORTRAN modules in the SEL data base.

It is wise not to rely too heavily on the EMS data as an indicator of future Ada projects. There are several sound reasons why the costs could be higher or lower than those experienced with EMS.

Costs could be higher in the future because of the following:

- EMS was developed by a highly motivated staff eager to apply Ada. As the use of Ada becomes more routine, the staff may not be as motivated by the novelty of using a new language in an experimental setting.

- EMS had no documentation requirements, unlike typical SEL projects.

- EMS did not involve tasking.

- The application domain of EMS (electronic mail) was easier to understand than the flight dynamics area. As a result, the EMS effort in requirements analysis and acceptance testing was proportionally less than it would be for flight dynamics projects.

Costs of the Ada development may actually be lower than suggested by EMS because of the following:

- The staff will be better trained. Recall that EMS was a training exercise; teams in the future will be more experienced in Ada.

- The Ada team (with seven people) was too large for the EMS assignment. The size of the team was driven by the scope of the GRO dynamics simulator development. The cost of EMS would likely have been less if the team were smaller (approximately three people).

- The Ada development environment for EMS was not only new but also highly unstable. Only unvalidated Ada compilers were available when coding of EMS began. The team progressed through versions 1.3, 1.5, and 2.1 of the Telesoft compiler before the DEC Ada compiler arrived.

Figure 17 shows that the error rate for EMS was lower than that of FORTRAN systems in the SEL data base. Once again, this result should not necessarily be attributed to the use of Ada on EMS. The FORTRAN systems are much more complex, and the testing requirements in the flight dynamics area are much more rigorous than for EMS.

Figure 18 shows the distribution of effort among design, code, and test for EMS and typical FORTRAN systems. Whereas the relative effort for the three activities is roughly equivalent for FORTRAN systems, 60 percent of the EMS Ada effort was spent on design. Of course, the use of Ada raises the question of redefining the cutoff between design and code activities. If Ada is used as a process design language (PDL), the design activity can include the delivery of a design document of compiled specifications, Ada definitions of types, and Ada PDL. In such cases, it may be

understandable that more effort is spent on "design" activity, with proportionally less effort on "code." Again, the more substantial testing requirements for FORTRAN flight dynamics systems may explain the difference in relative effort devoted to testing EMS versus typical FORTRAN systems.

The profile of the EMS code in Figure 19 reveals that the EMS Ada modules were smaller on average. The lower percentage of lines of EMS that are blank or comment (39 percent versus 51 percent) may be due to the greater self-description possible with Ada object names and types.

STATUS AND OBSERVATIONS

Figure 21 revisits the experiment timeline to show the actual activity to date. The activity profiles of the two development teams confirm that progress is being made according to plan.

With the Ada experiment not yet complete, no definitive statements can be made on the effectiveness of Ada in the SEL environment. Nevertheless, Ada's influence is being felt on personnel issues, software products, the development environment, and the software development process (Figure 22).

The clearest observations relate to the activity that has dominated the early phases of the experiment--training. The need for effective training is real and should be included explicitly in Ada development plans. Training will occur whether or not it is scheduled; wise managers will plan for it. Two months of full-time training appears to be the right amount. The training exercise emerged as an extremely effective method and is strongly recommended.

The use of Ada led to a larger product than the student versions of EMS in SIMPL. It is premature to state whether Ada products will continue to be larger. EMS did demonstrate that many more design relations are expressible in Ada. The use of Ada will likely lead to changes in recommended intermediate products, for example, at design reviews. Current recommendations are oriented to FORTRAN implementations, so the design products highlight the invocation structure of the code. Ada design products can express other relations in addition to invocation--for example, the "uses" relation, exception handling, and the management of the name space.

The use of Ada has not degraded the performance of the development environment. Stress test are now in progress, but the early indications are that the use of the DEC Ada Compilation System (ACS) is not adversely affecting the performance of the system. Both compilation time and execution time appear to be within acceptable limits, although more complete testing is being performed.

The most important tool is a validated compiler. The DEC ACS has demonstrated that it is a production-quality system. Although other Ada support tools may be used by the team in the future, the DEC ACS has been adequate by itself to support development. The library management facility built into the ACS has been especially helpful.

Although such conculsions may appear less than daring, the Ada experiment has demonstrated that Ada is learnable and that an Ada project is measurable. The results thus far lead the study team to be optimistic that they will be able to meet their experimental objectives and establish an empirical basis for understanding the effect of Ada in the flight dynamics software development environment.

ACKNOWLEDGMENTS

The Ada experiment is managed by F. McGarry and R. Nelson of NASA/GSFC and actively supported by representatives from all SEL participating organizations (NASA/GSFC, CSC, and the University of Maryland)--especially V. Basili, E. Katz, Y. Benoit, G. Page, and V. Church.

REFERENCES

1. Software Engineering Laboratory, SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

2. Webster's New World Dictionary, World Publishing Co., New York

3. G. Booch, Software Engineering With Ada. Menlo Park, California: Benjamin/Cummings Publishing Co., Inc., 1983

4. G. W. Cherry, "Advanced Software Engineering With Ada--Process Abstraction Method for Embedded Large Applications," Language Automation Associates, Reston, Virginia, 1985

5. W. Agresti, "An Approach to Developing Specification Measures," Proceedings, Ninth Annual Software Engineering Workshop, NASA/GSFC, November 1984

6. American National Standards Institute, Inc., ANSI/MIL-STD-1815A-1983, Reference Manual for the Ada Language, February 17, 1983

7. Alsys, Inc., Waltham, Mass., "Ichbiah, Barnes, and Firth on Ada," videotape series, 1983

8. R. Murphy and M. Stark, Ada Training Evaluation and Recommendations, SEL-85-002, NASA/GSFC, October 1985

THE VIEWGRAPH MATERIALS

for the

W. AGRESTI PRESENTATION FOLLOW

FIGURE 1

MEASURING ADA* AS A SOFTWARE DEVELOPMENT TECHNOLOGY IN THE SEL

BILL AGRESTI
(COMPUTER SCIENCES CORPORATION)

AND THE SEL STAFF

*ADA IS A REGISTERED TRADEMARK OF THE U.S. GOVERNMENT (ADA JOINT PROGRAM OFFICE).

FIGURE 2

TECHNOLOGY ASSESSMENT IN THE SEL

REQUIREMENTS LANGUAGES

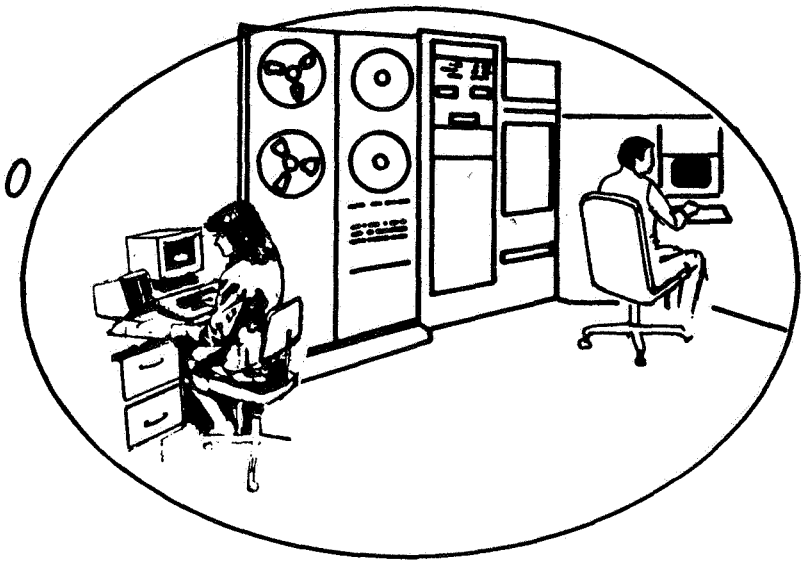
WORKSTATIONS

COST MODELS

TESTING TECHNIQUES

IV&V

CHIEF PROGRAMMER TEAM



NASA/GSFC FLIGHT DYNAMICS SOFTWARE DEVELOPMENT ENVIRONMENT

- SCIENTIFIC GROUND SYSTEMS
- 85% FORTRAN, 15% ASSEMBLER MACROS
- IBM-COMPATIBLE MAINFRAMES, DEC VAX
- SIZE: 3K TO 160K SOURCE LINES OF CODE

FIGURE 3

OBJECTIVES

- DETERMINE COST-EFFECTIVENESS AND IMPACT OF ADA
- ASSESS EFFECTIVENESS OF OBJECT-ORIENTED DESIGN (OOD), COMPOSITE SPECIFICATION MODEL (CSM), ETC.
- DEVELOP APPROACHES FOR REUSABLE SOFTWARE
- DEVELOP MEASURES FOR SPACE STATION

FIGURE 4

- **EXPERIMENT PLANNING**
- **TRAINING APPROACHES**
- **DATA FROM ADA TRAINING EXERCISE**
- **STATUS AND OBSERVATIONS**

831-AGR-(109m)*

FIGURE 5

SEL ADA EXPERIMENT

- PARALLEL DEVELOPMENT IN FORTRAN AND ADA
- PROJECT: GAMMA RAY OBSERVATORY (GRO) DYNAMICS SIMULATOR
  - SIZE (ESTIMATED): 40,000 (FORTRAN) SOURCE LINES OF CODE
  - DURATION: 18 TO 24 MONTHS
  - ENVIRONMENT: VAX-11/780
  - STAFFING: 7 PEOPLE
  - EFFORT: 8 TO 10 STAFF-YEARS

FIGURE 6

EXPERIMENT ORGANIZATION

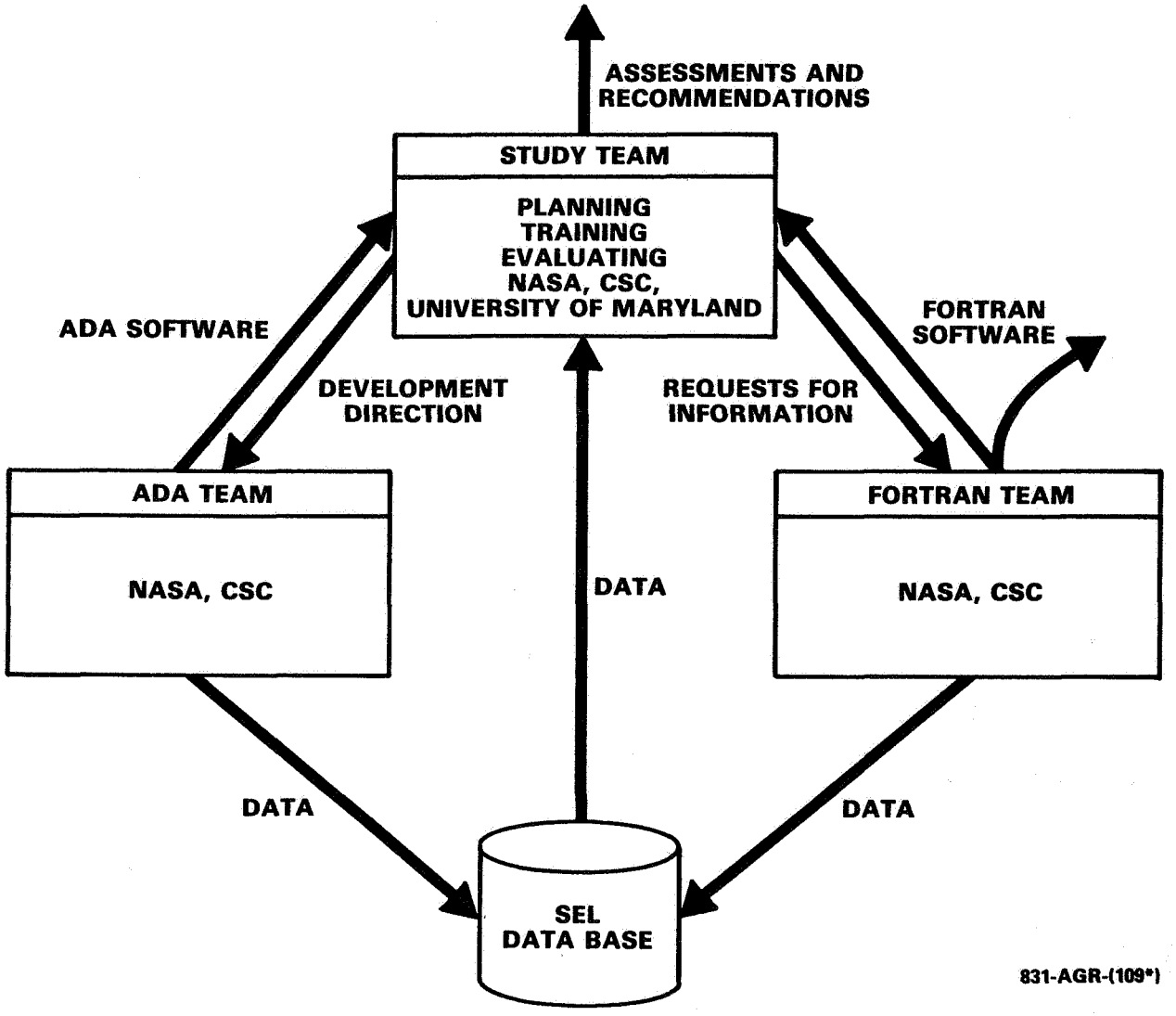


FIGURE 7

TEAM PROFILES

CHARACTERISTIC	FORTRAN TEAM	ADA TEAM
NUMBER OF LANGUAGES KNOWN (MEDIAN)	3	7
NUMBER OF TYPES OF APPLICATION EXPERIENCE (MEDIAN)	3	4
NUMBER OF YEARS OF SOFTWARE DEVELOPMENT EXPERIENCE (MEAN)	4.8	8.6
TEAM MEMBERS WITH DYNAMICS SIMULATOR EXPERIENCE	66%	43%

831-AGR-(109*n)

FIGURE 8

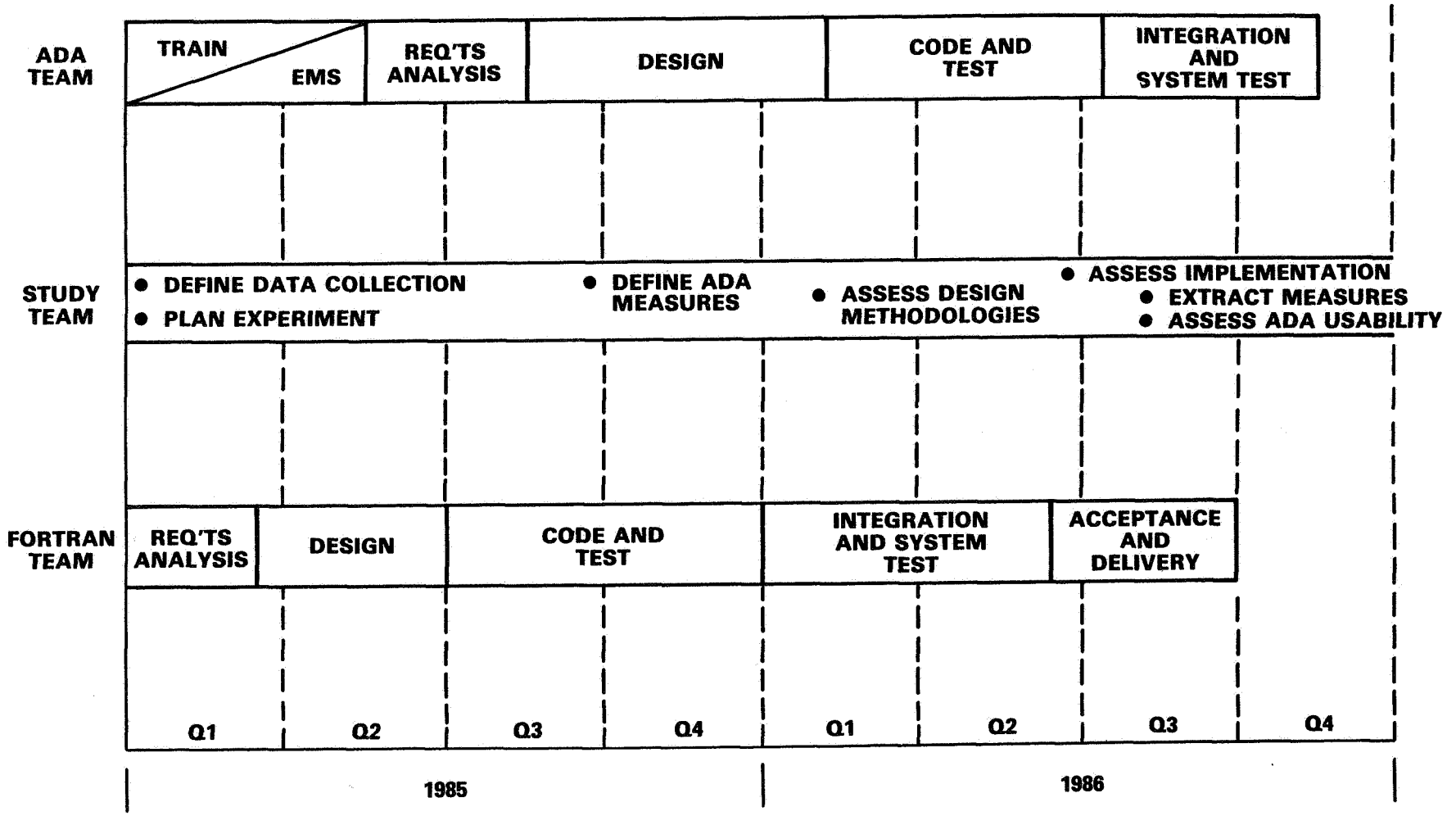
TEAM DIFFERENCES

CHARACTERISTIC	FORTRAN TEAM	ADA TEAM
DESIGN HERITAGE	SIMILAR TO PAST SYSTEMS	NEW DESIGN APPROACH
CODE REUSE	15 TO 30%	NONE
DEVELOPMENT ENVIRONMENT	STABLE	NEW COMPILER
LANGUAGE/ METHODOLOGY EXPERIENCE	AVERAGE	NEW LANGUAGE/ METHODOLOGY

831-AGR-(109*)

FIGURE 9

SCHEDULE



831-AGR-(109*)

FIGURE 10

DATA COLLECTION

RESOURCES	PROJECT	PRODUCT
• EFFORT – MANAGEMENT, TECHNICAL, SUPPORT – BY ACTIVITY – BY COMPONENT	• STAFF PROFILE • METHODOLOGY/TOOL USAGE • ESTIMATION OF SIZE, EFFORT, SCHEDULE	• GROWTH HISTORY • DOCUMENTATION • ERRORS
• COMPUTER USAGE	• SUBJECTIVE EVALUATION	• CHANGES

831-AGR-(109*)

FIGURE 11

- **EXPERIMENT PLANNING**
- **TRAINING APPROACHES**
- **DATA FROM ADA TRAINING EXERCISE**
- **STATUS AND OBSERVATIONS**

831-AGR-(109*aa)

FIGURE 12

TRAINING RESOURCES

- ADA LANGUAGE REFERENCE MANUAL (LRM)
- TEXTBOOK — GRADY BOOCH, "SOFTWARE ENGINEERING WITH ADA"
- VIDEOTAPES — ALSYS, INC.
- DISKETTES — HYPERGRAPHICS, INC.
  — ALSYS, INC.
- GEORGE CHERRY SEMINAR

831-AGR-(109*aa)

FIGURE 13

COVERAGE OF TRAINING APPROACHES

TOPIC	ADA LRM	TEXT	VIDEOTAPES, DISKETTES, DISCUSSIONS	LECTURES	TRAINING EXERCISE
ADA LANGUAGE	●	●	●		●
SOFTWARE ENGINEERING WITH ADA		●	●	●	●
VARIOUS DESIGN METHODS		●		●	●

831-AGR-(109*cc)

FIGURE 14

TRAINING RECOMMENDATIONS

- **TOPIC — SOFTWARE ENGINEERING WITH ADA**
- **METHODS — VIDEOTAPES**
  CLASS DISCUSSION
  HANDS-ON EXPERIENCE
  TRAINING EXERCISE
- **TIME PERIOD — 2 MONTHS FULL-TIME**

831-AGR-(109z*)

FIGURE 15

- **EXPERIMENT PLANNING**
- **TRAINING APPROACHES**
- **DATA FROM ADA TRAINING EXERCISE**
- **STATUS AND OBSERVATIONS**

FIGURE 16

ADA TRAINING EXERCISE
ELECTRONIC MESSAGE SYSTEM (EMS)

- **FUNCTION:** SEND/RECEIVE ELECTRONIC MAIL; MANAGE USERS AND GROUPS
- **HERITAGE:**
  - STUDENT GROUP PROJECT AT UNIVERSITY OF MARYLAND
  - 1000-2000 SOURCE LINES OF SIMPL CODE
- **DESIGN APPROACH:** OBJECT-ORIENTED DESIGN

FIGURE 17

ADA EMS PROJECT SUMMARY

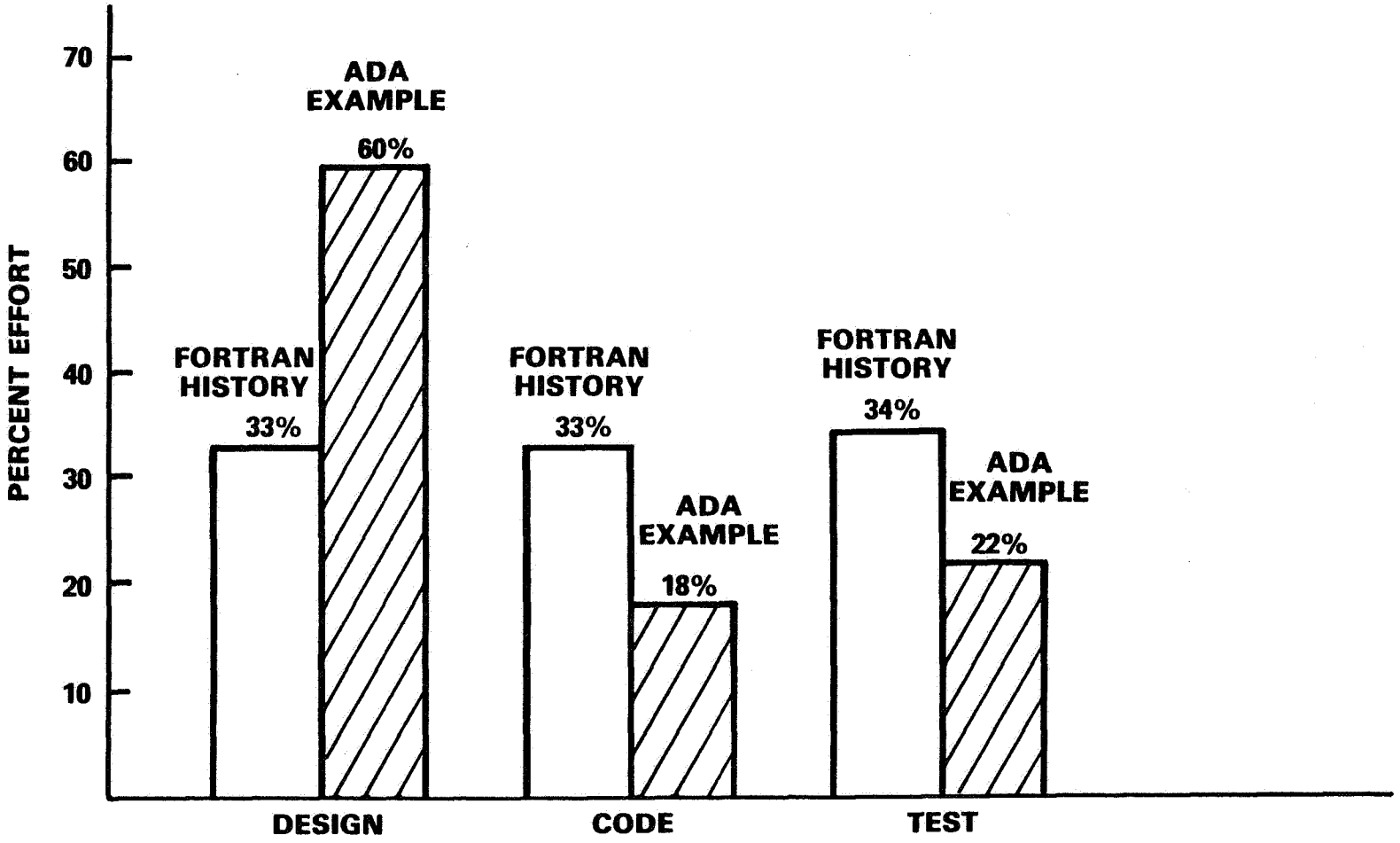
SIZE	EFFORT
• 5730 SOURCE LINES OF CODE	• 1336 HOURS
• 1402 EXECUTABLE STATEMENTS	• 570 TRAINING HOURS
	1906 TOTAL HOURS

	ADA EMS EXAMPLE		FORTRAN HISTORY
COST (HOURS/1000 EXEC. STMTS.)	950	(WITHOUT TRAINING HRS.)	720
	1360	(WITH TRAINING HRS.)	
ERROR RATE (ERRORS/1000 EXEC. STMTS.)	9		12

831-AGR-(109*bb)

FIGURE 18

EFFORT DISTRIBUTION BY ACTIVITY TYPE



831-AGR-(109*aa)

FIGURE 19

PROFILE OF ADA EMS CODE

	FORTRAN HISTORY	ADA EMS EXAMPLE
MODULE SIZE		
• LINES OF CODE	232	143
• EXECUTABLE STATEMENTS	69	35
LINES		
• BLANK OR COMMENT	51%	39%
• PROGRAM TEXT	49%	61%
STATEMENTS		
• DECLARATIONS	42%	32%
• EXECUTABLE	58%	68%

831-AGR-(109*bb)

FIGURE 20

- **EXPERIMENT PLANNING**
- **TRAINING APPROACHES**
- **DATA FROM ADA TRAINING EXERCISE**
- **STATUS AND OBSERVATIONS**

831-AGR-(109*bb)

FIGURE 21

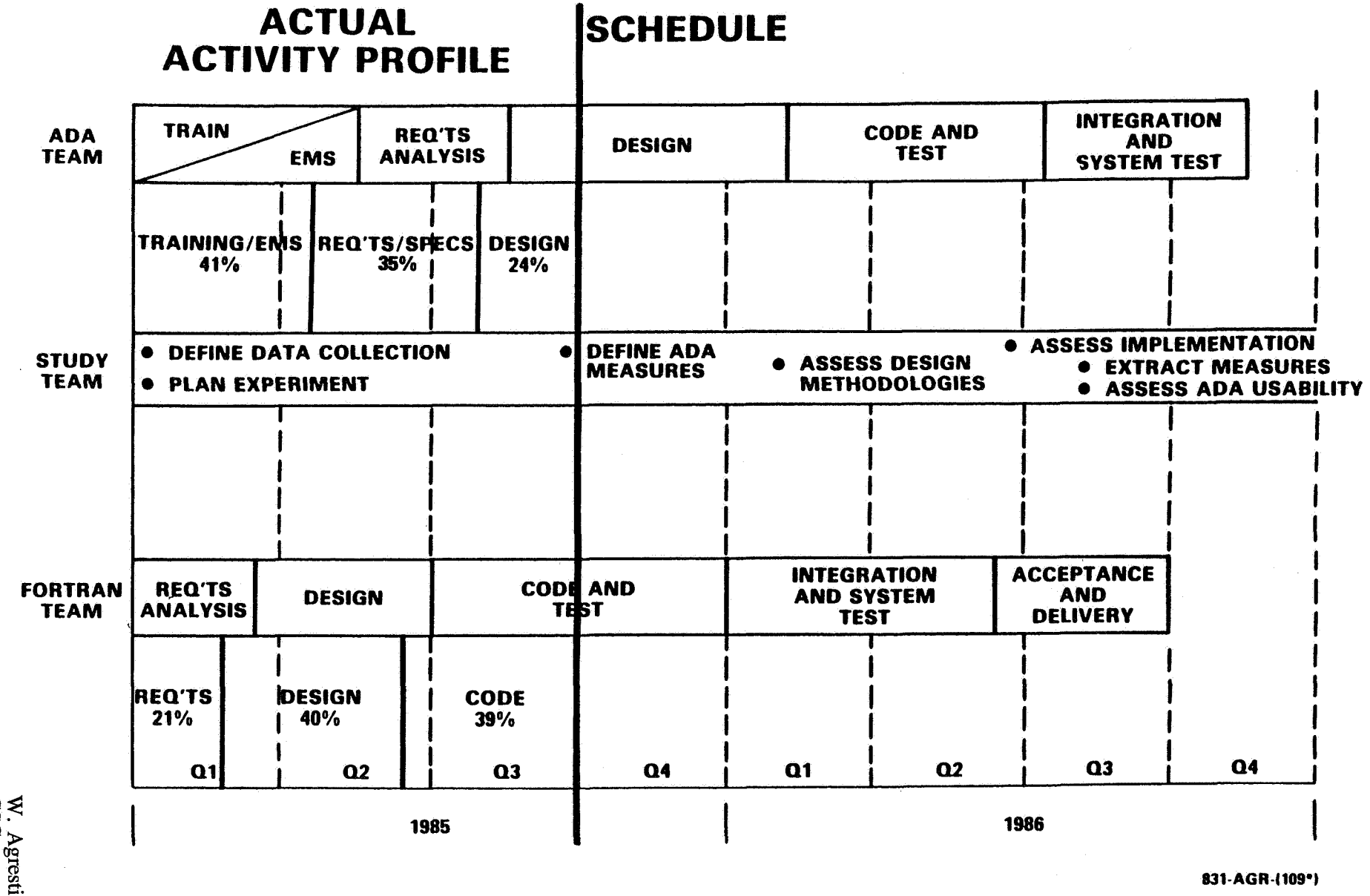

ACTUAL ACTIVITY PROFILE
SCHEDULE
ADA TEAM
TRAIN
EMS
REQ'TS ANALYSIS
DESIGN
CODE AND TEST
INTEGRATION AND SYSTEM TEST
TRAINING/EMS 41%
REQ'TS/SPECS 35%
DESIGN 24%
STUDY TEAM
• DEFINE DATA COLLECTION
• PLAN EXPERIMENT
• DEFINE ADA MEASURES
• ASSESS DESIGN METHODOLOGIES
• ASSESS IMPLEMENTATION
• EXTRACT MEASURES
• ASSESS ADA USABILITY
FORTRAN TEAM
REQ'TS ANALYSIS
DESIGN
CODE AND TEST
INTEGRATION AND SYSTEM TEST
ACCEPTANCE AND DELIVERY
REQ'TS 21%
DESIGN 40%
CODE 39%
Q1
Q2
Q3
Q4
Q1
Q2
Q3
Q4
1985
1986
831-AGR-(109*)

FIGURE 22

OBSERVATIONS ON THE IMPACT OF ADA

RESULTS OF THE

WORKSHOP

QUESTIONNAIRE

RESULTS OF THE WORKSHOP QUESTIONNAIRE

W. W. Agresti

Computer Sciences Corporation

To help mark the tenth anniversary of the Software Engineering Workshop, the planning committee distributed a questionnaire to everyone on the workshop mailing list (approximately 1000 people). The purpose of the questionnaire was to obtain information from the respondents concerning their

- Role in software development
- Data collection activity
- Perception of changes in the quality of software
- Opinions regarding the progress (or lack thereof) in various areas of software engineering

Figure 1 shows the questionnaire that was distributed; 195 were completed and returned. The results are summarized in Figures 2 through 4.

Figure 2 shows the answers to the first five questions. Approximately 69 percent of the respondents collect some data on software development, and a similar percentage have been able to use Software Engineering Laboratory (SEL) documents or workshop results. The quality of software has improved both nationally and in the respondents' own organizations.

Figure 3 summarizes the results of questions 6 and 7 on areas of software engineering that have experienced the greatest improvement and the most disappointing progress. Tools and methods have provided the greatest improvements over the past 5 to 10 years. Metrics and management are cited as areas of greatest improvement by only 8 percent

of the respondents, while 52 percent list these areas as the biggest disappointments. These results may be related to the experiences of the SEL over the past decade as recounted by V. Basili elsewhere in the proceedings of this workshop. His conclusion is that collecting data and administering a program aimed at software technology improvement is a difficult undertaking. It is very easy for an organization to make mistakes and thus not obtain the benefits anticipated. Perhaps the reported disappointment with metrics and management is due to high expectations that have been unmet because the metrics and management programs have been difficult to implement successfully.

Figure 4 shows a sample of the write-in selections for areas of improvement and disappointment. Tables 1 through 7 provide the complete numerical results and show how respondents in different categories (manager, developer, etc.) answered each question.

Overall, the questionnaire succeeded in obtaining a sample of opinions on issues in software engineering.

ACKNOWLEDGMENT

John Cook of NASA/GSFC maintained the questionnaire data and results.

QUESTIONNAIRE
TENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP
For each question, please check one option.

1. What is your role in software development?
   - _ manager
   - _ developer
   - _ product assurance
   - _ teacher
   - _ researcher
   - _ student

2. Does your organization collect internal data (e.g., on effort, errors, changes) on software development projects?
   - _ yes
   - _ no

3. Has your organization been able to use information from past NASA/SEL workshops or NASA/SEL documents?
   - _ yes
   - _ no
   - _ never attended SEL workshops; don't have SEL documents

4. What has happened to the quality of software in your organization over the past 5-10 years?
   - _ greatly improved
   - _ improved somewhat
   - _ stayed about the same
   - _ quality has declined

5. What, in your opinion, has happened to the quality of software nationally over the past 5-10 years?
   - _ greatly improved
   - _ improved somewhat
   - _ stayed about the same
   - _ quality has declined

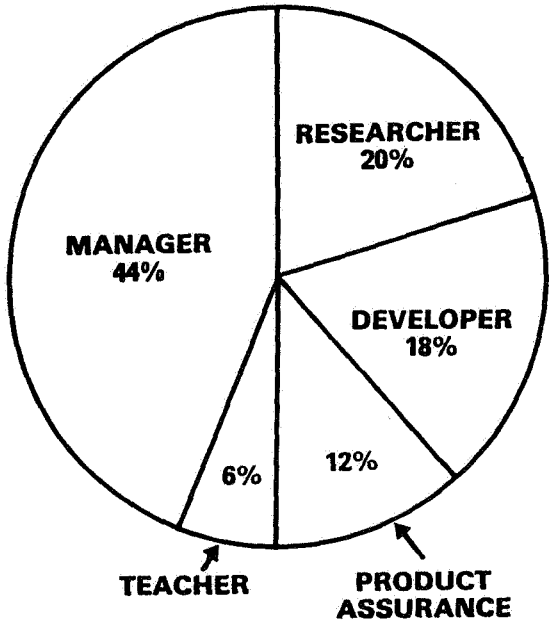
6. In what area of software engineering has there been the greatest improvement in the state-of-the-art over the past 5-10 years?
   - _ standards
   - _ software tools
   - _ methods or practices
   - _ languages
   - _ metrics
   - _ management
   - _ quality of people
   - _ other -- please specify:_____

7. What area of software engineering has had the most disappointing progress over the past 5-10 years?
   - _ standards
   - _ software tools
   - _ methods or practices
   - _ languages
   - _ metrics
   - _ management
   - _ other -- please specify:_____

Please return to Mr. Frank McGarry, Code 552, NASA/Goddard Space Flight Center,Greenbelt, MD 20771
Results will be summarized at the Tenth Annual Software Engineering Workshop.

Figure 1. Questionnaire - Tenth Annual Software Engineering Workshop

1. YOUR ROLE?



2. COLLECT DATA?

YES	69%
NO	31%

3. USE SEL RESULTS?

YES	68%
NO	8%
N/A	24%

4-5. QUALITY OF SOFTWARE?

	YOUR ORGANIZATION	NATIONALLY
GREATLY IMPROVED	29%	16%
IMPROVED SOMEWHAT	57%	69%
STAYED THE SAME	12%	13%
DECLINED	2%	2%

831-AGR-(110)

Figure 2. Questionnaire Results

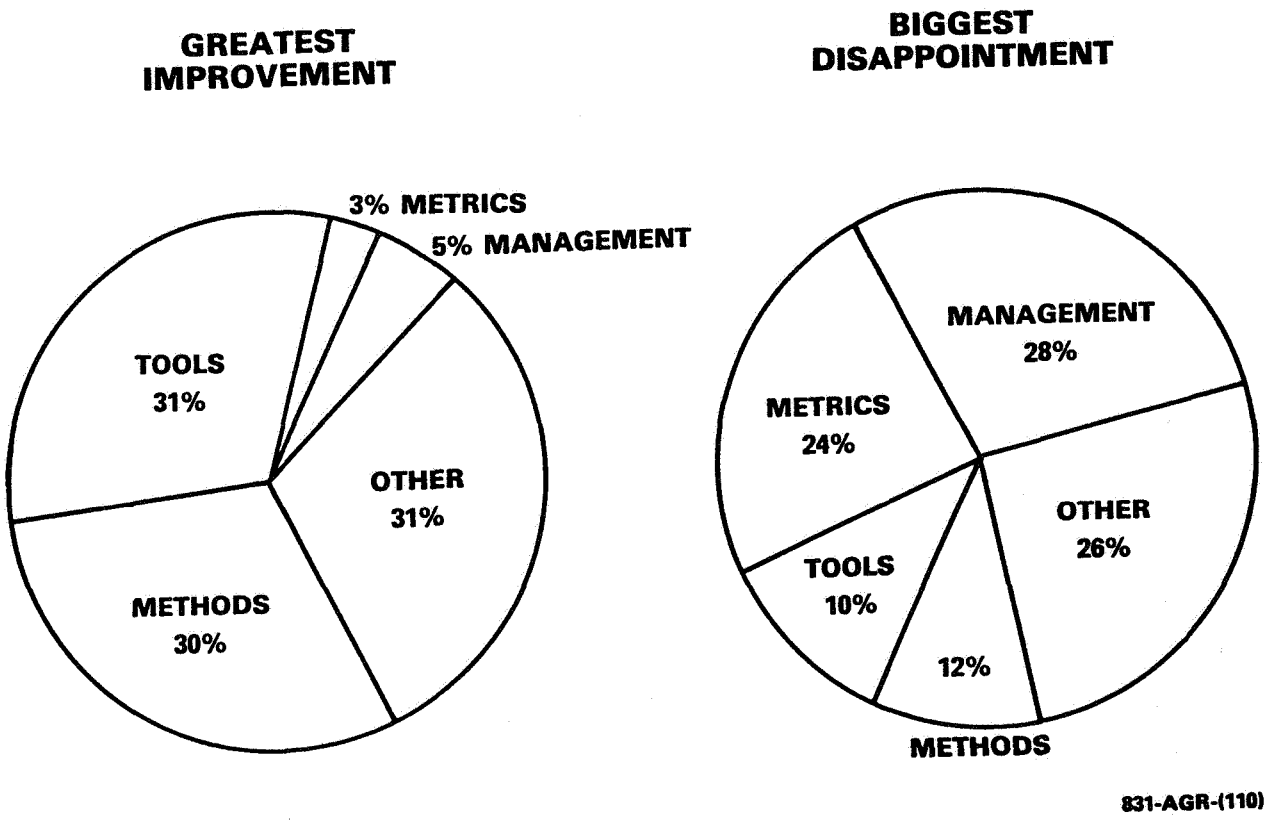


Figure 3. Responses to Questions 6 and 7 on Changes in Software Engineering

"WRITE-IN" VOTES

AREAS OF SOFTWARE ENGINEERING...

- **GREATEST IMPROVEMENT**
  - **PCs/MICROS — SOFTWARE PACKAGES**
  - **"USER FRIENDLINESS"/HUMAN FACTORS**
  - **JAPANESE SOFTWARE FACTORIES**
  - **"NONE"**
- **BIGGEST DISAPPOINTMENT**
  - **SOFTWARE SIZE ESTIMATING**
  - **DESIGN PROCESS**
  - **TECHNOLOGY TRANSFER**
  - **"ALL AREAS"**

831-AGR-(110)

Figure 4. "Write-in" Votes

Table 1. Question 1: What Is Your Role in Software Development?

ROLE CATEGORY	RESPONDENTS*
TOTAL QUESTIONNAIRES RECEIVED	195
MANAGER	96
DEVELOPER	40
RESEARCHER	44
PRODUCT ASSURANCE	26
TEACHER	12
STUDENT	0

0129 (117*al/86

*THE SUM OF THE QUESTIONNAIRES RECEIVED BY CATEGORY IS GREATER THAN 195 BECAUSE SOME PEOPLE CHECKED MORE THAN ONE CATEGORY.

Table 2. Question 2: Does Your Organization Collect Internal Data (e.g., on effort, errors, changes) on Software Development Projects?

ROLE CATEGORY	RESPONSE	
	YES	NO
TOTAL RESPONSES	134	60
MANAGER	73	22
DEVELOPER	28	12
RESEARCHER	23	21
PRODUCT ASSURANCE	19	7
TEACHER	8	4

0129 (117*al 86

Table 3. Question 3: Has Your Organization Been Able To Use Information From Past NASA/SEL Workshops or NASA/SEL Documents?

ROLE CATEGORY	RESPONSE		
	YES	NO	N/A
TOTAL RESPONSES	132	16	46
MANAGER	68	7	21
DEVELOPER	22	2	16
RESEARCHER	33	3	8
PRODUCT ASSURANCE	14	4	7
TEACHER	10	0	2

0129-(117*al/86

Table 4. Question 4: What Has Happened to the Quality of Software in Your Organization Over the Past 5-10 Years?

ROLE CATEGORY	RESPONSE			
	GREATLY IMPROVED	SOMEWHAT IMPROVED	STAYED SAME	QUALITY DECLINED
TOTAL RESPONSES	52	105	22	4
MANAGER	27	49	10	3
DEVELOPER	12	25	2	0
RESEARCHER	10	23	8	1
PRODUCT ASSURANCE	8	17	2	0
TEACHER	4	6	1	0

0129-(117*)/86

Table 5. Question 5: What Has Happened to the Quality of Software Nationally Over the Past 5-10 Years?

ROLE CATEGORY	RESPONSE			
	GREATLY IMPROVED	SOMEWHAT IMPROVED	STAYED SAME	QUALITY DECLINED
TOTAL RESPONSES	32	134	26	4
MANAGER	16	62	17	2
DEVELOPER	5	30	4	1
RESEARCHER	5	33	5	1
PRODUCT ASSURANCE	4	18	3	1
TEACHER	2	8	1	1

0129 (117*)/86

Table 6. Question 6: In What Area of Software Engineering Has There Been the Greatest Improvement in the State-of-the-Art Over the Past 5-10 Years?

SOFTWARE ENGINEERING AREA	TOTAL RESPONSES	RESPONSES BY ROLE CATEGORY				
		MANAGER	DEVELOPER	RESEARCHER	PRODUCT ASSURANCE	TEACHER
STANDARDS	30	11	7	6	9	1
SOFTWARE TOOLS	79	39	22	16	9	5
METHODS/PRACTICES	76	44	9	18	10	4
LANGUAGES	30	14	8	8	1	1
METRICS	7	5	0	3	1	0
MANAGEMENT	14	7	0	3	2	1
QUALITY OF PEOPLE	12	10	1	1	0	0
OTHER	10	4	3	1	3	2

0129-1117-1 86

Table 7. Question 7: What Area of Software Engineering Has Had the Most Disappointing Progress Over the Past 5-10 Years?

SOFTWARE ENGINEERING AREA	TOTAL RESPONSES	RESPONSES BY ROLE CATEGORY				
		MANAGER	DEVELOPER	RESEARCHER	PRODUCT ASSURANCE	TEACHER
STANDARDS	25	17	6	2	3	2
SOFTWARE TOOLS	24	17	1	5	2	0
METHODS/PRACTICES	27	12	7	9	3	2
LANGUAGES	18	13	2	2	0	1
METRICS	55	27	7	12	9	6
MANAGEMENT	64	32	17	12	9	2
QUALITY OF PEOPLE	2	1	1	0	0	0
OTHER	14	5	2	6	1	2

0129-(117)-86

ATTENDEES OF THE 1985 SOFTWARE ENGINEERING WORKSHOP

ADAIR, B.	NASA/MSFC
AGRESTI, W.	COMPUTER SCIENCES CORP
AICHEOE, D.	NASA/MSFC
ANDREW, E.	
ARNOLD, R.	MITRE CORP
ARTHUR, S.	VIRGINIA POLYTECH
ASTILL, P.	SIGMA DATA SERVICES
ATZINGER, E.	ABERDEEN PROVING GROUNDS
AYERS, E	ARINC INC
BABST, T.	HARRIS CORP
BACA, J.	KIRKLAND AIR FORCE BASE
BALTER, L.	
BARRETT, C.	NASA/GSFC
BASILI, V.	UNIVERSITY OF MARYLAND
BAUMERT, J.	SPACE TELESCOPE SCIENCE INSTITUTE
BAYNES, P.	VITRO
BELLARD, B.	F C C
BENOIT, Y.	UNIVERSITY OF MARYLAND
BERG, R.	COMPUTER SCIENCES CORP
BIGWOOD, D.	U S D A
BISHOP, J.	NASA HEADQUARTERS
BITAR, I.	T R W
BOEHM-DAVIS, D.	GEORGE MASON UNIVERSITY
BOLAND, D.	INTERNAL REVENUE SERVICE
BOND, J.	N S A
BOND, R.	ARINC INC
BOONE, D.	COMPUTER SCIENCES CORP
BORGES, C.	
BOROCHOFF, R.	DEPT. OF JUSTICE
BOYER, R.	NASA/MSFC
BRASLAU, R.	T R W
BREDESON, R.	O A O
BREDESON, M.	SPACE TELESCOPE SCIENCE INSTITUTE
BRENNEMAN, D.	INTERNAL REVENUE SERVICE
BRETT, D.	LOCKHEED
BRINKER, E.	NASA/GSFC
BROWN, D.	WASHINGTON NAVY YARD
BUELL, J.	COMPUTER SCIENCES CORP
CARD, D.	COMPUTER SCIENCES CORP
CARMODY, C.	PLANNING RESEARCH CORP
CASHOUR, J.	N S A
CEPHAS, A.	NASA/GSFC
CHEEK, A.	NASA/GSFC
CHENOWETH, H.	WESTINGHOUSE
CHRISTELLER, H.	
CHU, R.	MARTIN MARIETTA
CHUNG, A.	F A A
CHURCH, V.	COMPUTER SCIENCES CORP
CISNEY, L.	NASA/GSFC
CLAY, W.	ABERDEEN PROVING GROUNDS

CLAYTON, J.	LOCKHEED
CLEMENTS, P.	NAVAL RESEARCH LABS
CLIFTON, C.	INTERNAL REVENUE SERVICE
CLINEDINST, W.	COMPUTER SCIENCES CORP
CLUBB, K.	IITRI
COHEN, J.	T R W
COHEN, V.	E P A
COOK, J.	NASA/GSFC
COOK, L.	G S C
COPP, P.	F A A
COUCHOUD, C.	SOCIAL SECURITY ADMIN
COYNE, C.	BURROUGHS
CRAFT, R.	NASA/MSFC
CRUICHSHANK, R.	IBM CORP
CYPRYCH, G.	IBM CORP
CZYSCON, C.	ROME AIR DEVELOPMENT CTR
DANIELE, C.	NASA/LERC
DASHIELL, C.	IBM CORP
DASKALANTONAKIS, M.	
DECKER, W.	COMPUTER SCIENCES CORP
DELONG, S.	COMPUTER SCIENCES CORP
DICKSON, C.	U S D A
DIECKHANS, R.	F C C
DILDY, C.	F C C
DOIRON, M.	IITRI
DOLBERG, S.	WESTINGHOUSE
DOUBLEDAY, D.	UNIVERSITY OF MARYLAND
DUNHAM, J.	RESEARCH TRIANGLE INST
DUNIHO, M.	N S A
EBERHART, H.	IITRI
ELLIS, J.	T R W
ELLIS, W.	IBM CORP
ENG, E.	NASA/GSFC
ESFANDIARI, M.	NASA/GSFC
FABISZAK, C.	LOCKHEED
FANG, A.	NASA HEADQUARTERS
FISHKIND, S.	
FOERTSCH, D.	DIGITAL EQUIPMENT CORP
FORSYTHE, R.	NASA/WALLOPS
FRANKEL, S.	NAT'L BUREAU OF STANDARDS
FRANKS, C.	
FRYER, R.	G T E
GANNETT, M.	N S A
GARBER, O.	
GARY, J.	NASA/GSFC
GIESE, C.	STARS
GILL, C.	BOEING COMPUTER CORP.
GINTNER, M.	VIRGINIA POLYTECH
GODFREY, S.	NASA/GSFC
GOLDBERG, A.	MARTIN MARIETTA
GOLDEN, J.	EASTMAN KODAK
GORDON, D.	JET PROPULSION LAB

GRAHAM, D.	NASA/GSFC
GREEN, D.	PENTAGON
GREEN, S.	NASA/GSFC
GREENGRASS, E.	N S A
GRIEF, S.	APL
GRIEN, S.	IITRI
GRIM, C.	IBM CORP
GRIMES, G.	PLANNING RESEARCH CORP
GROVER, J.	GEORGIA TECH
HALTERMAN, K.	O A O
HANNAN, J.	
HAWKINS, R.	F A A
HEASTY, R.	COMPUTER SCIENCES CORP
HENNING, H.	LOCKHEED
HENRY, S.	VIRGINIA POLYTECH
HENSLEN, T.	IITRI
HERRING, E.	NASA/GSFC
HIGGINS, L.	LOCKHEED
HILLIARD, J.	NASA/MSFC
HODGE, D.	ABERDEEN PROVING GROUNDS
HOGGAND, J.	F C C
HOGUE, M.	MCCABE & ASSOC
HOLMES, B.	G S C
HOLT, R.	GEORGE MASON UNIVERSITY
HOOT, J.	FORD AEROSPACE
HOUSTON, R.	IITRI
HOWLETT, A.	IITRI
HUGHES, A.	GENERAL RESEARCH CORP
HUNTER, K.	COMPUTER SCIENCES CORP
HUSETH, S.	
HYBERTSON, D.	LOCKHEED
IDELSON, N.	APL
ISSACS, J.	T R W
JAMIESON, L.	NASA/GSFC
JAWORSKI, A.	FORD AEROSPACE
JELETIC, J.	NASA/GSFC
JENKINS, D.	F A A
JENNINGS, W.	HARRIS CORP
JESSEN, W.	RAYTHEON
JONES, C.	IITRI
JONES, J.	IITRI
JOO, B.	
JORDAN, L.	COMPUTER SCIENCES CORP
KAFURA, D.	COMPUTER SCIENCES CORP
KARDATZKE, O.	NASA/GSFC
KATZ, B.	UNIVERSITY OF MARYLAND
KATZ, S.	D I A
KAUSCH, C.	NASA/GSFC
KELLY, A.	NASA/GSFC
KESTER, R.	GENERAL ELECTRIC
KIRK, D.	NASA/GSFC
KLITSCH, G.	COMPUTER SCIENCES CORP

KNABLEIN, R.	T R W
KNIGHT, J.	UNIVERSITY OF VIRGINIA
KOERNER, K.	COMPUTER SCIENCES CORP
KOLACKI, R.	SPACE & NAVAL SYSTEMS CMD
KOVORIK, V.	HARRIS CORP
KRAMER, L.	PLANNING RESEARCH CORP
KRAMER, P.	PLANNING RESEARCH CORP
KRAMER, N.	PRC/GIS
KUHN, R.	NAT'L BUREAU OF STANDARDS
KURIHARA, T.	DEPT. OF TRANSPORTATION
KYNARD, M.	NASA/MSFC
LABAW, B.	NAVAL RESEARCH LABS
LAMAS, M.	
LAMONTAGNE, G.	G T E
LANGDON, N.	COMPUTER SCIENCES CORP
LEADER, K.	IITRI
LEBAIR, B.	NASA/GSFC
LEBER, R.	GENERAL ELECTRIC
LEWIS, J.	CENSUS BUREAU
LIN, M.	LOCKHEED
LIN, R.	LOCKHEED
LIU, J.	COMPUTER SCIENCES CORP
LO, P.	COMPUTER SCIENCES CORP
LONG, D.	
LORD, Y.	WESTINGHOUSE
LOVE, B.	NASA/GSFC
LOVE, D.	LOCKHEED
LUCZAK, R.	COMPUTER SCIENCES CORP
LUPTON, G.	DIGITAL EQUIPMENT CORP
LYTTON, V.	U S D A
MACK, M.	NASA/GSFC
MADDOX, B.	GENERAL DYNAMICS
MAURI, J.	GRUMMAN
MAYBURY, F.	T R W
MCCALL, J.	SCIENCE APPLICATIONS
MCCARRON, S.	NASA/GSFC
MCCLIMENS, M.	MITRE CORP
MCCOEY, P.	
MCCOY, W.	NAVAL SURFACE WEAPONS CTR
MCGARRY, F.	NASA/GSFC
MCGARRY, M.	IITRI
MCGARRY, P.	GENERAL ELECTRIC
MCGOVERN, D.	F A A
MCKEEN, C.	MARTIN MARIETTA
MCKENNA, J.	N S A
MCLEOD, J.	JET PROPULSION LAB
MCPHEE, J.	DEPT. OF COMMERCE
MERWARTH, P.	NASA/GSFC
MIDDLETON, M.	F C C
MILLER, W.	COMPUTER SCIENCES CORP
MILLICAN, J.	SOCIAL SECURITY ADMIN
MILLNER, D.	IITRI

MIYA, E.	NASA/ARC
MOORE, J.	VITRO
MOOREHEAD, J.	T R W
MOOREHEAD, D.	INTERMETRICS
MOWDAY, B.	GENERAL DYNAMICS
MUCKEL, J.	COMPUTER SCIENCES CORP
MURPHY, B.	COMPUTER TECHNOLOGY ASSOC
MURPHY, R.	NASA/GSFC
MYERS, P.	COMPUTER SCIENCES CORP
NELSON, R.	NASA/GSFC
NICHOLAS, D.	JET PROPULSION LAB
NOONAN, R.	WILLIAM & MARY
NORCIO, A.	NAVAL RESEARCH LABS
NUMKIN, L.	LOCKHEED
O'NEIL, L.	AT&T BELL LABS
OHLMACHER, J.	SOCIAL SECURITY ADMIN
OLSON, L.	F C C
OSBOURNE, W.	NAT'L BUREAU OF STANDARDS
OVERDECK, B.	INTERNAL REVENUE SERVICE
OWINGS, J.	NASA/GSFC
PACKARD, C.	NASA/GSFC
PAGE, G.	COMPUTER SCIENCES CORP
PANLILIO-YAP, N.	UNIVERSITY OF MARYLAND
PARKER, K.	G T E
PARKER, D.	NASA/GSFC
PASSALACQUA, T.	CENSUS BUREAU
PAVNICA, F.	
PAYTON, T.	SOFTWARE DEVELOPMENT CORP
PENNEY, L.	PENNY ASSOCIATES
PETER, M.	GENERAL SERVICES ADMIN
PETERS, K.	NASA/GSFC
PETERSEN, B.	AUTOMETRIC
POPE, J.	
PRESTON, D.	IITRI
PURINTON, S.	NASA/MSFC
PUTNEY, B.	NASA/GSFC
QUANN, E.	COMPUTER SCIENCES CORP
QUANN, J.	NASA/GSFC
RAMSEY, J.	UNIVERSITY OF MARYLAND
RAMSEY, C.	UNIVERSITY OF MARYLAND
RATTE, G.	U S D A
REEDY, A.	PLANNING RESEARCH CORP
REIFER, D.	REIFER CONSULTANTS
RICE, B.	U.S. NAVY
RICHARDSON, C.	PLANNING RESEARCH CORP
RINN, P.	F C C
RIZZARDI, G.	PENTAGON
ROBBINS, D.	N S A
ROBERTS, R.	U S D A
ROBERTS, R.	PLANNING RESEARCH CORP
ROBERTS, M.	FORD AEROSPACE
ROHLEDER, M.	IBM CORP

ROHR, J.	JET PROPULSION LAB
ROSS, K.	INTERMETRICS
ROSSIN, R.	GENERAL ELECTRIC
ROY, D.	CENTURY COMPUTING
SAGAT, D.	IBM CORP
SAMII, M.	COMPUTER SCIENCES CORP
SANBORN, J.	NASA/GSFC
SAVOLAINE, C.	AT&T BELL LABS
SCALISE, G.	DEPT. OF TRANSPORTATION
SCHRADE, T.	
SCHULTZ, A.	GEORGE MASON UNIVERSITY
SCHWARTZ, M.	IITRI
SEIDEWITZ, E.	NASA/GSFC
SENN, E.	NASA/LARC
SERAFIN, P.	EG & G
SHANKLIN, R.	INTERNAL REVENUE SERVICE
SHEN, V.	M C C
SHEPPARD, S.	COMPUTER TECHNOLOGY ASSOC
SHMICONICA, Y.	
SIMON, R.	UNIVERSITY OF MARYLAND
SIMOS, M.	
SMITH, D.	FORD AEROSPACE
SMITH, G.	NASA/GSFC
SMITH, K.	NASA/LARC
SMITH, N.	NASA/GSFC
SNYDER, G.	COMPUTER SCIENCES CORP
SNYDER, P.	MITRE CORP
SOLDERITSCH, J.	BURROUGHS
SOLOMON, D.	COMPUTER SCIENCES CORP
SONTI, V.	BENDIX
SORKOWITZ, A.	DEPT. OF HUD
SPEIZER, H.	CENSUS BUREAU
SPENCE, C.	COMPUTER SCIENCES CORP
SPIEGEL, D.	NASA/GSFC
STAMENT, A.	PLANNING RESEARCH CORP
STANLEY, P.	GEORGE MASON UNIVERSITY
STARK, M.	NASA/GSFC
STEINBACHER, J.	JET PROPULSION LAB
STEVENS, B.	NASA/MSFC
STEVENS, W.	MITRE CORP
STEWART, L.	COMPUTER TECHNOLOY ASSOC
STONE, B.	PLANNING RESEARCH CORP
STRAETER, T.	GENERAL DYNAMICS
SUDDITH, S.	G S C
SUKRI, J.	UNIVERSITY OF MARYLAND
SULLIVAN, S.	
SZULEWSKI, P.	C.S. DRAPER LABS
TALCOTT, G.	
TAORMINA, L.	NASA/MSFC
TARDIF, M.	NASA/GSFC
TASKY, D.	CENSUS BUREAU
TAUSWORTHE, R.	JET PROPULSION LAB

THAYER, R.	LOCKHEED
THIBODEAU, R.	GENERAL RESEARCH CORP
THOMPSON, J.	FORD AEROSPACE
THURMAN, J.	U S D A
TICE, G.	TEKTRONIX
TIGHE, M.	INTERMETRICS
TILLOTSON, K.	CINCOM
TOLSTIKHIN, N.	CENSUS BUREAU
TOM, K.	ARINC INC
TRAHAN, D.	IBM CORP
TSONOS, D.	
TURNER, A.	
USHER, G.	U S D A
VALETT, J.	NASA/GSFC
VOIGT, S.	NASA/LARC
WALIGORA, S.	COMPUTER SCIENCES CORP
WALLACE, S.	NAT'L BUREAU OF STANDARDS
WANG, C.	NASA/MSFC
WANG, P.	INTERNAL REVENUE SERVICE
WARTIK, S.	UNIVERSITY OF VIRGINIA
WATKINS, J.	NASA/MSFC
WEICK, D.	BURROUGHS
WEINREB, M.	SOHAR
WEISER, M.	UNIVERSITY OF MARYLAND
WEISS, P.	COMSAT
WEISS, S.	G T E
WELDON, L.	UNIVERSITY OF MARYLAND
WENDEL, C.	NASA/GSFC
WENNESON, G.	INFORMATICS
WERKING, R.	NASA/GSFC
WHITE, C.	NASA/MSFC
WILKE, R.	
WILLIAMS, K.	PLANNING RESEARCH CORP
WILSON, J.	IBM CORP
WOLT, N.	GRUMMAN
WONG, W.	NAT'L BUREAU OF STANDARDS
WOOD, D.	SOFTECH
WOOD, R.	COMPUTER SCIENCES CORP
WOODRUFF, J	NASA/GSFC
WRIGHT, F.	COMPUTER SCIENCES CORP
WU, H.	
WU, S.	IITRI
WU, Y.	IITRI
YEN, W.	LOCKHEED
YODIS, E.	PLANNING RESEARCH CORP
YOUNG, L.	WESTINGHOUSE
YOUNGERS, M.	PLANNING RESEARCH CORP
YOUNT, L.	SPERRY CORP
ZAVELER, S.	
ZLATIN, S.	MCCABE & ASSOC
ZYGIELBAUM, A.	JET PROPULSION LAB

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-77-001, The Software Engineering Laboratory, V. R. Basili, M. V. Zelkowitz, F. E. McGarry, et al., May 1977

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-003, Structured FORTRAN Preprocessor (SFORT), B. Chu and D. S. Wilson, September 1977

SEL-77-004, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-001, FORTRAN Static Source Code Analyzer (SAP) Design and Module Descriptions, E. M. O'Neill, S. R. Waligora, and C. E. Goorevich, February 1978

SEL-78-003, Evaluation of Draper NAVPAK Software Design, K. Tasaki and F. E. McGarry, June 1978

SEL-78-004, Structured FORTRAN Preprocessor (SFORT) PDP-11/70 User's Guide, D. S. Wilson and B. Chu, September 1978

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-78-202, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 2), W. J. Decker and W. A. Taylor, April 1985

SEL-79-001, SIMPL-D Data Base Reference Manual, M. V. Zelkowitz, July 1979

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-001, Functional Requirements/Specifications for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, A. L. Green, and C. E. Goorevich, February 1980

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

SEL-80-104, Configuration Analysis Tool (CAT) System Description and User's Guide (Revision 1), W. Decker and W. Taylor, December 1982

SEL-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker and F. E. McGarry, March 1981

SEL-81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

SEL-81-012, The Rayleigh Curve As a Model for Effort Distribution Over the Life of Medium Scale Software Systems, G. O. Picasso, December 1981

SEL-81-013, Proceedings From the Sixth Annual Software Engineering Workshop, December 1981

SEL-81-014, Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL), A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-102, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide Revision 1, P. Lo and D. Wyckoff, July 1983

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-106, Software Engineering Laboratory (SEL) Document Library (DOCLIB) System Description and User's Guide, W. Taylor and W. J. Decker, May 1985

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-203, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, P. Lo, June 1984

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-82-001, Evaluation of Management Measures of Software Development, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-003, Software Engineering Laboratory (SEL) Data Base Reporting Software User's Guide and System Description, P. Lo, September 1982

SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

SEL-82-007, Proceedings From the Seventh Annual Software Engineering Workshop, December 1982

SEL-82-008, Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1), W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-306, Annotated Bibliography of Software Engineering Laboratory Literature, D. N. Card, Q. L. Jordan, and F. E. McGarry, November 1985

SEL-83-001, An Approach to Software Cost Estimation, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983

SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-83-104, Software Engineering Laboratory (SEL) Data Base Retrieval System (DARES) User's Guide, T. A. Babst, W. J. Decker, P. Lo, and W. Miller, August 1984

SEL-83-105, Software Engineering Laboratory (SEL) Data Base Retrieval System (DARES) System Description, P. Lo, W. J. Decker, and W. Miller, August 1984

SEL-84-001, Manager's Handbook for Software Development, W. W. Agresti, F. E. McGarry, D. N. Card, et al., April 1984

SEL-84-002, Configuration Management and Control: Policies and Procedures, Q. L. Jordan and E. Edwards, December 1984

SEL-84-003, Investigation of Specification Measures for the Software Engineering Laboratory (SEL), W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, Proceedings From the Ninth Annual Software Engineering Workshop, November 1984

SEL-85-001, A Comparison of Software Verification Techniques, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team, R. Murphy and M. Stark, October 1985

SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985

SEL-85-004, Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics, R. W. Selby, Jr., May 1985

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

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

SEL-RELATED LITERATURE

Agresti, W. W., Definition of Specification Measures for the Software Engineering Laboratory, Computer Sciences Corporation, CSC/TM-84/6085, June 1984

[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

[3]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: Computer Societies Press, 1981

Basili, V. R., "SEL Relationships for Programming Measurement and Estimation," University of Maryland, Technical Memorandum, October 1979

[3]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: Computer Societies Press, 1980 (also designated SEL-80-008)

[1]Basili, V. R., "Quantitative Evaluation of Software Methodology," Proceedings of the First Pan-Pacific Computer Conference, September 1985

[3]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

[3]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

[1]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

[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

[3]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

[1]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P--A Prototype Expert System for Software Engineering Management," Proceedings of the IEEE/MITRE Expert Systems in Government Symposium, October 1985

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity and Cost, October 1979

[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

[1]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environments's Characteristic Software Metric Set," Proceedings of the Eighth International Conference on Software Engineering, August 1985

Basili, V. R., and R. W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, University of Maryland Technical Report, TR-1501, May 1985

[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

[1]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984

[3]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

[3]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

[3]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: Computer Societies Press, 1978

[1]Card, D. N., "A Software Technology Evaluation Program," Annais do XVIII Congresso Nacional de Informatica, October 1985

[1]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," Proceedings of the Eighth International Conference on Software Engineering, August 1985

[3]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: Computer Societies Press, 1981

[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: Computer Societies Press, 1983

Higher Order Software, Inc., TR-9, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977 (also designated SEL-77-005)

[1]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

[1]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

[1]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," Proceedings of the Eighth International Conference on Software Engineering, August 1985

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

[1]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

[3]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: Computer Societies 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

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

[1]This article also appears in SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985.

[2]This article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

[3]This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.