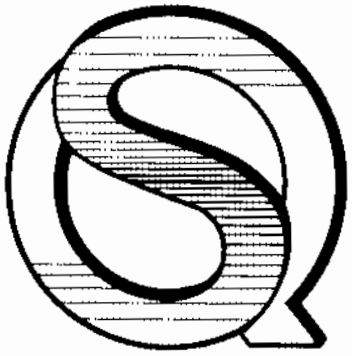
1984 and 1983 Proceedings
Northwest Software Quality Conference

White Pages (1984)
"Designing for Software Quality"
Portland, Oregon
September 24, 1984

Blue Pages (1983)
"Measuring Software Quality"
Corvallis, Oregon
September 26, 1983

2nd Annual
Northwest Software Quality Conference



DESIGNING for SOFTWARE QUALITY

September 24, 1984
Portland, Oregon

Last year, 231 people attended the first Northwest Software Quality Conference where the theme was "Measuring Software Quality." This year, software professionals from numerous companies and universities in the Northwest have again assembled to share information and ideas in the quest for high quality software.

The theme of this year's conference is DESIGNING FOR SOFTWARE QUALITY. Mr. Ronald LaFleur, nationally known project management consultant, speaks to this subject for the keynote address.

I would like to extend a cordial welcome to last year's attendees and newcomers alike. We have an outstanding slate of speakers and I believe that this year's conference will be even more successful than last year's.

Chuck Martiny,
Conference Chairman

1984 Northwest Software Quality Conference

Table of Contents

Introduction and Chairman's Message

Charles F. Martiny

By nature, software engineers have a tremendous amount of pride. We get great satisfaction out of knowing that we can significantly improve the lives of those who use our products. We all want to have the highest quality possible, but normally do not have all the answers necessary to attain that goal.

Software quality is something we have all discussed and often tried to define. Although we may not be able to agree on a single definition, we all agree that it is a vital part of our careers. Since the cost of poor quality software is completely prohibitive, our very careers and the success of our companies are directly tied to the quality of our software. Our productivity is so interwoven with the quality of our software that they are inseparable. The quality must be designed-in and maintained at every step by every person associated with each product. The task of trying to insert the quality at other phases, such as evaluation and testing, for example, is nearly impossible.

The concept of productivity and quality is so simple. We know that high quality software is far less expensive and requires much less effort over the life of the product than lesser-quality products. The toughest part is in knowing how to design that quality into the product right from the very start. That is the issue we are addressing at this conference.

A number of successful people are here to discuss ideas and techniques for designing quality into software products. Now it is our responsibility to identify the ideas that fit our specific needs and implement them in our own work environment.

Organizing Committee
1984 Software Quality Conference

Chairman

Mr. Chuck Martiny, Software Center Manager
Tektronix MS 50-487
P.O. Box 500
Beaverton, OR 97077

Co-Chairmen

Mr. Steve Shellans, Manager Software Information Services
Tektronix MS 50-487
P.O. Box 500
Beaverton, OR 97077

Mr. Ron Swingen, Senior Project Manager
Intel MS HF2-2-251
5200 N.E. Elam Young Parkway
Hillsboro, OR 97124

Committee

Mr. Ron Blair
Tektronix MS 92-826
P.O. Box 500
Beaverton, OR 97077

Mr. Rich Martin, Manager, Xenix Development/Project Control
Intel
5200 N.E. Elam Young Parkway
Hillsboro, OR 97124

Mr. Glenn Meldrum, Senior Software Engineer
John Fluke Mfg. Co.
P.O. Box C 9090
Everett, WA 98206

Mr. LeRoy Nollette, Program Manager, Corporate Quality Assurance
Tektronix MS 78-541
P.O. Box 500
Beaverton, OR 97077

Mr. Ken Oar, Software Quality Manager, Portable Computer Division
Hewlett Packard
1010 Circle Blvd.
Corvallis, OR 97330

Ms. Sherry Sisson, Member of Technical Staff
Hewlett Packard
1010 Circle Blvd.
Corvallis, OR 97330

Mr. George Tice, Senior Software Engineer
Tektronix MS 92-606
P.O. Box 500
Beaverton, OR 97077

Speakers and Panelists

Mr. Thomas N. Anderson
Teltone Corporation
P.O. Box 657
Kirkland, WA 98033

Mr. David Cassing
Tektronix MS 63-205
P.O. Box 1000
Wilsonville, OR 97070

Ms. Karen R. Cunningham
Tektronix MS 61-183
P.O. Box 1000
Wilsonville, OR 97070

Mr. William S. Edmark
Intel Corp.
5200 N.E. Elam Young Parkway
Hillsboro, OR 97124

Dr. Thomas J. Edwards
Teltone Corporation
P.O. Box 657
Kirkland, WA 98033

Mr. Pete Fratus
Hewlett-Packard
19447 Pruneridge Ave.
Cupertino, CA 95014

Mr. Michael Garvey
First Interstate Services Co.
P.O. Box 230
Portland, OR 97207

Ms. Monika Hunscher
Floating Point Systems
P.O. Box 23489
Portland, OR 97223

Mr. Martin Kennedy
Tektronix MS 92-525
P.O. Box 4600
Beaverton, OR 97075

Mr. Ronald LaFleur
Project Mgmt. Assistance Co.
18 Lincoln Ave.
Scituate, MA 02066

Ms. Cary Lamoureux
Tektronix MS 92-525
P.O. Box 4600
Beaverton, OR 97075

Mr. Michael E. Meyer
Tektronix MS 50-560
P.O. Box 500
Beaverton, OR 97077

Mr. Leonard E. Miller
Teltone Corporation
10801 120th Ave. N.E.
Portland, OR 98033

Dr. Michael C. Mulder
University of Portland
5000 N. Willamette Blvd.
Portland, OR 97203

Mr. Allen Sampson
Tektronix MS 92-789
P.O. Box 4600
Beaverton, OR 97075

Ms. Janet Sheperdigian
Intel Corp.
5200 N.E. Elam Young Parkway
Hillsboro, OR 97124

Mr. Russell R. Sprunger
Graphic Software Systems
25117 S.W. Parkway
Wilsonville, OR 97070

Mr. George Tice
Tektronix MS 92-525
P.O. Box 4600
Beaverton, OR 97075

Ms. Susan Wechsler
Hewlett-Packard
1000 N.E. Circle Blvd.
Corvallis, OR 97330

Ms. Donna J. Wengrowski
Mentor Graphics Corp.
8500 S.W. Creekside Place
Beaverton, OR 97005-7191

Mr. Don Zocchi
Tektronix MS Y6-546
P.O. Box 500
Beaverton, OR 97077

Keynote

Software Quality Begins with Project Design

Ronald LaFleur
President
Project Management Assistance Company, Inc.

Most often, when the relationship between the design process and software quality is cited, it is **product** design that comes to mind. However, software quality actually takes root — or fails to take root — during the **project** design process.

Successful project design requires a clear understanding of the goals, followed by clearly stated project objectives, and a knowledge of the distinct stages through which a project must pass to reach those objectives, and strict task accountability within each stage. What is **not** well understood today by many project managers is that each of the stages must be managed differently if the goals are to be reached in a timely and cost-effective manner, including the goal of software quality.

Biography

Mr. LaFleur is a professional project manager with 25 years of experience at top levels of industry and government. He is a former director of project management for Honeywell Mini-computers, where he was responsible for all phases of project management. He has served as business and project manager of large defense contracts for Raytheon and Hazeltine.

Mr. LaFleur is president of Project Management Assistance Company which he founded in 1976. PMAC provides corporations with project management services consisting of public and private seminars, and direct on-site consulting geared toward various levels of personnel.

1 Quality in the Design Phase —First Session

Speakers and Titles:

Mr. Leonard E. Miller; Teltone
"Software Quality Assurance in Design"

Ms. Janet Sheperdigian; Intel
"Involvement of Software Testing Organizations in Better Software Design"

Mr. Michael E. Meyer; Tektronix
"The Software Quality Mindset"

SOFTWARE QUALITY ASSURANCE IN DESIGN

by

Leonard E. Miller
Teltone Corporation
10801 120th Avenue N.E.
Kirkland, WA 98033

ABSTRACT

This paper addresses the roll of the Software Quality Assurance organization in software design. What constitutes a quality design, efficiency? speed? creativity? maintainability? testability? flexibility? Or how about being on time and within budget? There are cases where speed at execution, size limitations and other constraints are critical to a software design. But in most of my experience, what makes a quality software design is the ability to complete it on schedule and within budget, and still have a design that is easily altered to meet new applications.

Quality software starts with defining what is to be built -no moving targets- and how it is to be built -management, design, code and test standards. This establishes the foundation necessary for a quality design. To avoid the development of standards or to postpone the development of requirements is an expensive alternative, which normally results in confusion, incorrect schedules and an inordinate amount of rework and testing.

How does the Software Quality Assurance organization fit into the picture? Software Quality Assurance provide the consistency necessary in the software development process. They ensure that the standards, requirements and procedures become "living" documents that are either followed or changed, that a design or code review is consistent no matter who conducts it or who participates, and that action items are documented, tracked and eventually closed. They also ensure that adequate configuration control exists, that testing is adequate, and that all software is properly documented. In short Software Quality Assurance can provide the program manager the assurance that the tools upon which he must depend are actually implemented and implemented consistently.

SOFTWARE QUALITY ASSURANCE IN DESIGN

INTRODUCTION

What is an organization like Quality Assurance doing in design? In fact why are we interested in design? Shouldn't we be more interested in system testing? After all, the product is really a hardware system which just happens to contain code. Does this sound familiar? How about, "We do not need a software quality assurance program because____."? Endings are many and varied, including: it is only a small program, it is a simple program, we are only slightly modifying an existing program, or worst of all we do not have time or budget. No one ever has enough time or money to do it right the first time, but they sure can find the time and budget to fix it and fix it and fix it and...

SOFTWARE DEVELOPMENT

Sending a group of software engineers into a corner for six months is a risky way to develop software. Software development is not a black box process. Nor can the management of this development process be ignored. Software systems are labor saving, but they are labor intensive to develop. Expensive, highly skilled labor is required up front, and that creates a high risk, critical situation. A situation that can be effectively managed when managed in detail. But it requires a knowledge of the available tools, dividing responsibilities among independent organizations and a knowledge of how to manage all available resources. Short cuts do not pay.

ORGANIZATIONAL INDEPENDENCE and BASELINES

The establishment of independent organizations to accomplish the various tasks can not be over emphasized. If the customer establishes requirements, you are assured of an independent review by your design organization. The formal interaction will result in clarifications, and agreements before any formal commitment is made. This is no less important than when the requirements come from your own marketing organization. The formal interface between marketing and design organizations is required to define exactly what the product is to be. This work must be done before detailed work begins. The interaction will result in usable requirements. A similar interface will eventually establish a coherent functional specification. The process should involve and encourage the involvement of all organizations. Test, Quality Assurance, Configuration Management, Program Planning, Customer or Marketing, System Engineering, Hardware Engineering, Software Engineering should all participate. Each has a distinct viewpoint from which to evaluate the proposed program. It is the focusing of these viewpoints that solidifies the concept of the product.

The completion of each phase should be by formal review, each of which establishes a baseline. Baselining the concept at each stage of development allows the concept to be challenged, and possibly modified or discarded, as a result of the challenge. A baseline also provides the data necessary for detailed program planning. Failure to baseline provides for a product which everyone understands, but no two understandings are the same. Failure to provide baselines at the proper level also blurs organizational interfaces. For instance, a system level specification baseline may be sufficient for an all hardware or all software system, however, it is not adequate if hardware/software interfaces exist.

Each phase of software development normally requires the involvement of more people than the last. Therefore cost effective management will try to minimize the ambiguities and errors transmitted to the next phase. Formal baselines promote clear concise descriptions of the product. Failure to establish baselines deprives management of the information necessary to make good decisions. Products may eventually prove to be unprofitable, and if that is the case, the earlier it can be detected, the earlier the development effort can be canceled.

So far we have addressed two management tools: baselines and organizational independence. Organizational independence is not always clear cut. Many times design, development and test will be a close knit group reporting to the program manager. This can lead to a conflict of interest. To prevent this the Quality Assurance organization should be established by company charter and be independent of the program manager. This provides the company and the project manager with an effective management tool: a tool that can monitor, measure and audit development activities without fear of reprisal, a tool that can be cost effectively applied at the beginning and throughout the software development program. But how do you apply this tool when you do not have a traditional product? Quality Assurance must treat each baseline as a product, a product upon which the next phase is built, but also a product which can be evaluated on its own. Baselines include: requirements baseline, specifications baseline, design baseline, module baseline, test requirements baseline. But how can this product be evaluated? For instance, how good are the requirements? Can they be implemented as stated or are they ambiguous or incomplete? What is the baseline supposed to look like? To answer these questions requires a yardstick, a tool to measure against.

STANDARDS

Even a company new to the software business will acquire skilled personnel, and with them a pool of knowledge: knowledge on software management, requirements, design, test and product assurance, knowledge which can be used both to direct and measure software development. This knowledge must be solidified and applied to all programs. The mechanism that should be used is Corporate Standards. Standards define what is expected during each phase of the development process, assigns responsibility, define baselines and can be used to monitor development activities. Standards will minimize misunderstandings, short cuts and oversights. The following is a list of proposed standards and topics to be considered. The list is a guide and is not intended to be complete.

(1) Management Standards management objectives, planning, organizational interfaces, technical control, progress evaluation, change control, procurement, cost management, software quality assurance and facilities.

(2) Requirement Standards content and scope of requirement phase including documentation, system analysis, trade studies, system requirements review, hardware and software subsystem concept definition, interface requirements, system design review, software specification and software requirements review.

(3) Software Design Standards requirements allocation, requirements traceability, module interfaces, database design, sizing and timing, programming language selection, implementation plan, documentation, basic design review, design language, design language processor, design language standards, detail design documentation, sizing and timing, detailed design review, coding, code documentation, test procedure review, module test, module integration, test readiness review.

(4) Test Standards module test, software integration test, software/hardware integration test, system test, test planning, test scheduling, test requirements, test requirements traceability, test case selection, test control, test personnel, test analysis and report, and test documentation including plans, descriptions, procedures and reports.

(5) Product Assurance Standards software configuration management function, software configuration management interface, software configuration management life cycle, identification of software, computer program library, software problem reporting and change control, patches, software review board, change records, redline

drawing control, production records, engineering release, product assurance audits and documentation.

The existence of standards, however, is not sufficient in and of itself. They must be concise, current, usable and followed. This of course, also, applies to the schedules, plans, documents, drawings, procedures, etc developed under the standards. The Software Quality Assurance organization, as mentioned above, becomes the management tool to monitor and audit software development activities to assure that the standards are current and followed, or when plans require a specific exception, that this exception is properly documented and approved. Software Quality Assurance also monitors the compliance of development activities with plans, procedures, and requirements.

In addition to compliance with standards, plans and procedures, it is necessary that development activities be consistently implemented. As an example, module design and module code reviews are usually carried out in small groups, where usually no one person will attend more than a small percentage of the reviews. These reviews are intentionally kept small and informal to avoid embarrassment to the designer, but they must be consistent and provide assurance that requirements are being met, the design will be successful and standards are being followed. Again, Quality Assurance provides the independent monitoring activity. A simple review sheet can be used to assure that all essential areas are covered. The reviewer can provide the proof that all discrepancies have been corrected by signing the review sheet. Software Quality Assurance personnel would attend selected reviews and audit the review files to assure consistency.

MEASUREMENT OF ATTRIBUTES

We have established the value of Quality Assurance as a management tool with respect to monitoring compliance with existing standards, plans and procedures. This is important to the program's success, but it does not necessarily indicate how good or how bad the development effort is going. Quality Assurance should also measure the product attributes at each phase of the development process. How can this be accomplished? Certainly Qualification testing will measure the final product, but what can be done to measure requirements, specifications or design?

These questions are not trivial. While the methodology for handling hardware is well defined and proven, the methodology for handling software is not. The first thing that must be done is to list and define the attributes which should be measured. Suggested attributes include:

ATTRIBUTE[1]	DEFINITION
Correctness	Extent to which a program satisfies its specifications and fulfills the user's objectives.
Reliability	Extent to which a program can be expected to perform its intended functions with the required precision.
Efficiency	The amount of computing resources and code required by a program to perform a function.
Integrity	Extent to which access to software or data by unauthorized persons can be controlled.
Usability	Effort required to learn, operate, prepare input, and interpret output of a program.
Maintainability	Effort required to locate and fix an error in an operational program.
Testability	Effort required to test a program to insure it performs its intended functions.
Flexibility	Effort required to modify an operational program.
Portability	Effort required to transfer a program from one hardware configuration and/or software system to another.
Reusability	Extent to which a program can be used in other applications, with respect to packaging and scope of the functions that the program performs.
Interoperability	Effort required to couple one system with another.

Obviously not all of these attributes apply to a particular program and the weighting of the other attributes may not be equal. Further it probably is not cost effective to attempt to measure all attributes even if they were applicable, nor is it be necessary to measure them at each baseline. But what needs to be done, and done up front, is

---

SOFTWARE QUALITY ASSURANCE presented by the Education and Training Institute of the American Society for Quality Control instructor C. L. (Skip) Carpenter, Jr.

to analyze the system and its intended uses and make an assessment of which attributes would result in the greatest risk if not monitored. Next, a decision has to be made as to which phase of the development process should be monitored. Again, this is based on the highest return on investment. As an example, if a software program was dedicated to a particular processor and operating environment, then portability would not be important. In turn, if it is a complex program, which is designed to serve several customers, maintainability and flexibility may be key features and costly features if not properly managed. But since correctness seems to be high on everyone's list, let us use it as an example. So where do we start to measure correctness? We start as early as possible, where the largest cost impact can be achieved.

Requirements are the foundation for the whole program, if they are not correct the whole program is suspect. So we measure the correctness of requirements, but how? We must define what is necessary for correctness. Candidates include; complete requirements, consistent requirements and requirements that comply with standards. There are others, but let us assume that we have determined that these will result in the highest return, i.e. cost avoidance to cost to provide. As an example, let us take "complete requirements" to the next step. Again you define the important factors.

1. All functions are unambiguous
2. Source of all data defined
3. All units consistent
4. All external interfaces defined
5. System operational limits defined

We can measure functions for ambiguity, if we define what we will do. For instance, we can select 25 of the functions at random and ask three independent interpretations of the meaning of each. If all opinions agree, then it is an unambiguous function. If they do not, then there is ambiguity. A ratio of unambiguous functions to the total sample is a measurement of correctness as we have defined it. The key to success is that we have defined what we are going to do and how we are going to do it before we actually do the measurement, allowing everyone to participate in the definition of what is important. This provides positive management, rather than an after the fact or passive management.

Obviously, correctness is also important in the functional specifications and many of the same measurements can be made. All may not be applicable, and other attributes may come into play. For instance, traceability to requirements becomes a key measurement of correctness. The

methodology is the same for specifications as it was for requirements and as it will be for design, code and test. Each step should be formally completed and evaluated before continuing to the next. Only in this way can the program be managed with the assurance that everyone is working toward the same goal and that the goal is what the customer wants.

SUMMARY

A software quality assurance program is a cost avoidance program designed to:

1. Encourages participation by all and at the earliest possible phase of development,
2. Provides management tools,
3. Provides measurement methods and
4 Provides accountability.

Software Quality Assurance is the organization that coordinates these activities and assures that the software quality assurance program is a living entity.

SELLING SOFTWARE QUALITY ASSURANCE

But before it can function, a software quality assurance program philosophy must exist. This can only be accomplished by the Software Quality organization marketing its product. Many barriers will appear, most of which will be emotional or political, but they can be overcome if the company is serious about software development and the program is properly sold. Software Quality Assurance is not a burden or a company overhead function to be tolerated, but rather, it is a cost effective management tool. I have seen successful software development efforts which did not include a Software Quality function, but there success was based almost totally on a strong program manager who understood the principles and singlehandedly enforced a defacto software quality assurance program. The tragedy is that it only works if the manager is indeed strong enough to resist external pressures and that all this knowledge is confined to one person and never gets established as company policy.

Henry Ford learned how to manage the mass production of hardware. The mass production of software is easy. Some day we will learn how to build that one software original and build it right the first time and save all that time and energy required to fix it and fix it and...

BIOGRAPHY

Mr. Miller has 20 years of Quality Assurance Engineering experience, including 6 years in software. He is presently developing an integrated software quality

assurance program for the Teltone Corporation. He has directed Quality Assurance research and development projects for The Boeing Company, where he also was involved in the development of software quality assurance programs on the 757/767 airplane projects and on various military projects.

A graduate of the University of Washington, he is a certified Quality Engineer and a member of the American Society for Quality Control.

SOFTWARE QUALITY ASSURANCE
IN DESIGN

ASSUMPTION:

CORPORATION IS SERIOUS ABOUT MANAGING SOFTWARE DEVELOPMENT

WHY?:

- LABOR INTENSIVE
- SKILLED PERSONNEL
- COSTS ARE UP FRONT

THREE ESSENTIALS:

- ORGANIZATIONAL INDEPENDENCE
- BASELINES
- STANDARDS

ORGANIZATIONAL INDEPENDENCE:

- CUSTOMER
- SYSTEM ENGINEERING
- SOFTWARE ENGINEERING
- SOFTWARE ENGINEERING
- SOFTWARE ENGINEERING
- TEST ENGINEERING
- QUALITY ASSURANCE ENGINEERING

- REQUIREMENTS
- SPECIFICATIONS
- DESIGN ARCHITECTURE
- DETAILED DESIGN
- CODE
- VERIFICATION/VALIDATION
- VALIDATION/QUALIFICATION

BASELINE:

- REQUIREMENTS
- SPECIFICATIONS
- DESIGN ARCHITECTURE
- CODE
- QUALIFICATION

- CUSTOMER/SYSTEM ENGINEERING
- CUSTOMER/SYSTEM ENGINEERING
- SYSTEM ENGINEERING/SOFTWARE ENGINEERING
- SOFTWARE ENGINEERING/TEST ENGINEERING
- TEST ENGINEERING/QUALITY ASSURANCE ENGINEERING/ SYSTEM ENGINEERING/CUSTOMER

STANDARDS:

- MANAGEMENT
- REQUIREMENT
- SOFTWARE DESIGN
- TEST
- PRODUCT ASSURANCE

MEASUREMENT OF ATTRIBUTES:

- HOW WELL IS IT GOING?
- ARE THE REQUIREMENTS COMPLETE?
- ARE THE SPECIFICATIONS CORRECT?
- IS THE DESIGN MAINTAINABLE?
- IS THE DESIGN TESTABLE?

WHAT TO MEASURE:

- CORRECTNESS
- RELIABILITY
- EFFICIENCY
- INTEGRITY
- USABILITY
- INTEROPERABILITY
- MAINTAINABILITY
- TESTABILITY
- FLEXIBILITY
- PORTABILITY
- REUSABILITY

THE IMPACT OF NOT SPECIFYING OR MEASURING SOFTWARE QUALITY FACTORS

	Development			Evaluation	Post-Development			Expected Cost Saved Cost to Provide
Life-cycle Phases / Factors	Requirements Analysis	Design	Code & Debug	System Test	Operation	Revision	Transition	
Correctness	O	O	O	●	●	●		High
Reliability	O	O	O	●	●	●		High
Efficiency	O	O	O		●			Low
Integrity	O	O	O		●			Low
Usability	O	O		●		●		Medium
Maintainability		O	O			●	●	High
Testability		O	O	●		●	●	High
Flexibility		O	O			●	●	Medium
Portability		O	O				●	Medium
Reusability		O	O				●	Medium
Interoperability	O	O		●			●	Low

Legend: O where quality factors should be measured ● where impact of poor quality is realized

* EDUCATION AND TRAINING INSTITUTE OF THE AMERICAN SOCIETY FOR QUALITY CONTROL

REQUIREMENTS - CORRECTNESS:

- COMPLETE
- CONSISTENT
- COMPLY WITH STANDARDS

REQUIREMENTS CORRECTNESS - COMPLETE:

- ALL FUNCTIONS ARE UNAMBIGNOUS
- SOURCE OF ALL DATA DEFINED
- ALL UNITS CONSISTENT
- ALL EXTERNAL INTERFACES DEFINED
- SYSTEM OPERATIONAL LIMITS DEFINED

SOFTWARE QUALITY ASSURANCE PROGRAM:

- ENCOURAGES PARTICIPATION
- PROVIDES MANAGEMENT TOOLS
- PROVIDES MEASUREMENT METHODS
- PROVIDES ACCOUNTABILITY

SOFTWARE QUALITY ASSURANCE:

- ASSURE STANDARDS IMPLEMENTATION
- ASSURE CONSISTENT REVIEWS
- MONITOR/AUDIT PER PLANS
- ASSURE AUDIT TRAIL EXISTS
- VERIFY QUALIFICATION RESULTS

SELLING SOFTWARE QUALITY ASSURANCE:

- DEVELOP CORPORATE PHILOSOPHY
- COST EFFECTIVE
- MANAGEMENT TOOLS
- ATTRIBUTE MEASUREMENT

Biography - Leonard E. Miller

Mr. Miller has 20 years of Quality Assurance Engineer experience, including six years in software. He is presently developing an integrated software quality assurance program for the Teltone Corporation. He has directed Quality Assurance research and development projects for the Boeing Company, where he also was involved in the development of software quality assurance programs on the 757/767 airplane projects and on various military projects.

A graduate of the University of Washington, he is a certified Quality Engineer and a member of the American Society for Quality Control.

Involvement of Software Evaluation Organizations in Better Software Design

Janet Sheperdigian

Integrated Systems Operation
Intel Corporation

July 22, 1984

INTRODUCTION

The purpose of many evaluation organizations is to provide independent verification of product specifications and otherwise assure the quality of the product. The ruling concept is that two heads are better than one when it comes to interpreting the specifications of a product. In applying this concept, evaluation organizations are often given a completed software product and asked to assure product quality. Unfortunately, when design problems arise at this stage, few programs are actually redesigned, that is, any fixes implemented are done so in spite of the design or around the design or in the middle of the design, rarely as part of the design. The best a Software Evaluator or Quality Assurance Engineer (hereafter referred to as evaluators) can do is to evaluate the level of quality in a product and report it. To assure the quality of the design, the evaluator should be involved at the design phase.

This early involvement gives the evaluator the chance to validate the design as well as verify it. Validation of a software product is the act of determining whether specifications and code are a reasonable, usable, doable solution to the design requirements. Two heads are equally as useful during this phase as they are in verifying the product. One method of accomplishing this is to include an evaluator on the development team.

THE METHODOLOGY

The basic idea is that while the developer designs and codes, the evaluator develops a test suite and an acceptance test package. By the time the product reaches the Quality Assurance stage, the evaluator will have reviewed and agreed to the high level design, the low level design and the actual code. In addition, a subset of the test suite will have been run as acceptance tests to provide the evaluator with some confidence that the product will meet its specifications.

The evaluator should join the team no later than at the high level design review. Once the high level design has been agreed upon the developer begins the low level design while the evaluator begins writing a test plan. This test plan can consist only of black box test cases at this point. When the low level design has been reviewed design specific test cases can be included. The code review, which should take place as soon as the developer can compile without errors, will provide the last piece of information needed to write test cases.

The timing of a code review, as soon as the code is written, is a good analogy for an underlying idea in this methodology: Problems should be isolated as early as possible throughout the development cycle. In the case of the code review, the usual result is that some code should be changed. If these changes are determined as soon as the code is written, then they can be incorporated into the design with relative ease. On the other hand, if a program has already been tested and debugged, it is difficult to ignore the old adage, "If it ain't broke, don't fix it." This means that such quality factors as

efficiency and maintainability are ignored if the program is already functional. Early reviewing and testing allow more quality factors to be taken into consideration.

The Acceptance Test Package is a method of getting testing done early. The Acceptance Test Package contains a subset of the evaluation test suite. The test cases included should be those which will provide a reasonable level of confidence that the program works. When the program is satisfactorily tested and debugged by the developer, the acceptance tests can be run. It is important that the developer tests and debugs the program independently. Otherwise, the independent verification of the evaluator is lost. Also, acceptance tests are not necessarily debugging tools.

While it is valid for any member of the project team to run these tests, it is recommended that the developer run them, for two reasons. First, it keeps the developer involved with the program while bugs are likely to be found. This results in more cohesive fixes than the disjoint patches hurriedly applied, as so often is the case. Second, the time it takes to report a problem, have it fixed and receive new code is greatly reduced, which ultimately shortens the schedule of the project. Acceptance Test Packages fit smoothly into the development cycle of a project at the point when the product is ready to be transmitted to the evaluator. Instead, the evaluator transmits an Acceptance Test Package to the developer.

Finally, the product is transmitted to the evaluator. Under this plan, rather than just getting started on the product, the evaluator is almost done. The high level design, low level design and code should be re-reviewed to ensure that they are up to date. The remainder of the evaluation test suite (and possibly some of the acceptance tests) should be run for a final overall check. Few, if any, problems should surface during the verification phase.

A CASE STUDY

This methodology was applied to the second release of the Xenix[+] 286 product from Intel Corp. The product team included sixteen development engineers and five evaluation engineers, a ratio of three to one. The product itself consists of the Xenix Kernel, a C compiler, several device drivers and well over two hundred utilities. The compiler and the other utilities were not developed by Intel so their evaluation was nothing more than black box functional testing. About one quarter of the Kernel was developed by Intel, mainly the machine dependent code. The device drivers were entirely designed and developed by Intel and were, therefore, best suited to the methodology.

In general, one developer and one evaluator worked on a driver. The phases outlined in the methodology were followed fairly closely from the high level design to the final verification. Because people tend to struggle against change, it is interesting to look at how and

[+]Xenix is a trademark of Microsoft Corp.

why the methodology was accepted.

The evaluator joined the team in the review of the high level design document. Because the test plan had to be based on this document, evaluators were much more careful to see that it was done correctly. Developers were careful because they understood that their program would be tested against this document. They did the high level design early because evaluation schedules, and therefore project schedules, depended on it.

Code reviews were another story. While most of the drivers had code reviews in time enough to feel comfortable about making changes, it was very difficult to convince a developer to make potentially, buggy code public, even to a few select reviewers. In the future code reviews should be made less imposing, more like passing the first draft of a paper around to a few trusted readers. A second difficulty arose in defining what reviewers were looking for in the code. Lacking a clear direction, we experimented. By the time the last code review was scheduled, a workable guideline existed in the form of a list of ten quality factors and examples of each.

The same type of problem arose concerning the contents of the Acceptance Test Packages (ATPs). After some discussions between developers and evaluators, it was agreed that they should contain no more than one week worth of tests and a minimal number of hardware configurations. (Given eight controller boards, there were a lot of possibilities). The somewhat arbitrary period of one week to run the tests was predicated on all the tests passing. Any time taken to fix problems and rerun the tests extended the period. It was difficult to agree in general on what test cases were reasonable to include because we had never broken a test suite into basic and advanced test cases before. It was decided that the ATP should be mutually agreed upon by the developer and evaluator concerned. There were few problems that were not worked out at this level. If the evaluator could not develop an ATP in time for the project schedules, the developer transmitted without running it. All drivers went through the ATP phase, but none of the Intel developed utilities did, for this reason.

The proof of the pudding, of course, is how well the drivers did in the final verification phase. At this writing three drivers have completed the process successfully and one has had its historic troubles. It is important to note that this process gave evaluators more time in the verification phase to play with the advanced test cases, nonstandard configurations and specialized testing. The obvious problems had been solved and gave evaluation time to look for the less obvious. In the future, it may open the door to spending more time evaluating functional limits and undertanding the capacities of a program.

As discussed above, several documents were undefined and caused a lot of confusion. Before we begin the next project we will create a usable template for high level and low level design documents and acceptance test packages. Practice and new perspectives will make the code reviews more useful. These problems are not specific to this methodology, but because of the dependencies on such documents, the lack of clear definitions was felt more sharply. Finally, with developer testing taking place as each developer saw fit, some ATPs acted as verification tests and others became preliminary debugging tools.

CONCLUSION

The overall effect of this plan produced a tighter project team with fewer surprises at the end of the road. Working together, development had the benefit of the "second head" earlier while evaluation was given a better chance of becoming the experts they had to be. All this added up to a better quality product.

The Acceptance Test Packages eliminated arguments about whether or not a program is ready for evaluation. They provided well-defined criteria for acceptance which development was given the opportunity to approve. If nothing else in this methodology can be used, the ATPs should be considered. If an evaluation group already exists in the project environment, it is fairly simple to integrate this plan. The time spent on writing ATPs is made up at the end of the cycle.

While the project in the case study is not yet complete, the benefits are already noticeable and there is no doubt that we will continue in this direction with future projects.

Quality is designed in

Quality Assurance = Reliability Assurance

Put Evaluation on the Design team!

Parallel Functions

- Developer designs and codes
- Evaluator reviews and writes tests

Evaluator responsible for:

- High level design review
- Low level design review
- Code review

Get testing done early

When it compiles without errors!

Acceptance Test Package

- Subset of evaluation test suite
- Run by the developer

intel

A CASE STUDY

12 Device drivers

- 9 Developers; 3 Evaluators
- Actually, 6:1 5:1 1:1
- 10 drivers followed methodology

Each step drives the previous one

Evaluator needs high level design

Developer needs ATP

Problems? Well....

- Code Reviews
- Definitions

The proof of the pudding:

- Will know by Sept. 24

CONCLUSION

- Closer Team
- Fewer Surprises
- Quality Drivers

Janet Sheperdigian is the Project Leader for Systems Software Evaluation Engineering at Intel Corp. She has worked in the group for two years evaluating operating system software, including organizing and carrying out the evaluations of two major operating system releases. She graduated from the University of Michigan with a B.S. degree in Computer Science.

The Software Quality Mindset

Michael E. Meyer

Manager, CAX Data Management
CAX Center
Computer Science Center
Tektronix, Inc.
Beaverton, OR 97077

ABSTRACT

The last few years have seen a large increase in the availability and variety of software quality control tools for use by Software Engineers. These tools endeavor to ensure the quality of the software product at many points in the development process and are based upon a myriad of software development methodologies. They run on various operating systems using mainframes, minicomputers and personal computers. In addition to these tools, there has also been a very significant increase in the number and types of discrete methods used for software control and quality.

While the development of these tools and methods has certainly been positive, there has been such a rapid deployment of them in the industry that many Software Engineering organizations have been overwhelmed. More significantly, organizations have been slow to adopt an environment in which tool use is controlled and encouraged. As a result, these tools and methods have, in many cases, only served to add confusion to the already perplexing software quality question.

Software Engineering organizations intent upon producing quality software must adopt a philosophy and accomplish some very basic tasks before they can effectively use any tools or methods. The key task must be the adoption of a software development methodology (life cycle). In addition, the organization's members must be taught the value of a controlled development environment. Only if an organization adopts a holistic position on software quality, will software quality control tools and discrete methods become valuable.

BACKGROUND

Ultimate software quality has been addressed for years and will continue to be sought long after the pages of these proceedings have turned to dust. Each of us comes to this conference with aspirations to learn from the software sages just how to develop the ultimate in quality software. Indeed, some of us secretly hope that when we leave today, we shall have the key that will solve our software quality problems.

Unfortunately, the ultimate tools for developing quality software simply do not exist. Most of us are well aware of that fact. Others continue to plod on, seeking to find the methodology or the tool that will unlock the secrets of software quality. It is the

contention of this author that the solution does not lie in the new and improved comprehensive XYZ method or the elegant new WHIZBANG software tool. The solution exists in the philosophy of the software engineering organization endeavoring to develop quality software in a constantly evolving environment.

CURRENT ENVIRONMENTS

It has been said that the only constant in our industry is change. With the daily introduction of faster hardware and the increased sophistication of software, it is clear that we work in an industry which measures change in terms of months and days. Any software we develop today will probably be obsolete before it is coded, much less tested and installed.

At the same time, the software that we install at the end of any project will probably not reflect the original concept or design. The significant time lag from software conception to completion cannot hope to keep up with the changing mood of the marketplace or the user community.

WHAT IS QUALITY SOFTWARE?

Software quality has been defined [COO 79] as, "The composite of all attributes which describe the degree of excellence of the computer software". While this is an elegant definition, it does not address what software quality means to a manager, a software engineer and a user in an organization. It is not clear that a concise definition exists.

If a definition does exist, it would probably be: that software which meets the target user's requirements, is easily maintainable, has been properly controlled during its development and has been developed on time and within budget.

Manager's Perspective

From a typical second or third level manager's perspective, quality software means a system that is built on schedule, within budget and leaves the user content. This level manager is rarely concerned with the elegance of the code or the fact that the software engineers used X-YZ routines employing the latest in coding techniques. He or she is also probably not concerned with the fact that the software engineers did or did not use good commenting techniques.

The first level manager, although concerned with many of the same issues as his or her superiors, is more concerned with maintainability and meeting schedules. He or she will probably be maintaining the system. As a result, quality software means software that has been well documented, commented well and has used a good method for change control.

Time is an enemy of all managers. Users want the system last month, and the temptation to cut corners and reduce the "bureaucracy" is extremely tempting. Unfortunately, reducing the bureacracy means reducing control of the project.

Quality to the manager then, is the ability to deliver what the user wants, on schedule, within budget, with a minimum of bureacracy but under control.

Software Engineer's Perspective

The typical software engineer is usually interested in elegance and state of the art. The fact that the code will run 10% faster using one routine over another is of significant interest. Whether the project is on time or within cost is usually not of interest, although the software engineer is concerned with meeting his or her own schedules. Documentation, the usual anathema to a software engineer, is grudgingly produced. Commenting within programs however, is not usually a problem, especially when the software engineer perceives it as an alternative to other documentation.

Quality to the engineer then, is the ability to code programs creatively and be able to provide a minimum set of documentation with a minimum management bureaucracy. Interestingly, it is usually the case that a software engineer enjoys maintaining good code, but perceives that he rarely has the time to create it.

User's Perspective

The user is concerned with the bottom line. Does the software work as he or she envisioned it to work? The fact that it does or does not meet that criterion is a measure of quality for the user. Documentation is a big factor as well. If the software provides significant assistance to the user but is poorly documented, it is no help at all.

Quality to the user is simply the fact the system satisfies his needs.

HOW TO MEET ALL PERSPECTIVES

The challenge for years, in our industry, has been to meet these varied perspectives and provide quality software that also can evolve and change with the changing requirements of the marketplace or user environment. There is no easy answer, and there are no methodologies or software tools that will provide the answer.

The application of some basic policies and techniques can provide an environment in which quality software will be produced. None of these policies are revolutionary, and none are products of the latest extensive research. They are common sense approaches and many are being used by many of you today. The significance is that they are probably not all being used.

This author feels that the basic steps that must be taken to provide an environment in which quality software can be produced are:

- APPLY A BUSINESS APPROACH
- ADOPT A SOFTWARE LIFE CYCLE METHODOLOGY
- ORGANIZE CONFIGURATION MANAGEMENT FUNCTION
- ESTABLISH BASELINES
- ESTABLISH STANDARDS

Business Approach

Most of us, when we want to buy a new automobile, do not go to the nearest dealer and pay sticker price without discussion or without some determination as to the utility of the vehicle, or the cost of operation, etc. How many of our organizations determine to write software without assessing the cost/benefits of the software to the organization? This is an especially easy trap to fall into when the software is to be used internally.

We cannot produce quality software if we do not determine that the benefit of the software will outweigh its cost. Many times, because of our experience, we have intuitive feelings about the cost and benefits of the software we want to produce. Often, however, when we have completed a cost/benefit analysis, we are surprised that the software may or may not provide the significant payback we expected. More often than not, other alternatives may provide a more significant return for our limited resource dollars.

The most important realization is that we all do have limited resources. Software engineers can only produce so much. Contrary to popular belief, they do not enjoy working until midnight hacking out a program. Also, morale within a software engineering organization can deteriorate rapidly when software is developed only to find a limited use.

Although it has been often said [NIE 82] that a quality requirements and feasibility study will provide the basis for quality software, these steps are usually reserved for "large" projects [MAR 78]. We must apply these steps to all projects and in addition apply cost/benefit analyses to insure that the cost of the system will be recovered by the benefits that it accrues. This approach can reduce the overall cost of the development of the software, [DEP 84] [HEN80] and indeed save the project from failure before it starts.

STEP 1: ADOPT A BUSINESS APPROACH TO THE DEVELOPMENT OF ALL SOFTWARE.

Software Life Cycle Methodology

We all recognize that we should use some kind of a software life cycle methodology to develop software [PUT 80]. None of us would hire a group of carpenters, buy some land, buy some material and ask them to start building a house. All of us would plan the construction carefully, determine the milestones (especially when payments are based on level of completion), and stage the construction. The surprising fact is many organizations do not follow this software life cycle approach to software development. In other cases, organizations only give the approach lip service in order to satisfy upper management. In only 10% of the cases, in the experience of the author, have organizations approached software developent using a software life cycle methodology which was tightly controlled.

There are many software life cycle methodologies being promoted in the industry today, some of them even providing automated tools to assist the software engineer. We can probably agree that there is no end-all do-all software life cycle methodology.

In the opinion of the author, all of these software life cycle methodologies are complicated, overly complex and intimidating. The job of the software engineer is to make the complex simple, not the simple complex. We cannot hope to instill a mindset promoting controlled quality software development by throwing upon the software engineer a myriad of forms, manuals, and tools. The fact that these may be automated will not reduce the resistance that is inherent when promoting any new system.

Another aspect of large complex software life cycle methodologies is the cost of administering the system. It certainly does not pay to have a system to administer which could swallow the resources which could be more effectively used to develop the system you are trying to control.

All software engineering organizations should adopt a simple software life cycle methodology that provides easy to remember steps for the design, engineering, testing and installation of software. If the software life cycle can be assisted with automated tools fine, but it is not necessary. The fact that the engineer is following a staged development technique with review points will go a long way toward development of quality software.

STEP 2: ADOPT A SIMPLE SOFTWARE LIFE CYCLE METHODOLOGY.

Configuration Management Function

Many organizations have adopted a software life cycle methodology [BER 79]. They have printed manuals and gone to great lengths to follow the steps in the process they have adopted [BER 80]. Unfortunately, in some cases these same organizations have not provided the crucial function needed to manage the methodology. In addition, many organizations have inserted this function within the software engineering group developing the software. This is rather like allowing the fox to guard the chickencoop. It does not provide the checks and balances needed to ensure software quality.

In order to provide adequate Configuration Management, an organization must organize an independent Configuration Management function [DAL 79]. This Configuration Management function does not need to be large and can consist of one person. The important point is that it must have a level of authority commensurate to the needs of the organization. It must be given the authority to enforce change control.

STEP 3: ORGANIZE AN INDEPENDENT CONFIGURATION MANAGEMENT FUNCTION.

Baselines

When an organization has adopted a software life cycle methodology and has created an independent Configuration Management Function, the next step is the identification of baselines within the software life cycle methodology. Obviously, the Design Specification and the software itself must be controlled. In order to really control software development, the source of the change itself must be controlled: which means the user input or requirements. This fact is often overlooked, and yet user input provides a significantly greater percentage of change in a system than anything else [BRA 79].

Change control must occur for all items produced in the software life cycle, especially requirements. In order to create a quality system, every document produced should form the baseline on the software life cycle path and changes to them controlled.

STEP 4: VIEW EVERY DOCUMENT AS A BASELINE FOR CONTROL ON THE SOFTWARE LIFE CYCLE PATH.

Standards

Standards for documentation and coding vary from organization to organization. Unfortunately, many companies do not use any standards at all, or use a minimal set so as not to stifle the creativity of the software engineers. It is impossible to create an environment where quality is paramount when no effort is made to qualify that quality [FOR 80].

It is important that your organization create a set of standards that address documentation formats and content, coding rules, acceptable programming languages, program and file naming conventions, etc. It is crucial that the development of these standards be

accomplished by a group consisting of users, software engineers and managers. This is important because standards must be fostered from a grass roots level in order to be effective and accepted.

Standards must be flexible and must be able to evolve with the changing needs of the organization. The standards should be controlled just like any other baseline and changes should be heard by the Configuration Control Board and given the widest possible audience.

STEP 5: ESTABLISH FLEXIBLE DOCUMENTATION AND CODING STANDARDS.

SUMMARY

There is no magic formula for creating quality software. It requires common sense and hard work. It also requires a committment from both managment and software engineers. If quality software is going to be produced, it must be created in an environment where control is perceived to be simple yet effective.

If software engineers feel that they have the flexibility to be creative but have a responsibility to use a discipline in developing their products, the only result can be quality software. As these same software engineers begin to maintain the systems they have produced, the payback will become obvious [HIL 76].

REFERENCES

[COO 79] John D. Cooper, Matthew J. Fisher, "Software Quality Management," A Petrocelli Book, 1979.

[BER 79] Edward H. Bersoff, et. al., "Software Configuration Management: A Tutorial," IEEE EHO-169-3, p. 24.

[BER 80] Bersoff, Henderson, and Siegel, "Software Configuration Management," PRENTICE-HALL, 1980.

[BRA 79] Philip H. Braverman, "Managing Change," Reprinted in IEEE EHO-146-1, p. 270.

[DAL 79] Edmund B. Daly, "Organizing for Successful Software Development," DATAMATION, December 1979, pp. 107-120.

[DEP 84] Robert W. DePree, "The Long and Short of Schedules," DATAMATION, June 1984, pp. 131-134.

[FOR 80] Joel J. Forman, "Implementing Software Standards," COMPUTER, June 1980, pp. 67-69.

[HEN 80] K. L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," IEEE TSE, January 1980, pp. 2-13.

[HIL 76] Chester C. Hill, "A Look at Software Maintenance," DATAMATION, November 1976, pp. 51-55.

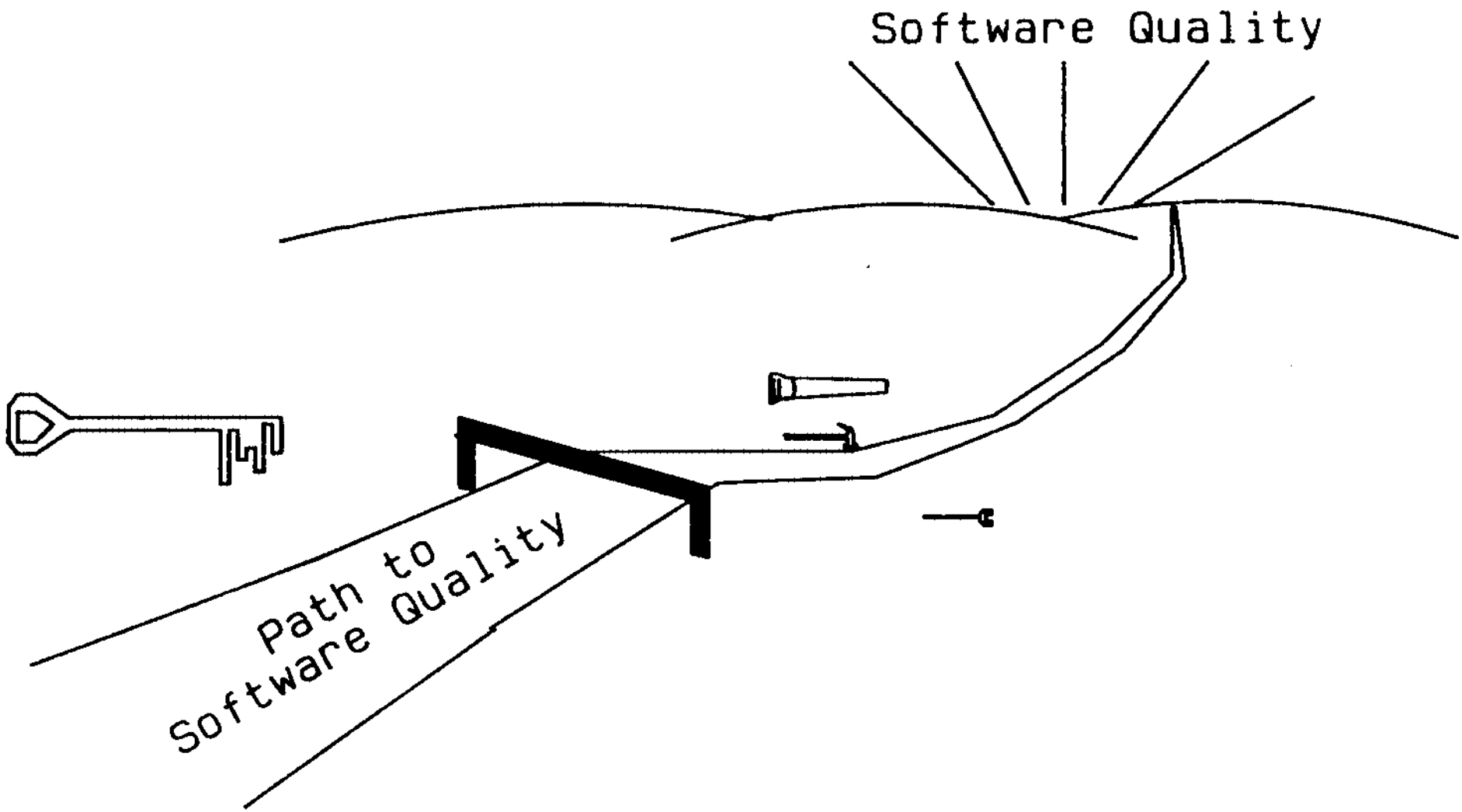
[MAR 78] Tom De Marco, "Structured Analysis and System Specification," YOURDON, INC., 1978.

[NIE 82] Raymond E. Nienburg, "Software Development Procedures Manual," Integrated Computer Systems, 1982.

[PUT 80] Lawrence H. Putnam, "The Software Life-Cycle Model Concept: Evidence and Behavior," IEEE EHO-165-1, p. 15.

The SOFTWARE QUALITY MINDSET

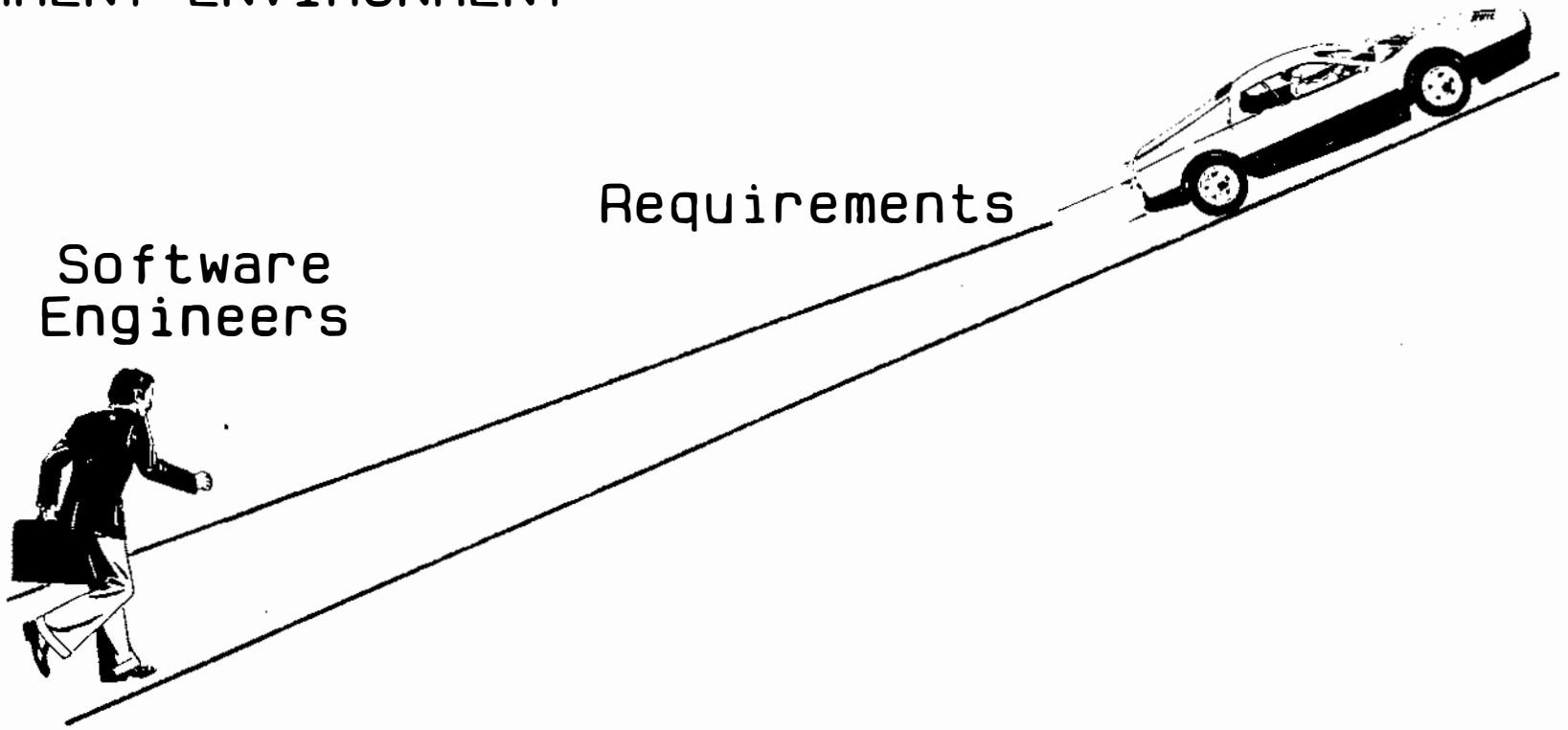
SOFTWARE QUALITY MINDSET



MEM - 9/84 - 2

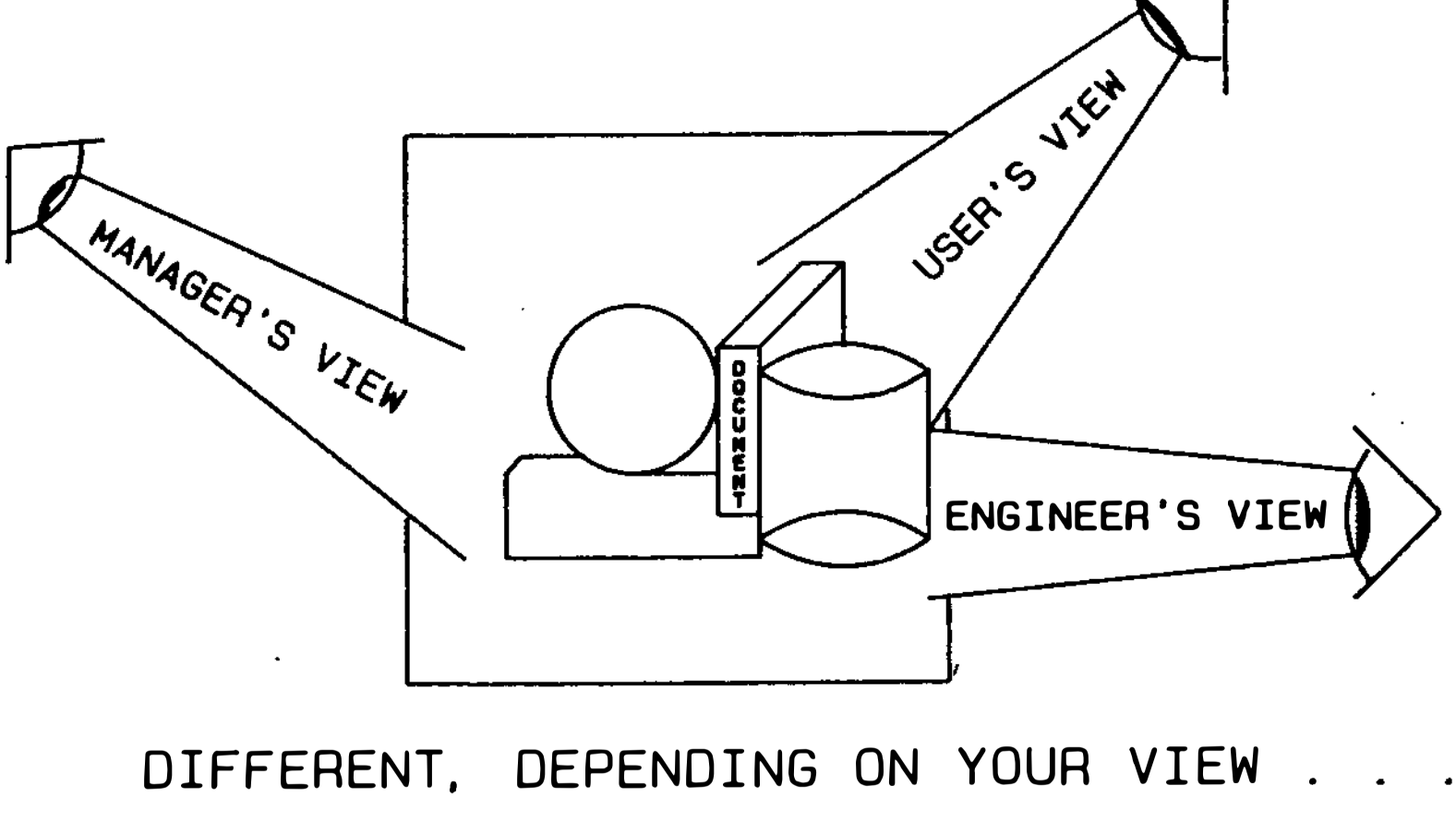
SOFTWARE QUALITY MINDSET

CURRENT ENVIRONMENT



SOFTWARE QUALITY MINDSET

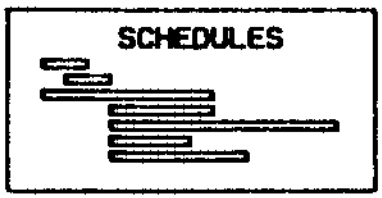
WHAT IS SOFTWARE QUALITY?

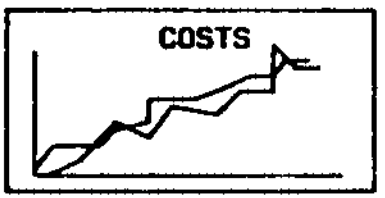


DIFFERENT, DEPENDING ON YOUR VIEW . . .

SOFTWARE QUALITY MINDSET

THE MANAGER'S VIEW

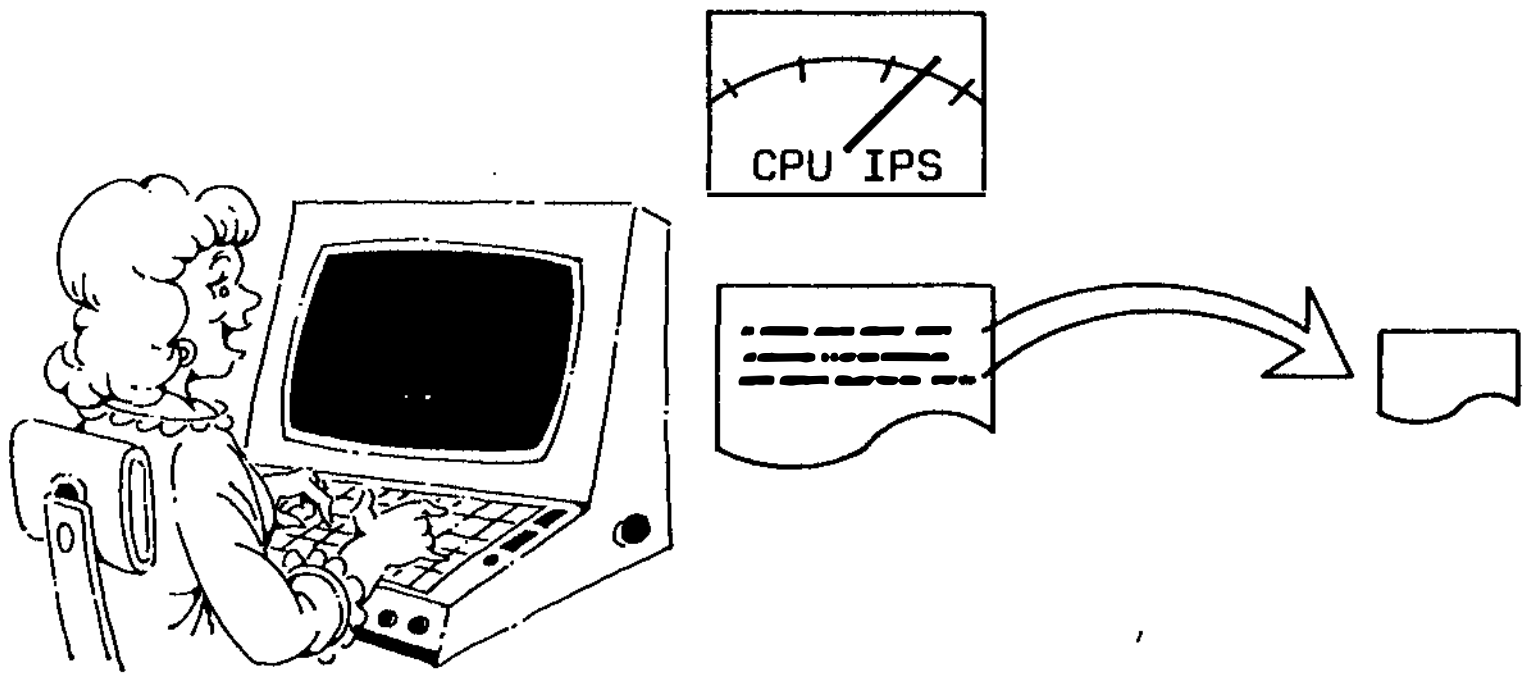




TIME AND
COSTS

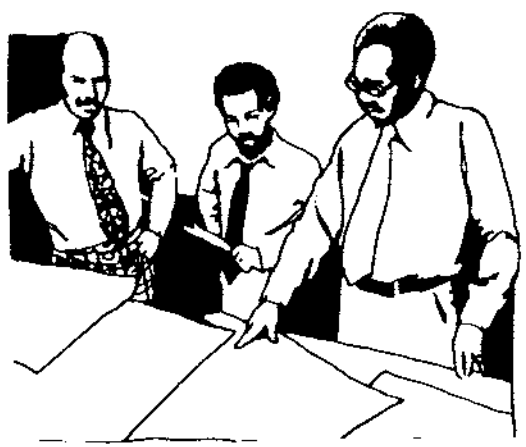
SOFTWARE QUALITY MINDSET

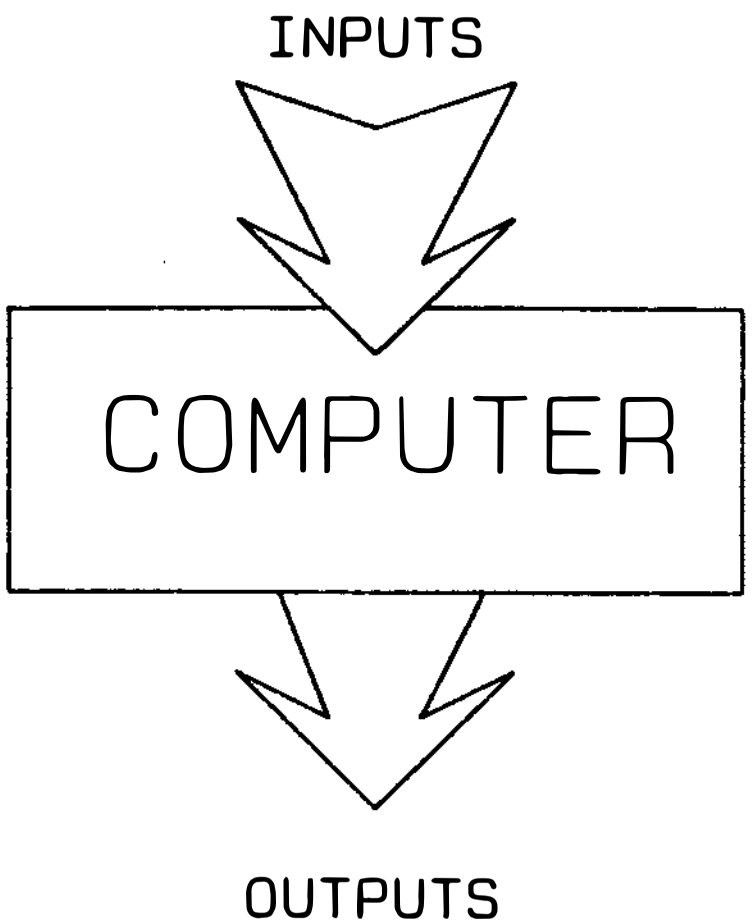
THE ENGINEER'S VIEW



SOFTWARE QUALITY MINDSET

THE USER'S VIEW





SOFTWARE QUALITY MINDSET

HOW DO WE MEET THESE DIFFERENT PERSPECTIVES?

MORE TOOLS. . .

MORE ELEGANT METHODOLOGIES. . .

REVOLUTIONARY NEW PRODUCTS. . .

JUST WHAT ???

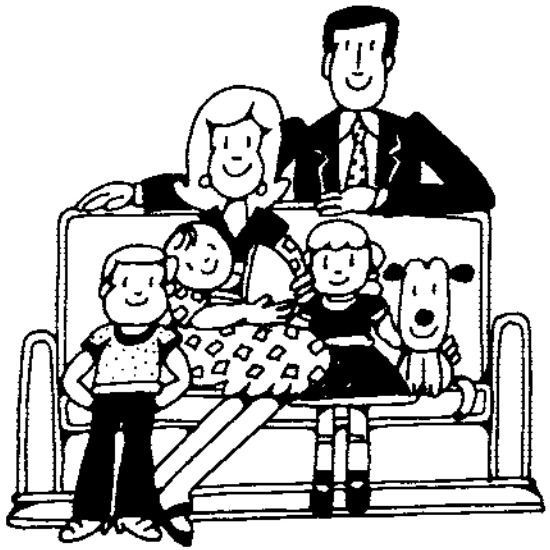
SOFTWARE QUALITY MINDSET

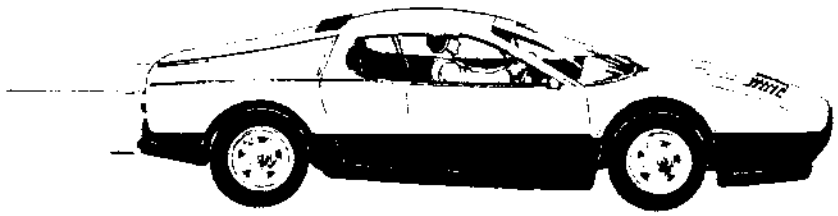
BACK TO BASICS !

- APPLY A BUSINESS APPROACH
- ADOPT A SOFTWARE LIFE CYCLE
- CREATE A CONFIGURATION MANAGEMENT FUNCTION
- ESTABLISH BASELINES
- ESTABLISH STANDARDS

SOFTWARE QUALITY MINDSET

BUSINESS APPROACH





DO A COST BENEFIT ANALYSIS. . .

SOFTWARE QUALITY MINDSET

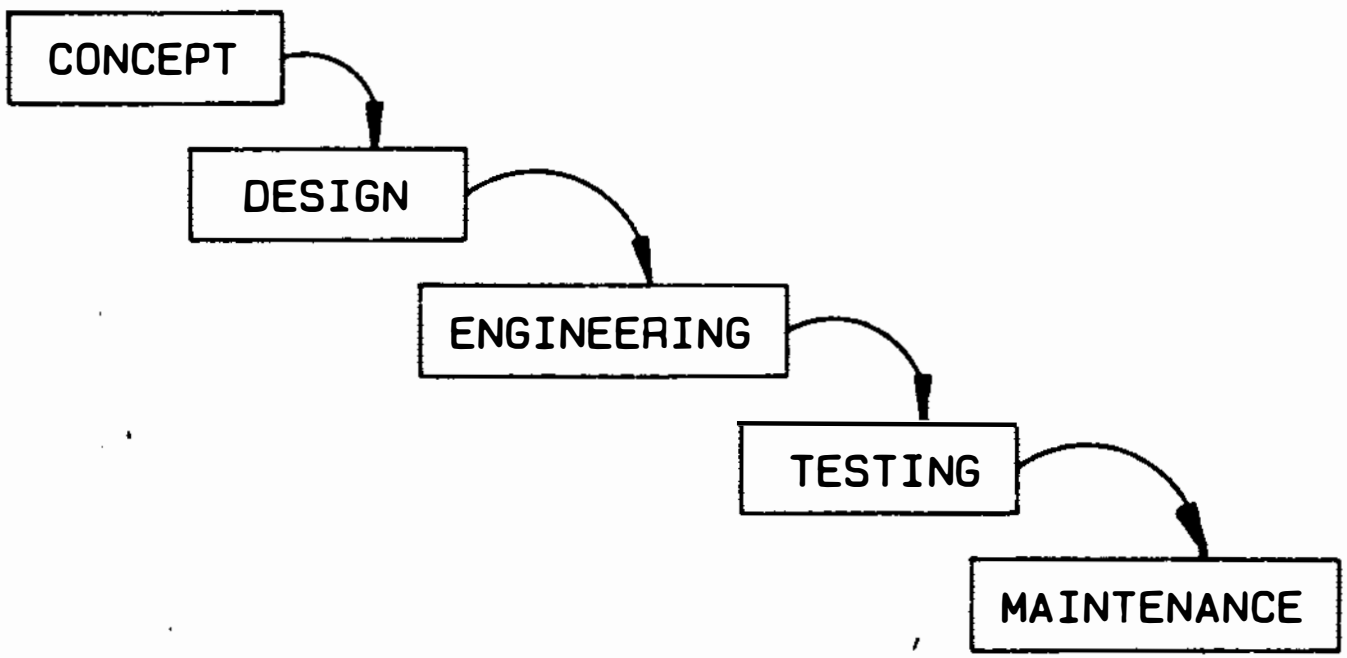
BUSINESS APPROACH

- LIMITED RESOURCES
- REALISTIC REVIEW OF
  - REQUIREMENTS
  - FEASIBILITY
- PRIORITIES

FOR EVERY SYSTEM ! !

SOFTWARE QUALITY MINDSET

SIMPLE SOFTWARE LIFE CYCLE METHODOLOGY



KEEP IT SIMPLE !!

SOFTWARE QUALITY MINDSET

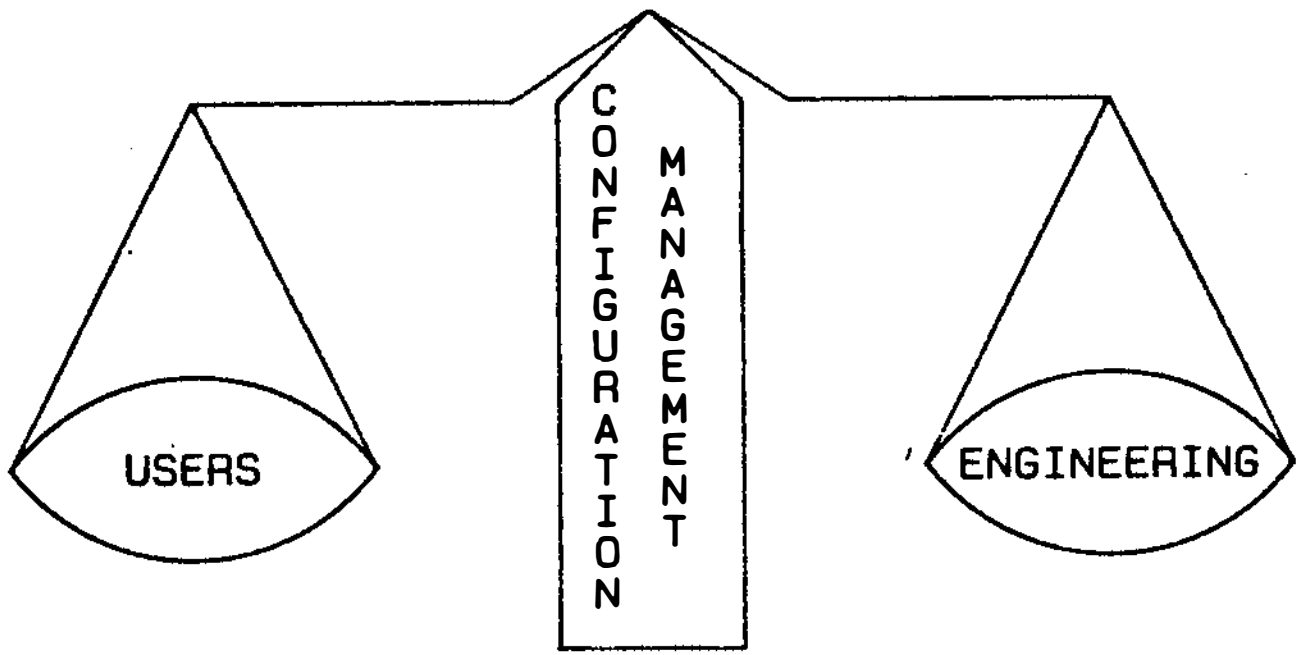
SIMPLE SOFTWARE LIFE CYCLE METHODOLOGY

- EASIER TO IMPLEMENT
- EASIER TO REMEMBER
- EASIER TO MECHANIZE
- LESS "BUREACRACY"

SOFTWARE QUALITY MINDSET

INDEPENDENT CONFIGURATION MANAGEMENT

PROVIDES BALANCE BETWEEN USERS
AND ENGINEERING ORGANIZATION



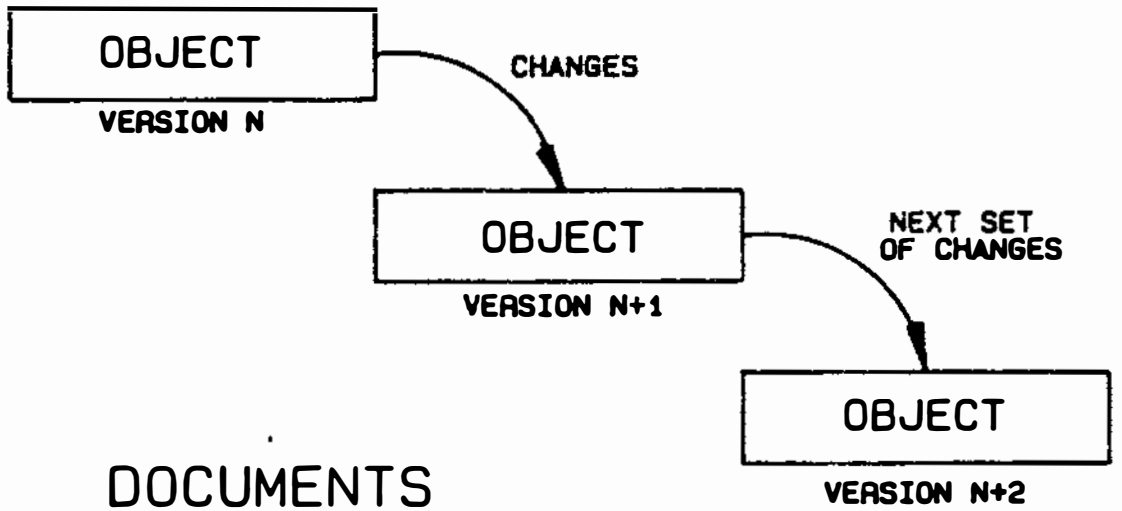
SOFTWARE QUALITY MINDSET

INDEPENDENT CONFIGURATION MANAGEMENT

- REDUCES CONFLICT OF INTEREST
- PROVIDES ARBITRATION
- MANAGES THE CONFIGURATION

SOFTWARE QUALITY MINDSET

EVERYTHING A BASELINE



DOCUMENTS
SPECIFICATIONS
SOFTWARE . . .

SOFTWARE QUALITY MINDSET

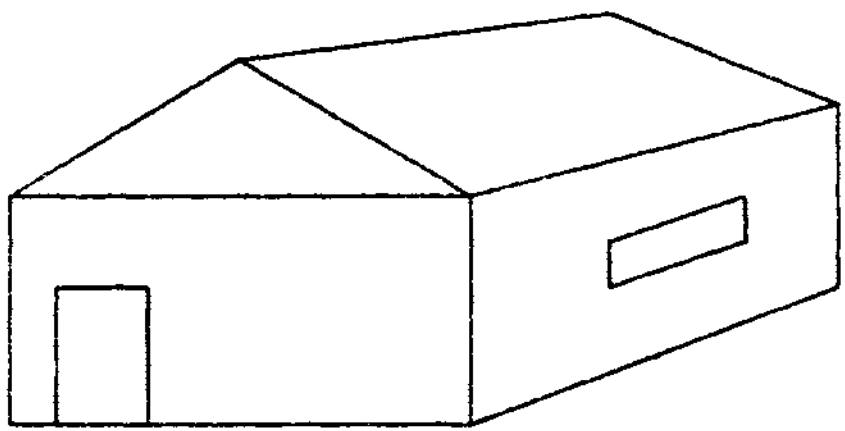
EVERYTHING A BASELINE

- CONTROLS ALL CHANGE
- INVOKES A DISCIPLINE

SOFTWARE QUALITY MINDSET

STANDARDS

A MUST FOR QUALITY. . .



BUILDING CODES,
COMMUNITY ARCHITECTURAL CONTROL
ETC. . .

SOFTWARE QUALITY MINDSET

STANDARDS

THESE SHOULD COVER. . .

- DOCUMENTS
  - TYPES
  - STYLE
  - CONTENT
- CODING
  - CONVENTIONS
  - FILE NAMING

SOFTWARE QUALITY MINDSET

STANDARDS

MANAGEMENT

CONSENSUS

SOFTWARE
ENGINEERS

SOFTWARE QUALITY MINDSET

SUMMARY

- NO MAGIC FORMULA
- COMMON SENSE ENVIRONMENT
- THEN COME TOOLS

QUALITY CAN ONLY FOLLOW. . .

Michael E. Meyer

Michael E. Meyer is the Manager of CAX Data Management within the CAX Center of the Computer Science Center. He joined Tektronix in July, 1983 after being with Honeywell Information Systems for over five years. With over twenty years experience in the area of Data and Systems Management he has worked for Computer Sciences Corporation, Planning Research Corporation and was the Assistant Director of the Computer Center at the University of Kansas. Assignments with these companies took him to several overseas and domestic locations.

A native of the Pacific Northwest, Mike spent four years with Weyerhaeuser Company in Tacoma, Washington and has an LL.B from LaSalle University, B.S. in Business from the University of Maryland, and a M.S.A. in Information Processing Technology from George Washington University.

2 Control and Tracking of Software Projects

Mr. William Edmark; Intel
"Maintaining File Integrity on a Large-Scale Development Project"

Ms. Monika Hunscher; Floating Point Systems
"MUTATE – A Module Tracking System"

Mr. David T. Cassing; Tektronix
"Yellow-Tag Software Project Management"

Maintaining File Integrity on a Large-Scale Development Project

William S. Edmark

Integrated Systems Operation-North
Intel Corporation
5200 NE Elam Young Parkway
Hillsboro, Oregon 97123

ABSTRACT

With the ever-increasing size and complexity of software projects, the need to accurately track and maintain every piece of code rapidly becomes imperative. Software projects are now at the point that they can no longer be maintained without automated help. This paper discusses a set of library and project management tools which will allow a large software project to be divided into smaller projects of a manageable size, track dependencies, control and track code modifications, and provide a centralized depository for the entire project. By providing the users with reliable information about the state of each file, the completeness of each project, and a means to recreate any phase of the development, the overall development cycle can be better assured of keeping on schedule.

The approach used was to take a known set of reliable utilities that provided this means for individual files and add an interface providing for multiple-file, as well as project, control. The interface makes use of Walter Tichy's well-known RCS system running on a UNIX system.

The implementation was done in three stages. Stage 1 was coded in C-shell for speed of implementation and contained a minimal set of utilities. Stage 2 added a full implementation of all functionality but was still coded in c-shell. Stage 3 was the conversion of everything to c code. The completion of Stage 3 should provide an decrease in turn-around time for the user.

By furnishing a relatively easy and painless way to control source code, the typical problems of large-scale projects can be minimized while giving the developers the freedom to concentrate on producing bug-free code. At the same time, the project leaders are given more time to control their projects.

July 25, 1984

Maintaining File Integrity on a Large-Scale Development Project

William S. Edmark

Integrated Systems Operation-North
Intel Corporation
5200 NE Elam Young Parkway
Hillsboro, Oregon 97123

INTRODUCTION

DECM stands for Development Environment Control Management. It was the result of a Software Design department evolving towards large-scale operating systems development and the inherent problems of projects of ever increasing size. It seemed like just overnight that we saw the quantities of files or modules double and the number of design engineers increase proportionately.

When we passed the 1000 file mark (files in a single project), it became apparent that we had to do something to maintain any control over our handywork. It was too easy to let a file or two (or more) slip through our fingers and get forgotten, duplicated, deleted, or put in the wrong place for a release.

It also was too easy for parallel modification to occur. Two or more engineers would need to modify the same file in order to complete their work, but would not know about the other engineer's modifications. This usually resulted in each copy being kept separately until release time and only one actually finding its way into the product, with the other "fixes" being folded into an update.

Another common problem was using unofficial versions of source, tools, utilities, compilers, or whatever while doing development work. Too often the result was incompatible code due to an inconsistent environment at compilation time. In addition, we found that it was highly desirable to keep all operating systems, tools, utilities, compilers, etc. in a centralized "library" so that each developer could easily obtain the "latest and greatest" to work with. The same should occur with source code in order to guarantee that everyone works from the same versions of source as well as the same versions of tools. However, maintaining a library of over 2000 files with an unknown number of revisions was a very difficult job. Similarly, doing development work on pre-production hardware does not insure that the software will work on actual production hardware.

Given the above problems, the design of DECM needed to fulfill several requirements. The first requirement was to automate the process of tracking every file or module as well as be able to determine quickly and easily which were ready for integration into the system, ready for inclusion in the product, or has bugs and needs additional work. This necessitated maintaining multiple revisions of each file. From a human standpoint, it required the ability to break a large project (Operating System) down into successively smaller pieces to make them more manageable. In this way, individual responsibilities could easily be assigned to engineers.

PHILOSOPHY

The approach taken utilized as many existing tools as possible. If we had found anything on the market that satisfied our needs, we would have used it. In addition to the above requirements, it was necessary that these tools would run under UNIX+ 4.1 bsd. The use of any tools to fulfill the above must do so without impeding any of the Development Engineers. Lastly, it was necessary to come up with a working solution in three months!

RCS

The available tools were narrowed down to RCS+, authored by Walter Tichy. It had the fewest commands for the novice user to learn, yet was the most powerful with all of its options. It was also virtually bug-free in Release 3 and provided better security than anything else found.

RCS consisted of a set of utilities that provide for the storage and retrieval of multiple revisions of a source or text file in a space efficient way. Changes of file content would not destroy the original because the previous revisions remained accessible. It also provided a means to resolve access conflicts to a file.

Separate lines of development could be maintained for each file. RCS would store the revisions in a tree structure representative of the ancestral relationships. Revisions could be assigned symbolic names and marked as released, stable, experimental, etc.

The RCS commands:

ci: check-in a file to an adjacent RCS directory.
co: check-out a file from an adjacent RCS directory.
rcs: change some of the attributes of an RCS file.
rcsmerge: combine two revisions of an RCS file
into a working version.
rcsdiff: display the differences between a previous revision
and the file in the current working directory.
rlog: display the history of modifications to an RCS file.

INTERFACE DESIGN

It was desirable to maintain, as much as possible, all of the RCS options and command conventions. It seemed that the easiest and cleanest way was to design a filter as an interface. As all of you (that are familiar with UNIX) know, there are probably as many different working environments as there are users. Due to this feature of UNIX, either a "safe" environment must be created or the interface must account for every possible deviation from a "normal" environment.

By piping through the command line, it was possible to add options to those already existing in RCS without having to modify RCS. We wanted a useable, portable tool that could be used on other UNIX's without having to worry about RCS code changes, author permissions, licensing, etc. It also means that we could use some other tool instead of RCS, if it became necessary.

One extension to RCS was to allow directories, including all of the subdirectories, to be accessed with a single command. A directory listing utility was developed that would explore all subdirectory paths. Another extension was the inclusion of the "Project" concept. To make this work, a very simple database was used. A project definition requires the inclusion of every file with any interdependencies to the project. This definition will allow all of the defined files to be accessed with a single command. The included files can exist in different directories, in the same directory, or at different directory levels. More than one project can include the same file or set of files. The Lp utility was developed to retrieve the contents of any project definition for the user.

The one drawback to RCS as the underlying tool was that it required that the user have read-write-execute (rwx) access on every directory. The files are kept to read only by owner and group, but with the world having directory write access, any file could be accidentally deleted. Our solution around this was to make DECM so easy to use that the user had no need to ever physically enter or access the DECM directories, and to make it impossible for the user to check-out files while the current working directory was a DECM directory. The user's current working

+ UNIX is a registered trademark of AT&T
RCS is a registered trademark of Walter Tichy.

directory could be anywhere desired OUTSIDE of the DECM partition. This will allow each developer to work and interface DECM from inside his/her own world, away from everyone else. The user could maintain their right to privacy. The implementation section will add some more to this problem.

IMPLEMENTATION

Safe Environment

Obtaining a safe environment in UNIX means checking for every possible alias, variable, path, etc. or forcing a known environment on the user. We chose the latter approach. If the user exists under the Bourne-shell, the C-shell would be invoked with a DECM-built .cshrc file which would include a known set of variables and aliases. If they were already in C-shell, we saved their .cshrc file for later restoration and replaced it with a copy upon which DECM information had been appended. Any user aliases that would not interfere with DECM operation would remain functional. One of the users that forced this action was an individual that would alias the @ sign (used by awk for math variables) as an execute or source command. All interrupts would be trapped to allow a graceful bail-out from DECM. The only event that we could not catch was a system/VAX crash.

Stages of Implementation

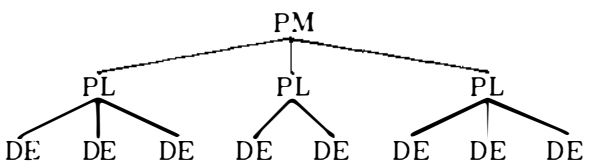
The nicest thing about using RCS was that most of the hard work was already done. All that was left was to implement the interface between the user and RCS. The implementation was performed in three stages. Stage One was a minimal implementation of the fewest RCS commands possible. This was done in C-shell to get the implementation running in the shortest amount of time. The second Stage added the full set of RCS commands as well as our extensions, but was still coded in C-shell. The third Stage, as of this writing, was just being started. This will be the conversion of the C-shell scripts to c code. Stage three will result in two desired benefits: faster response time for the user, and better security by being able to setuid. Setuid will allow the directories to be only read and executable by the group. The owner of all files will be the DECM administrator, with each legal user assuming write permissions only while executing the check-in and rcs commands. At that point, a reasonable sense of security will be achieved. By doing the debugging and functionality modifications in C-shell, we have minimized the time needed to become fully operational.

POLICIES

The implementation of some firm policies was necessary for the successful operation of DECM.

Project Leaders

There must be an hierarchy of individuals responsible for successively smaller portions of the overall project. The following hierarchy is typical of Intel:



The overall leader is called a Project Manager (PM). The PM is responsible for the overall project which is broken into smaller chunks also called projects. These projects are then assigned to Project Leaders (PL's) who are responsible for defining those projects in DECM, assigning the projects to Development Engineers (DE's), documenting the projects, and the

scheduling of the projects.

Project Assignment

The assigning of a project involves the PL checking the project out "locked" and in turn giving those files to the designated DE. The reason for locking the files on check-out is so that no one else can check any of those files in without the responsible PL knowing about it via electronic mail.

File/Project Check-In

All checked-in files are assigned a "state" as an indication of their current condition. "Exp" is the default state for untested files. A file or project which has passed Developer testing will be stamped with a Base Level state to indicate that the file is ready to be passed on to other departments. Another Base Level state will indicate a file that is ready for transition testing, and a third Base Level state will indicate a file that has been internally released. The fourth Base Level state will indicate a file included in an actual production release. The Base Level indicators are:

BL*nnn*X*nn* = ready for Internal Release
BL*nnn*I*nn* = released Internally
BL*nnn*T*nn* = released for Transition Testing
BL*nnn*R*nn* = included in Production Release

If a file fails in transition testing, the state will be returned to Exp. No file will be included in any release if it has not gone through DECM. No engineer will pass any software on to anyone else. All software will be transmitted through DECM.

RESULTS

DECM appears to be providing a positive contribution to the XENIX+ project. The turn-around time for system generation has been shortened by three quarters of the previous time. The status of every file included in the system is known. Every file included in a system defined in DECM will be present only where and when the definition specifies. Each system generated can be identified, even unauthorized or illegal ones. Future support of released products will be enhanced due to absolute identification and ease of retrieving everything known about it.

Development Engineering is relying on the security that

-they can not lose any files
-they know exactly what they are working with
-they do not lose any time due to "wrong" or "misplaced" files

As a result, the quality of the product is improving dramatically. Confidence in preliminary releases is increasing also.

+ XENIX is a registered trademark of Microsoft Corporation

intel

What is DECM?

- Development Environment Control Management
- Result of:
  - number of files
  - number of engineers
  - parallel modifications
  - unofficial versions
    - → source
    - → tools
    - → utilities
    - → compilers
    - → other

ISO-N

intel

DESIGN REQUIREMENTS

- automatic tracking of each file
- status information on each file
- maintain multiple revisions
- parcel large project down to small

PHILOSOPHY

- use existing tools
- run on UNIX 4.1 bsd
- run on UNIX 4.2 bsd
- must not impede any project development work
- working in 3 months

ISO-N

intel

RCS

- Best set of utilities available
  - fewest commands to start
  - most powerful later
  - Release 3 bug-free
  - provided best security

ISO-N

intel

RCS Functions

- Store and retrieve multiple revisions
- Stores only changes between revisions
- Allows files to be "locked" by user
- No one else can check-in a locked file
- Stores revisions in tree structure
- Uses symbolic names for revisions
- Uses states for revisions

intel

RCS Commands

These commands find a named file in users
current working directory
or in a RCS directory under the cwd

- ci: check-in a file to RCS
- co: check-out a file
- rcs: change attributes of RCS file
- rcsdiff: display revision differences
- rcsmerge: combine 2 revisions of RCS file into a working version
- rlog: display file attribute information

intel

Interface

- Maintain original RCS options
- Maintain original RCS command conventions
- Provide a "safe" environment
- Provide extensions to RCS
  - directories and subdirectories
  - projects (with files in different directories)
- Additional commands
  - Ls: recursive directory listing with full pathnemes
  - Lp: lists projects and contents
  - defp: append, remove files from project
  - mov: rename files in DECM; update projects
  - remov: delete files from projects and DECM
- Limitations for RCS
  - the cwd must not be in DECM file structure

intel

IMPLEMENTATION

- Safe environment
  - C-shell can not use setuid
  - RCS requires user have rwx access on dirs
  - conversion to c will allow setuid
- Stages
  - Stage One
    - → ci
    - → co
    - → rcs
    - → rlog
  - Stage Two
    - → rcsdiff
    - → rcsmerge
    - → defp
    - → Lp
    - → Ls
    - → mov
    - → remov
  - Stage Three
    - → faster response time
    - → directories rx only

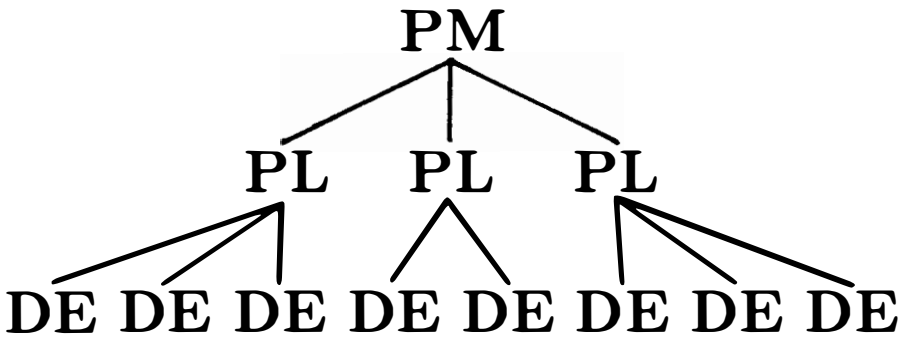
ISO-N

intel

POLICIES

- Responsibility hierarchy
  - PM (Project Manager)
    - → overall responsibility
    - → scheduling
    - → documentation
    - → assignments
  - PL (Project Leader)
    - → subset of main project
    - → works with individual files
    - → assigns files or subprojects to DE's
  - DE (Development Engineer)
    - → performs actual work
    - → designs
    - → implements
    - → debugs
    - → documents

Hierarchy



intel

PROJECT ASSIGNMENT

- Check-out "locked" project
  - RCS file then indicates "locker"
  - No one else can check-in a file in that project
  - Lock can be broken, but locker notified

ISO-N

intel

CHECK-IN PROJECT

- Assigned "state" to indicate status
  - default state is Exp (experimental)
  - Base
    - → BL for Base Level
    - → 3 digits for number of level
    - → number incremented for major source changes
    - → Letter indicates status
      - → X = ready for internal release
      - → I = released internally
      - → T = released for Transition testing
      - → R = production Release
- A file failing Transition Testing returns to Exp
- All software transmittals go through DECM

ISO-N

BIOGRAPHY

Bill Edmark is a Software Engineer at Intel where he has worked since June, 1981. During this time he has worked on a wide variety of products ranging from industrial automation applications to compilers and editors. During the past year, Bill has designed and implemented the Development Environment Control Management (DECM) system which maintains file integrity for the Xenix development project.

MUTATE: THE BEGINNINGS OF A SOFTWARE PRODUCT MANAGEMENT SYSTEM

BY MONIKA HUNSCHER
Methods, Standards, and Quality Assurance Mgr.
Floating Point Systems, Inc., Beaverton, OR

1: INTRODUCTION

MUTATE is the result of an effort to improve the control and tracking of software products at Floating Point Systems. Due to the large number of supported computers for the array processor, the quantity of software products increased dramatically and with these the number of problems and time associated with developing new software product releases.

A typical problem is as follows:

> We ship a new release of a software product to a customer. The customer encounters a problem that didn't exist in previous releases of the product and submits a software problem report. Engineering's first action is to reconstruct the previous release of the software to see if the problem existed there as well. If it turns out that the problem arose between releases, it is impossible to find out when it was introduced and engineering spends a lot of time correcting it.

With MUTATE this problem is easily identified and corrected. Using the file history information in MUTATE, all the modules that were changed for the new release are identified and the faulty code found and fixed quickly.

Change Control is only one of the many concerns that prompted us to develop a set of tools to help us better control and track our software products. The other major concerns are: the need to improve tracking of software product configurations, the need to improve our accuracy in tracking software problem reports, the need to improve the software product manufacturing process, and the need to ensure the integrity of our data bases.

The key activity for controlling and tracking software products is the establishment of a centralized software product data base [1]. Once you establish this data base, you can generate many tools to manage the stored information.

This is exactly the approach we took in developing MUTATE. To appreciate the power of MUTATE, you need to take a closer look at some of the problems creating the need for a software management system.

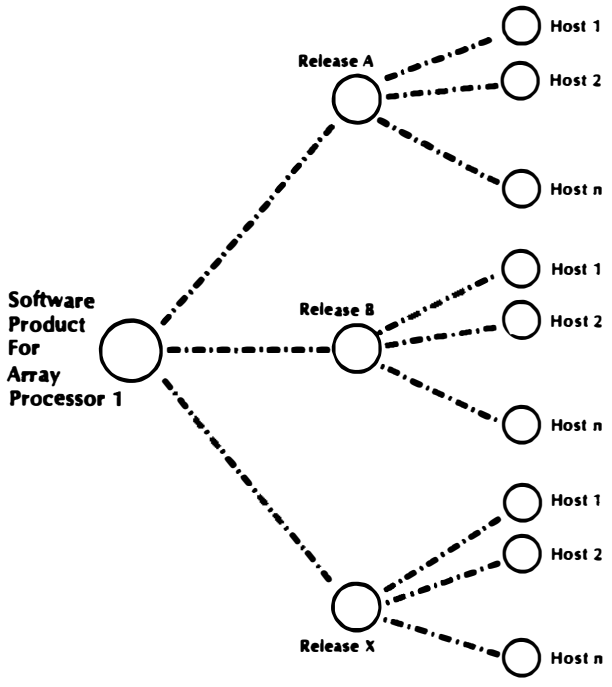
2: PROBLEM CHARACTERISTICS

2.1 General Industry Problems

Many of our software management problems at FPS are not unique to our environment, but belong to the software industry at large. We must develop more complex software at a faster rate, but find that more and more engineers spend their time maintaining and managing existing software products. Since software management is usually left to the individual, the final product is not integrated and well structured and therefore cannot be managed by a set of tools. We therefore use our highly-trained engineers for product management rather than new product development activities.

2.2 FPS Software Product Proliferation

FPS designs array processors to be attached to a host computer. Much of the software supporting the array processor executes on the host computer. Since the different FPS product lines support many different host computer interfaces and operating systems, the management of our software products is now quite complex. Since no two hosts have identical compilers and assemblers, our products, though functionally similar, require specific changes for particular hosts. This multiplicity of similar software products leads to large amounts of code and documentation, which we must manage.



FPS Software Product Proliferation

An added complication is the fact that not all supported hosts are easily available to FPS, and the correct management of these products is extremely important. Losing track of software product changes in such a case can be very expensive, since we could potentially be spending a great deal of time in the field supporting a poorly-managed software product.

2.3 Change Control Problems

Let's look at some of the main software change control problems FPS faced:

Traditionally, software product management was the responsibility of every product development group. Most of the engineering effort was spent on product development, and we ignored product management until a software product release was near. Then we attempted to pull the individually-developed product components, tools, and procedures together into a product data base. This effort was usually not a scheduled activity and, depending on the amount of software and the number of engineers involved in the development effort, the integration of the product often turned into a lengthy, error-prone process. This was especially true if the development effort had not been completed yet and many changes were still being added to programs.

Major problems tended to surface at this time. New versions of files appeared out of nowhere with no indication of what had happened to them since the previous release. The tools and procedures used for development could not always be found or recreated. Several engineers who worked on the same files changed them without proper communication causing the loss of some changes, and preventing other changes from ever making it into the data base because one engineer thought another one was going to do it.

Once we had a controlled data base the only way to make changes was by making copies of files and modifying them. Then we had to remember to change all occurrences of the same files in the data base.

Also, if a new version of a program didn't work, we could not go back to the previous working version, since we had no way to reconstruct it.

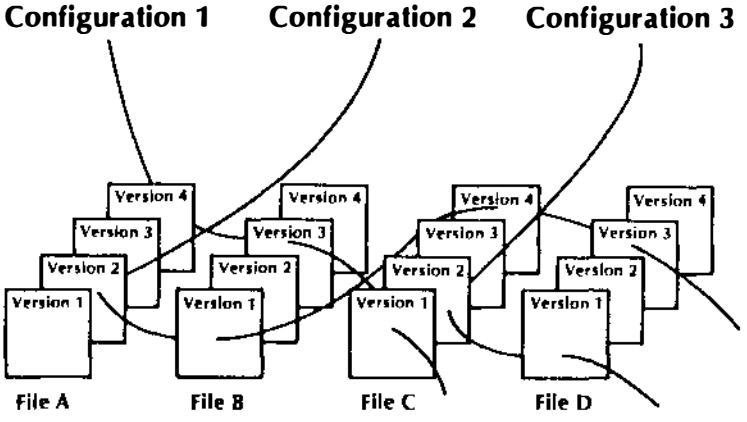
2.4 Configuration Management Problems

Managing software product configurations is complex at FPS due to the diversity of supported host computers and operating systems. A typical software product configuration for one host consists of about 500 files, which in turn were derived from about 2000 development files through various tools.

The old method of creating and maintaining software product configurations was largely manual and error-prone. To identify a particular configuration, we generated a list of pointers to files in the data base. In some cases, these product files still resided in an engineer's personal work area. Frequently, the pointers to files were "out of sync" with the latest version of files.

Since we could only manage changes by making copies of files, we constantly made copies of product data bases. This forced us to create new configurations and change existing ones manually. Trying to make the configurations for a product consistent was practically impossible, since different engineers worked on different data bases and had their own ideas of how things should be structured.

We archived all products after release and kept a manual log of the configurations. Using these methods, it was very difficult and time-consuming to bring back an old product configuration.



Configuration Management

2.5 Tracking of Software Problem Report

With so many different but similar products, we have a complicated software problem reporting system. Every report must be evaluated to determine if it applies to all host versions of the product (generic problem) or to only to one host (host-specific problem). For a generic problem, we need to find every occurrence of the defect. The manual tracking system did not work very well and, consequently, we could not guarantee that a problem was fixed in all releases and for all host computers.

Often, engineers fixed problems without updating the paper work. Similarly, some software problems didn't get fixed simply because we lost the paper work. This resulted in an inability to know which software problems were corrected and in what releases the corrections were made.

2.6 A Solution was Required

These problems added up to spending a lot of money and resources on recovering from totally unnecessary problems. The inability to correct all known software problems and release correct product configurations created an unprofessional image, and our customers demand high quality products.

Taking all of this into account, FPS management decided to support the development of MUTATE, hoping that a set of tools and improved procedures might reduce the cost and time of developing new software product releases.

3: OVERVIEW OF MUTATE

3.1 System Goals

The management of large amounts of software is complex, requiring many resources. We had neither the resources nor the understanding to completely automate the software management process at that time. Thus, we concentrated on a couple of key problem areas we felt we understood well enough to automate, and whose solution promised near term benefits.

Since we had made two previous attempts to implement MUTATE, our goals were already fairly well defined. Besides providing change control for files, configuration management for our products, and improved software problem report tracking, we had some specific goals based on the existing manual system.

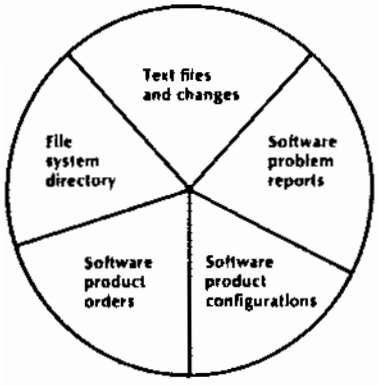
We wanted an integrated system with all tools interfacing with the central data base and consistently exchanging information for each other's use. Besides an integrated system, we also wanted tools which interacted with the user in a friendly and consistent manner, to reduce the learning curve of yet another set of tools.

We also wanted a portable system, since FPS develops software products on several computers, and we felt we should be able to provide MUTATE on whatever host the project decided to use to develop and manufacture the product. This also resulted in a requirement to move a MUTATE data base from one computer to another without a major conversion effort.

Our last two goals (and probably the more important ones) were to eliminate the paperwork associated with tracking software products and to reduce the errors associated with manual procedures.

3.2 System Description

To control and track all product software and have a set of tools which can exchange information about the software, we felt our the most important task was to establish a centralized software product data base, which is what we did.



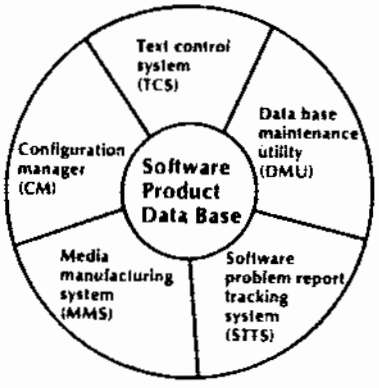
MUTATE Software Product Data Base

This data base contains all text files which are part of the product, including development files, tools used to build the product (like the RATFOR preprocessor), prototype files, and released product files. Every text file in the data base has a unique name, version number, and file type.

Besides the text files, the data base contains the file system directory information, text file history information, software product configuration information, software problem report information, and customer product order information.

Five distinct but integrated tools support the centralized product data base. These tools are the Text Control System (TCS), the Configuration Manager (CM), the Software Problem Report Tracking System (STTS), the Media Manufacturing System (MMS), and the Data Base Maintenance Utility (DMU). We manage the software with these tools and a set of manual procedures.

The following sections introduce each of the system components and describe the role each plays.



MUTATE System Tools

3.3 Text Control System (TCS)

TCS is the only component allowing the user access to files in the data base. It retrieves any file out of the data base on demand and inserts it back into the data base after changes. It generates and maintains a change history of all text files, and can reconstruct any version of any file that was ever in the data base.

To assure the structural integrity of the product data base, we introduced a few user restrictions. The user can only create and modify files. The privileges of creating the file system structure and deleting any information in the data base belong solely to the data base administrator. It is important to properly design the software product data base to avoid lengthy restructuring at a later time.

TCS performs some useful functions normally not available to users through the computer operating system. It prevents simultaneous access to a file by two users if one of them intends to update the file. It provides information on who currently "owns" the file and what changes have been made to files between specific versions, including when they were made and who made them.

3.4 Configuration Manager (CM)

The Configuration Manager tracks the different software product configurations and their current status. To do this, it creates and maintains the product definition files (PDF). A PDF contains three types of information: a list of files belonging to a manufacturable component, default manufacturing parameters, and a list of appropriate user documentation.

3.5 Media Manufacturing System (MMS)

The Media Manufacturing System accepts software product orders and, using the product definition files created by the Configuration Manager, manufactures released software product configurations for shipment to customers. It retrieves the appropriate files out of the product data base through the Text Control System, moves them to media for shipment, and generates the appropriate Bill of Materials information.

3.6 Software Problem Report Tracking System (STTS)

STTS keeps all information about software problem reports, whether they were reported by customers or internally by FPS engineers. It produces various reports about software problems and acts as an inquiry system for anyone needing information about the status of a particular software problem report.

3.7 Data Base Maintenance Utility (DMU)

DMU initializes new software product data bases and maintains existing data bases. It efficiently manages disk space use through archiving and restores whole software product releases or individual "old" files to and from tape. It also moves a product data base from one computer to another and insures the integrity of the data base at all times by verifying that the directory information and the text files match.

4: MUTATE DEVELOPMENT

FPS has struggled with software management for several years, and twice attempted to implement a software product management system before committing resources adequate to completing the task. We looked around the industry for software management systems matching our criteria, but we found nothing which came close. We found several programs though, like SCCS [2] and MAKE [3] which performed some of the needed functions, but they were unusable in our environment.

Before we could begin to develop a set of tools for the control and tracking of software products, we had to optimize some of our manual procedures until we saw enough repetition to justify automating them. This optimization did not happen overnight, but was the result of many procedures implemented by trial and error.

We confronted many difficult design decisions at the beginning of the development activity. We had no prior experience with automated software management tools, so some of our decisions were based on a coin toss rather than on facts.

A major decision involved the storage of information in the product data base. After many discussions, we decided to support our own file system structure instead of relying on the computer file system. Rather than using one of the available major data base management systems, we decided to use minimal ISAM (index sequential access method) host capabilities. We did not need all the features of a large data base management system and also feared too much overhead.

Several reasons helped shape the decision to support our own data base structure. First, not all of the hosts on which we wanted MUTATE to execute maintain the information we require, such as the last time a file was accessed, whether the file was online or offline at the time, and so on. To be able to provide the same information in the same manner on all hosts was a basic requirement of MUTATE, and the only way allowing us to do that was to have our own file system.

Our choice of development languages was limited at that time to Fortran or RATFOR (a structured language preprocessor for Fortran). Since FPS supports its own version of RATFOR, and since it allowed us to write structured programs which were more portable than equivalent programs in Fortran, we decided to use FPS RATFOR.

We carefully designed all components modularly and identified every routine as either host-independent or host-dependent. Since most of these conversion efforts consist of rewriting the host-dependent utility routines, this let us move MUTATE more easily to other hosts.

5: EXPERIENCES

5.1 During Development

We completed a functional specification for MUTATE in November, 1982. At that time, we felt comfortable that we understood the system requirements and that we could implement it in the allotted time.

Since they would be the ultimate users of the system, we attempted to have project engineers review the specifications, but we received little feedback. We feel this is due partly to the fact that they did not realize how important it would become to them, partly to the fact that they were busy with product development activities, and partly to their scepticism that the system would ever be operational.

During the coding, we realized that we would not meet our projected completion date. This was due to some resource problems, to the fact that we did not really answer all of the questions in the functional specification, and to the fact that the design specifications were not as clear as they should have been, making it difficult for our engineers to implement the system. Finally by November, 1983, we had the initial system completed and also had a long list of proposed enhancements.

By this time, MUTATE worked on the PRIME and VAX computers.

The first real test of the system came when we created a product data base for MUTATE itself. As the first users, we received early feedback on problems and unnecessary restrictions, so we were able to correct them before we turned the tools over to the project engineers, who would probably be less tolerant of some of MUTATE's new restrictions.

5.2 During Implementation

To familiarize the engineers with the system and overcome some of the resistance we knew was there, we developed a training course and user manuals, and also videotaped the training class for future use.

The acceptance of MUTATE was mixed. Engineers and managers who saw that it would solve some of their problems embraced it quickly, most resigned themselves to the fact that FPS would be managing software products with it, and some had few good words to say about it.

After six months of use some of the dust has settled. We have undergone one product conversion to a MUTATE data base system and are in the process of converting a second product. All new products start out using MUTATE, and within a year, all product data bases should be managed by MUTATE. MUTATE has worked reliably ever since we made it available. We currently divide our time between testing MUTATE's functions further and incorporating new features derived from user suggestions and criticisms.

6: BENEFITS AND SHORTCOMINGS OF MUTATE

MUTATE has given us some immediate benefits:

We now have complete text file version control. We can make changes to software products at any time and can link those changes to a particular software problem report. The paper work associated with changing controlled data bases is gone. Since nothing ever gets deleted, nothing is ever really lost, and any version can easily be recreated. Periodic verification helps ensure the integrity of the data base, and any problems due to computer crashes are immediately apparent and easily correctable.

We can easily identify specific software product configurations and the files which make up a particular release of a product.

Tracking software problem reports is now automated, including the generation of all reports. The system is reliable for determining the status of all software problem reports. When engineers correct a problem in the data base, they submit the software problem report number and MUTATE immediately updates the software problem report in the data base to reflect its new status.

Finally, we use much less disk space due to the fact that we no longer keep multiple versions of files.

On the other hand, MUTATE has a few shortcomings:

New overhead associated with extracting files out of the data base and storing them back again is inconvenient.

Additional personnel have been needed to manage the software product data base properly and perform all maintenance functions.

Since all accesses to the software product data base are through MUTATE, more computer resources are needed than before.

7: SUMMARY

The development of MUTATE is costly and so are the conversion of existing product data bases and the training of the people using the system. Consequently, we realize few short term gains.

On the other hand, uncontrolled software products eventually cost more in terms of increased resources required to maintain and manage them.

Tools like MUTATE give us a handle on the software we develop, and, in the long run, we expect to produce higher quality products with significantly fewer defects. Proper management of software enables us to develop new software product releases quicker. This should allow us to spend more of our time developing new and even better software products.

While many engineers feel threatened by the implied control of a system like MUTATE, its purpose is not to control their every action, but to provide an environment where computers perform tedious and repetitive tasks, and people pursue more creative activities.

We can accomplish this only if we manage our software products in a more structured and disciplined manner and a set of tools like MUTATE can help accomplish this.

8: ACKNOWLEDGMENTS

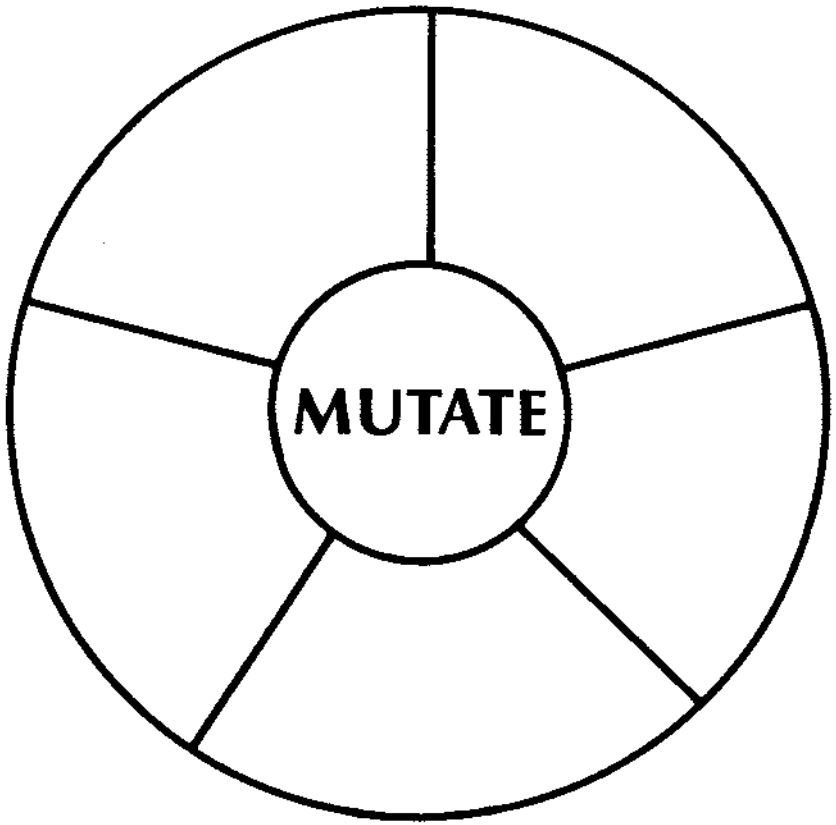
I want to thank Floating Point Systems for supporting the development and implementation of MUTATE. In particular, I want to thank David Carpenter, one of the principle designers of the system, for designing and preparing of some of the illustrations and for many enlightening discussions. I also want to thank Kori Deller of FPS Technical Publications for rendering the illustrations.

9: REFERENCES

[1] Boehm, Barry; Penedo, Maria H.; Stuckle, E. Don; Williams, Robert D.; and Pyster, Arthur B. "A Software Development Environment for Improving Productivity," Computer, Vol. 17 (6), June 1984, pp. 30-44.

[2] Rochkind, M. J., "The Source Code Control System," IEEE Trans. Software Eng., Vol. SE-1, No. 4, December 1975, pp. 364-370.

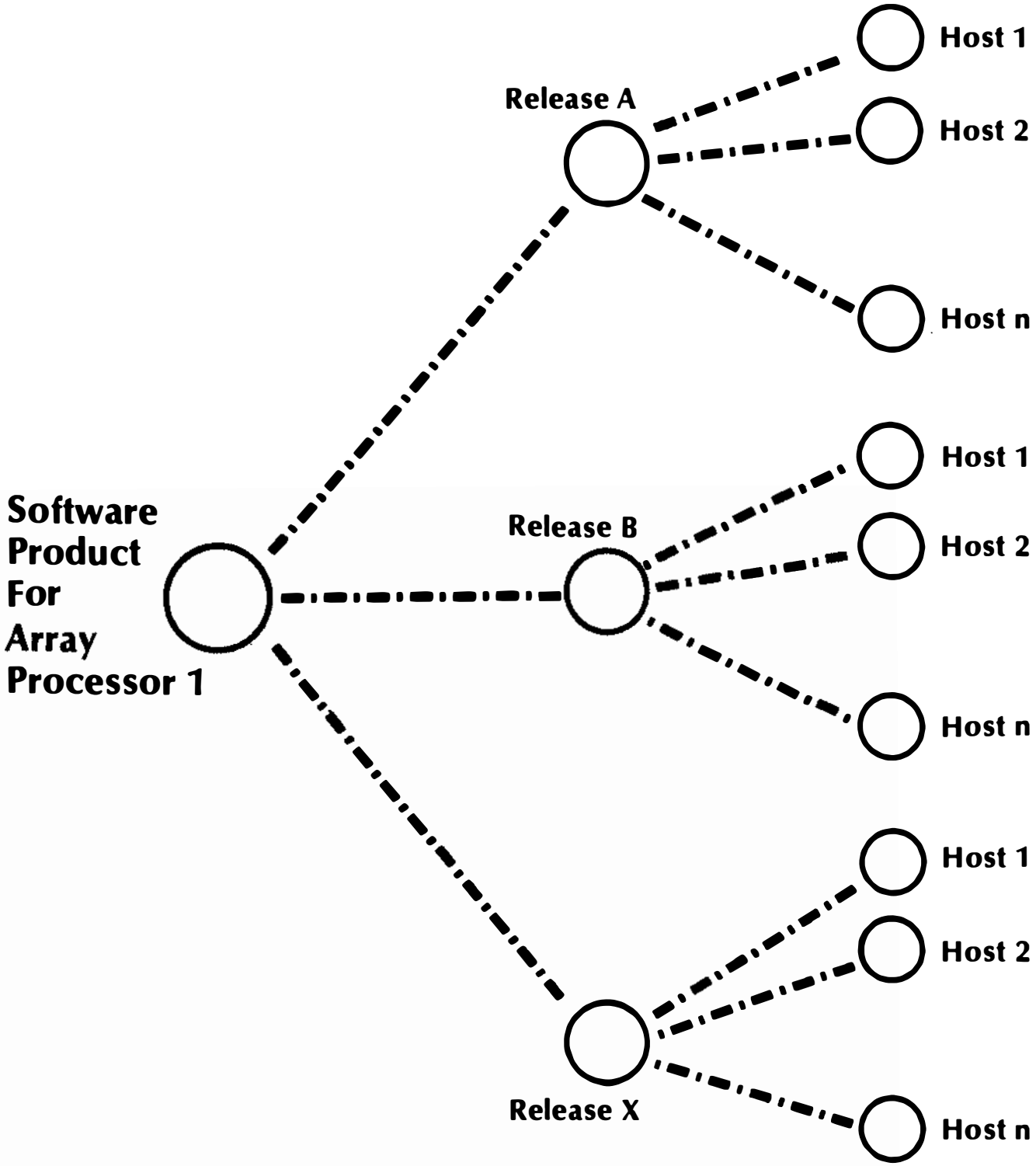
[3] Feldman, S. I., "MAKE--A Program for Maintaining Computer Programs," UNIX Programmer's Manual, Vol. 9, April 1979, pp. 255-265.



A Software Product Management System

Why do we need to manage Software?

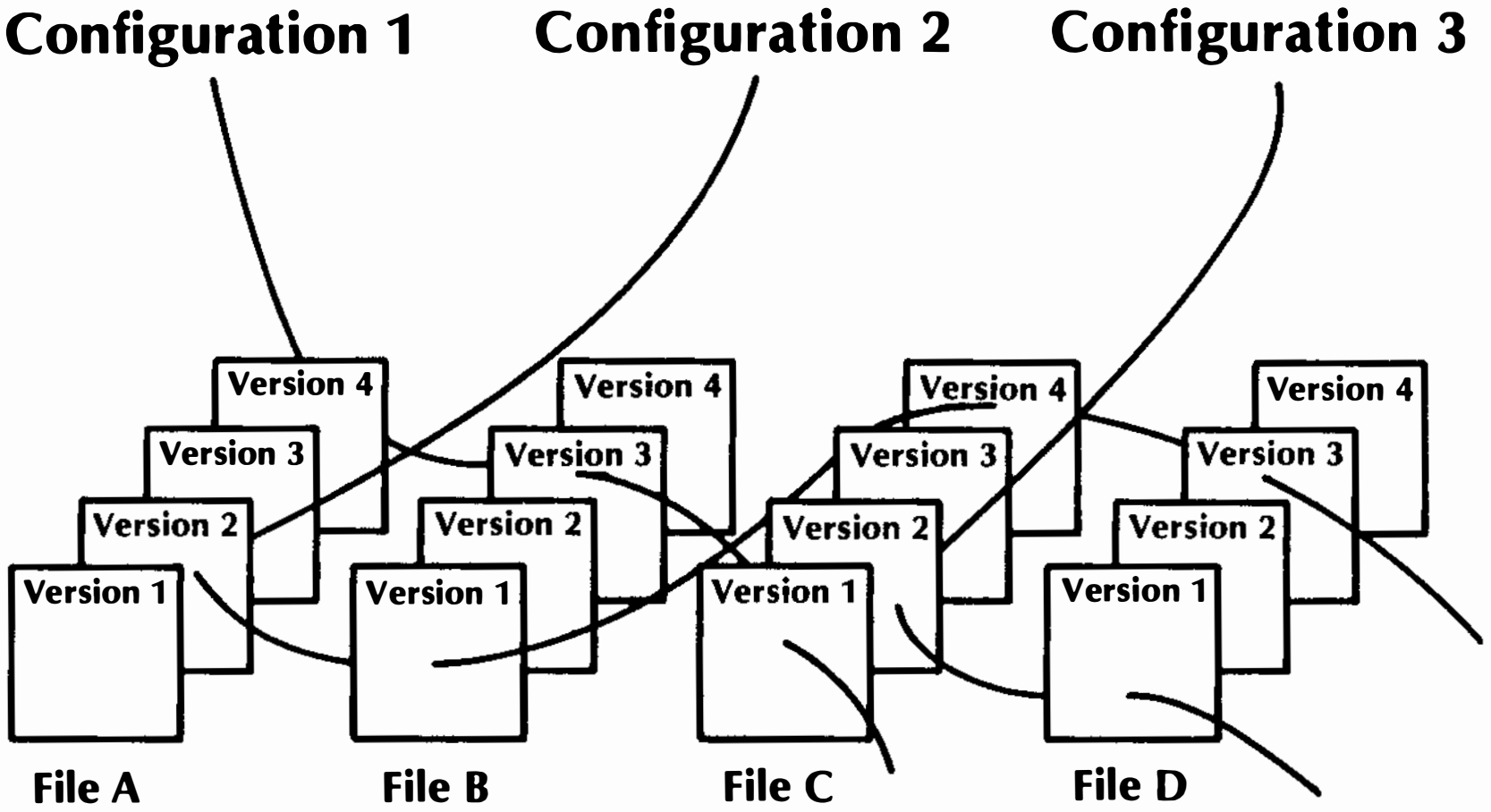
- To track all changes made to files
- To improve tracking of software product configurations
- To improve tracking of software problem reports
- To improve the software manufacturing process
- To assure data base integrity



FPS Software Product Proliferation

Specific FPS Problems

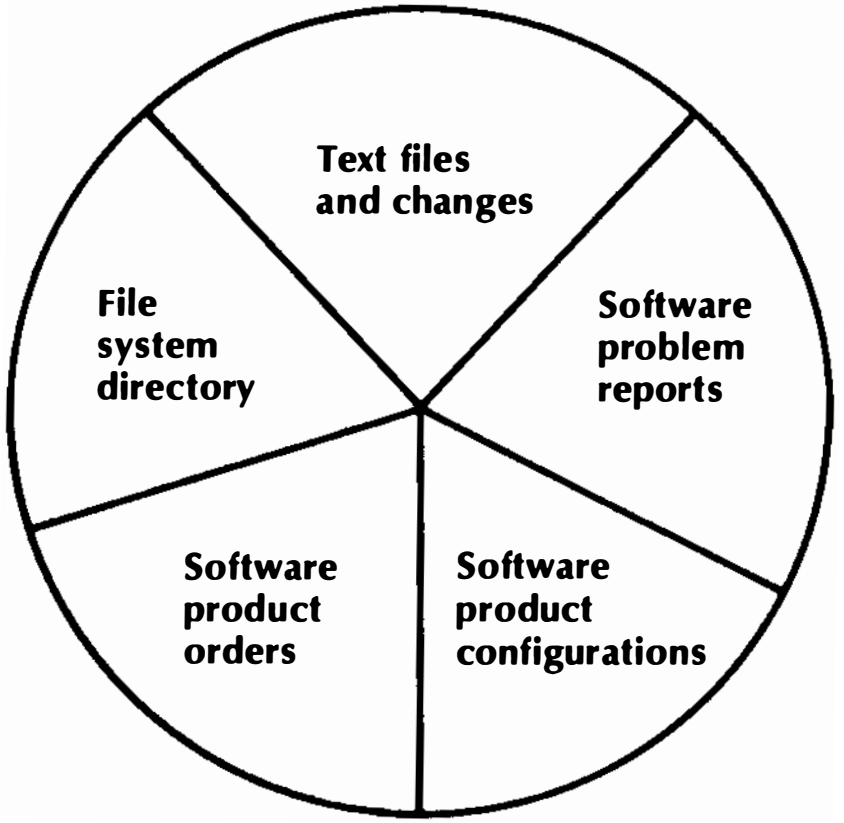
- Multiple host computers for every software product
- Uncontrolled software products
- No traceability of changes to programs
- Software problems not fixed reliably
- Unreliable software manufacturing process
- Large disk space requirements for software products
- Lots of paper work



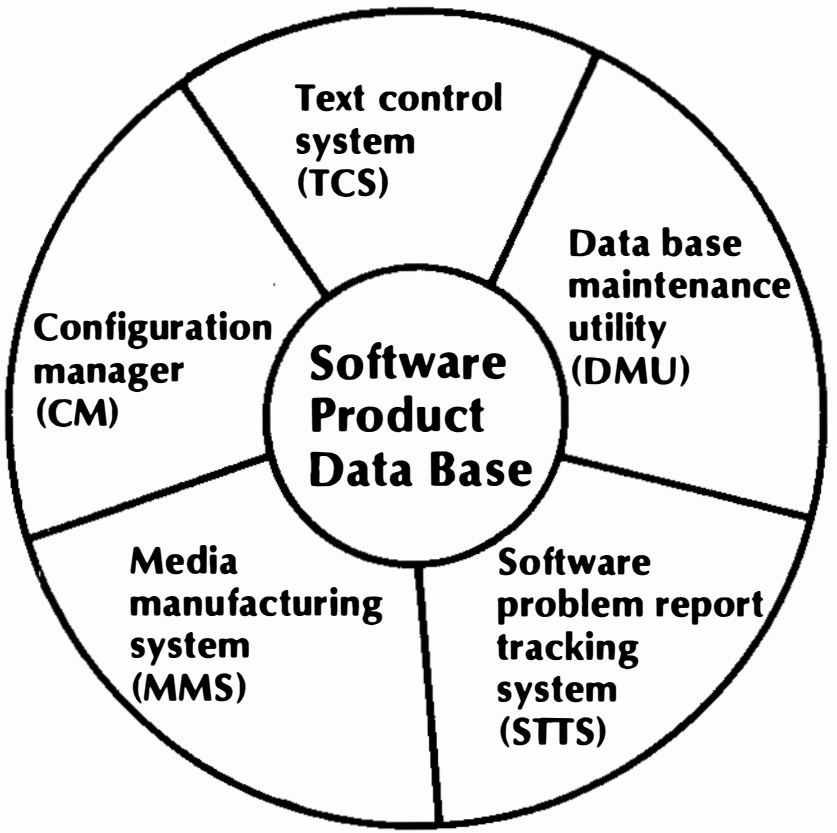
Configuration Management

MUTATE System Goals

- Integration of tools
- System portability
- Consistent user interface for all tools
- Elimination or reduction of paper work
- Automate manual operations to eliminate errors
- Ability to move product data bases to different hosts

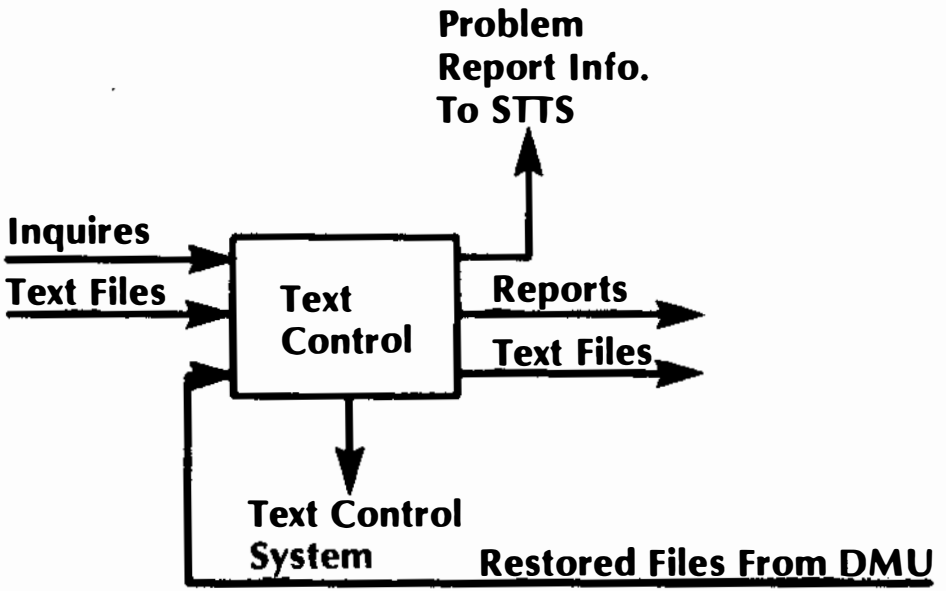


MUTATE Software Product Data Base



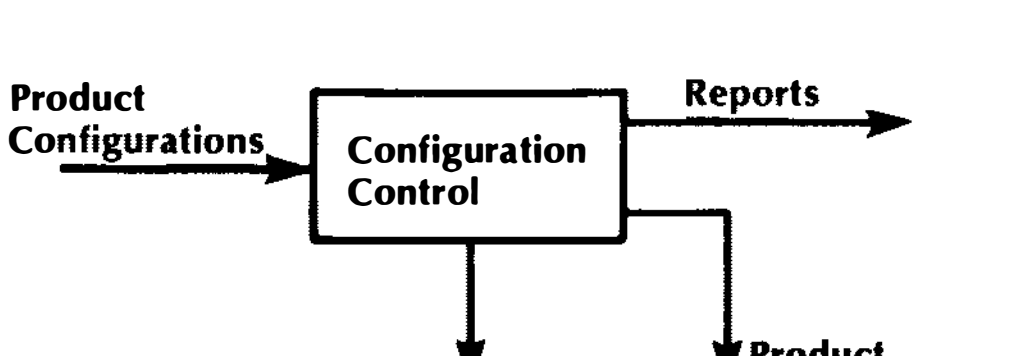
MUTATE System Tools

Text Control System (TCS)



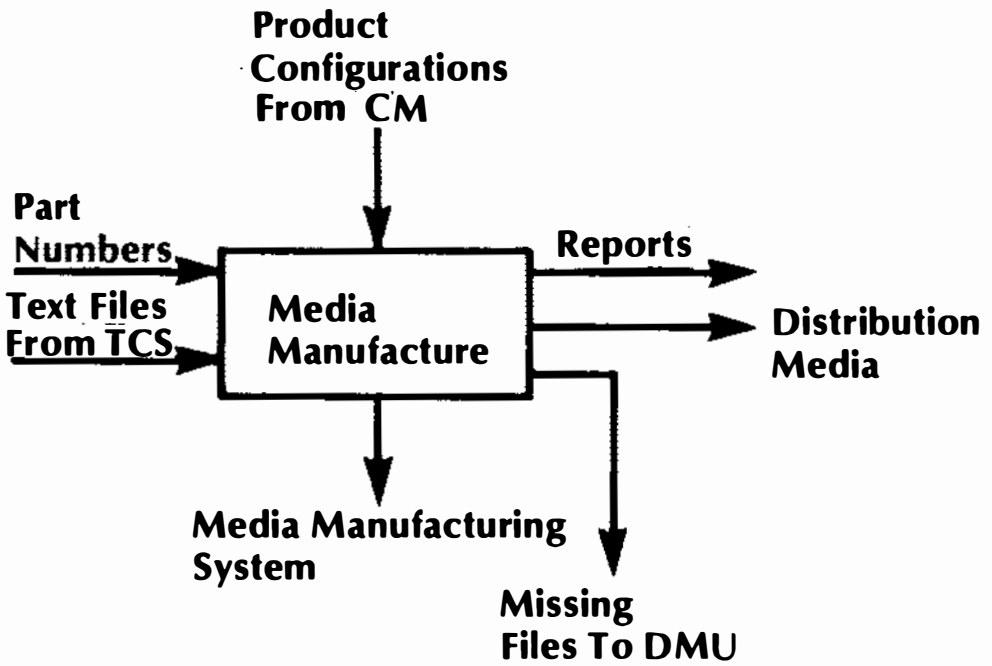
- Retrieves/inserts all files to data base
- Maintains a history of all changes
- Can reconstruct any version
- Provides information about files in data base

Configuration Manager (CM)



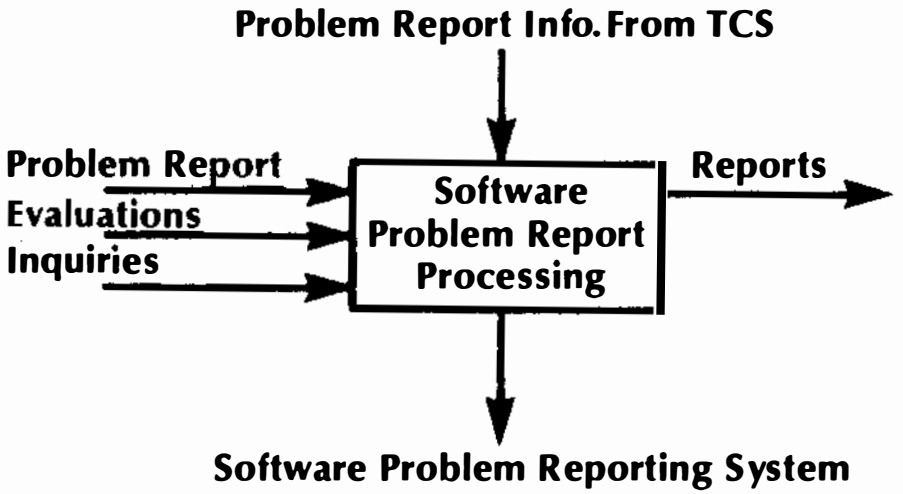
- Creates new software product configurations
- Tracks all software product configurations
- Maintains software product manufacturing parameters

Media Manufacturing System (MMS)



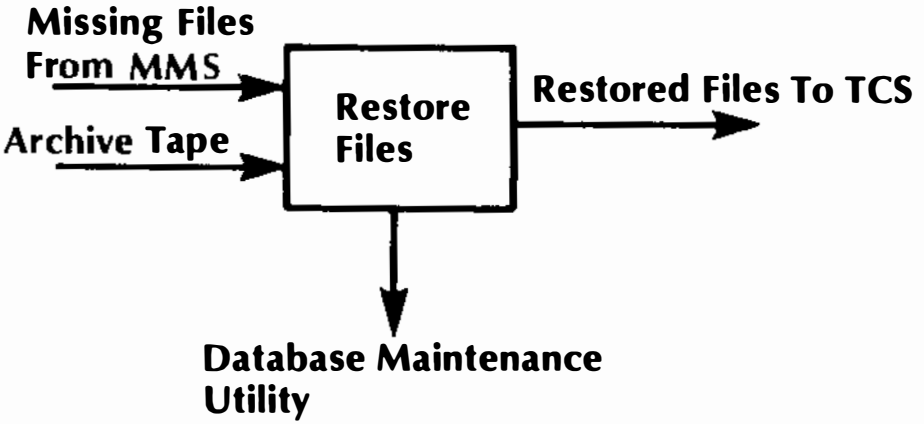
- Processes software product orders
- Uses product configurations to retrieve files out of data base
- Manufactures software product media for shipment to customers

Software Problem Reporting System (STTS)



- Tracks all software problem reports
- Generates software change control board reports
- On line status information about software problem reports

Data Base Maintenance Utility (DMU)



- Manages disk space via archives/restores
- Archives/restores on a product or least used file basis
- Verifies data base integrity
- Moves data bases from one host to another

MUTATE Design Decisions

- Support own file system structure using ISAM
- Use of RATFOR as development language
- Isolated host-dependent routines for ease of porting
- Standardized user interface for all tools
- Fast and reliable system

Experiences during Development and Implementation

- Little user involvement during MUTATE design
- Provided training, user manuals, and hands-on demo
- Acceptance of MUTATE was mixed
- People used the system in different ways
- Received lots of good suggestions for improvements

Benefits

- Complete change control
- Guaranteed data base integrity
- Engineers freed for more development
- Software problems corrected more accurately

Shortcomings

- Overhead associated with retrieving/inserting files to data base
- Uses more computer resources
- Need data base administrators
- Need engineers supporting MUTATE

BIOGRAPHY

Monika Hunscher is the Software Methods, Standards, and Quality Assurance Manager for the Engineering Department at Floating Point Systems, Inc. She oversaw the design and implementation of MUTATE, the software product management system described herein. Her other responsibilities cover the management of all software product data bases, the monitoring of software development activities, and the performance of independent software product testing activities.

She earned her B.S. degree in Computer Science at The Cleveland State University in Ohio, and received her M.S. degree in Computer and Information Science from Case Western Reserve University, also in Cleveland.

"Yellow-Tag" Software Project Management

or

How The 3-M Company Is Contributing To Software Quality

David T. Cassing
Firmware Engineering Manager
Graphics Systems Products
Tektronix, Inc.

"YELLOW-TAG" SOFTWARE PROJECT MANAGEMENT

or

HOW THE 3-M COMPANY IS CONTRIBUTING TO SOFTWARE QUALITY

David T. Cassing
Firmware Engineering Manager
Graphics Systems Products
Tektronix, Inc.

ABSTRACT

As systems get more complex and the pressures mount to get a product to market, software project leaders and managers need techniques that can help control and manage a project in this environment. This paper deals with a variety of "informally formal" techniques developed and used by the team that developed the code in the 4115B color raster terminal. Keeping a project within certain bounds of control can be done without "stifling creativity." A quality software product requires that interfaces between modules and "super-modules" be defined and controlled. On a large project, these technical interfaces are frequently reflected in interfaces between people and groups. The sequencing of events also becomes quite critical to being able to develop the software in a timely manner since a quick "time to market" usually requires many activities proceeding in parallel. This paper discusses the contribution to quality that is made by recognizing the difficulties imposed by such a "complex" environment. It offers a few simple survival methods such as using "those little yellow tags" to make developing a task precedence chart as painless as possible and proposes some guidelines on how to ask the right questions before it's too late. Emphasis is on simplicity and flexibility.

INTRODUCTION

The basic theme of this paper is that software quality rarely (if ever) results from projects that are out of control. There are a variety of elements that must come together in

order for a group of people to produce a body of quality software. This paper will mention some of these, in passing, such as requirements documents, "up front" specifications, design reviews, code reviews, testing planned for at design-time and others. The point, really, is that these things take time and time is available only when planned for and when the project is "under control." Further, a project can be kept "under control" without using elaborate mechanisms. The mark of a good system is one that stresses simplicity and flexibility, nurtures creativity and quality, and keeps the project on budget and on schedule - "Yellow-tag Management."

The advent of microprocessors and relatively inexpensive memory and computing has given rise to many large and complex software systems. There is typically a requirement to get the work done as quickly as possible in order to get it to the market and to the customer. On such a project, there are typically several (many) people involved and there are many things changing at once. In today's organizations, there are also many disciplines involved and numerous activities that all demand time. The information contained here is intended to give the reader an understanding of an approach for managing a software development effort that is "informally formal."

DEFINITIONS

Before continuing, there are a few terms and concepts that need to be defined as they will be used in this paper. These will be defined from the pragmatic perspective.

SOFTWARE - Certainly, this is a sequence of computer instructions in a computer language that directs/controls the computer and its peripherals. "Firmware," "microcode" and "JCL" are just other names for "software." "Software" also includes text of the code that is its form, the design concepts embodied in its implementation and implies an understanding of a problem to be solved. It is now possible to state the next definition.

SOFTWARE QUALITY - This is the degree to which software approaches excellence. Software of high quality works according to its specification when executed by a computer such that no (or minor or few) problems are encountered during its use. People who use it find that it solves a problem. Further, it is easily supportable by those who must modify it, extend it and build upon it. This usually implies that it is

cleanly written, commented internally and documented externally. Finally, it is exactly what the producer thinks it is. This implies proper configuration management mechanisms and attention to related details of backup and access.

It should be pointed out that "high quality" does not require perfection. There is certainly nothing wrong with striving for perfection as a superordinate goal, but this must be balanced against the other stated objectives such as those of the business or the need for a timely delivery. In any event, the quality level cannot be assessed without a metric of some sort. The most common measure of software quality seems to be bugs found after delivery. The tracking of this data will at least contribute to an incremental improvement in an existing software system. It is also common to track and chart problems in software for a period of time preceeding its general release. To make this meaningful, of course, test suites must be developed and used repetitively. Some techniques are available that provide for using this data as input to a computer model that can do some "predicting" of remaining problems based on current trends and historical data from similar systems. These methods are still in their infancy and only address one dimension of software quality, but there may be some hope in such an approach.

CONTROL - In the context of "controlling a project" this term really means management and direction. A project that is under "control" is one that is proceeding according to plan and most of the energy is going toward accomplishing the objective. Creativity is a tremendously important ingredient in generating quality software. A software project can be under control without stifling creativity.

COMMUNICATION - As applied to people (e.g. software designers) this term means that (at least) two thinking beings have a common, accurate understanding of the meaning of a message just sent from one to another. A phrase spoken is not communicated until it is understood by a receiver; neither is an objective communicated until it is understood by a doer or the interface to a computer subroutine communicated until it is understood by someone who must implement it or use it. Software systems cannot function without the subsystems communicating; these same systems cannot be built without its builders communicating. Communication is so important that it warrants some extra checking and effort to insure that it is taking place. This is what "Yellow-tag Management" is all about -- communication.

INFORMALLY FORMAL - This is a term to describe the notion that formality and rigor of thought (and action) do not have to be

embodied in elaborate mechanisms and forms. For example, a software system that is elegantly described with charts and "pretty" documentation is of no value if it is not correct or if it does not meet the stated requirement. The same idea applies to planning and scheduling a software project. A chart on a table with a bunch of yellow tags and pencil lines (a working document) that is accurate is more useful than an accurately plotted representation of an approach that will not work. Further, a plan representation (drawing or chart) that is difficult to change will usually not get changed. This leads to an official-looking document that is wrong. Such is often worse than no representation at all. (Note: While most of the techniques to be described in this paper have been largely manual, the author would like to note that the combination of interactive computer graphics, low cost graphic hardcopy devices and newly available software is beginning to make more automated methods available to project managers. The techniques and the thought processes behind "yellow-tag management" remain valid.)

CAUSES OF LOWER QUALITY

It is rare to find someone who says he is not in favor of a quality product. Even in the most intensive "schedule-driven" environments people want a quality product. What happens, then, to cause quality problems? The quality failures can be divided into the following four categories:

1. Incompetence. Simply stated, this means quality was understood and attempted but it could not be accomplished. This is probably the rarest cause of poor quality software.

2. Poor definition. In this case, the software implementors thought they were producing good quality software but, in reality, they were not. The difference between this category and the previous one is that in this case, the implementors did not understand the definition of the quality level that was required.

3. Conflicting objectives. In this case, lower quality results when other objectives are given higher priority. Life (and business) is full of compromise and this is another one. An overt decision to lower

product quality to attain some other objective is not bad in itself. It is possible to "lower" quality but still have a high quality product. (Remember, high quality does not require perfection.) The important thing to get right in this category is to make sure that the total costs and implications are understood when a decision is made to lower quality. There is no lurking evil in this category, it is simply a tradeoff. If the judgement of the value of a particular quality level is incorrect, then a quality failure results.

4. Carelessness. This is probably the most common reason for quality failure. When people become tired and get in a hurry they make mistakes. These mistakes cause problems. If a project is well-run and "in control" then fewer mistakes will be made. This raises the quality of a software system.

A well-run project, then, must endeavor to insure that none of these four conditions occur to a degree that the software project fails due to poor quality. Incompetence must be recognized and confronted. People must become educated to understand the definition of a good quality objective. When objectives conflict with quality then any decisions must be made in light of a total understanding of the cost of poor quality. Finally, a software project must be well-planned, scheduled and staffed in order to avoid errors that occur from problems that crop up at the end of a project that could have been found earlier and planned for.

SOME CONVICTIONS

There are some convictions that the author has and that have given rise to the methods described in this paper. These follow:

1. For a particular organization, there is a maximum time span that a project can last. When this length is exceeded then people become tired and/or restless and leave the project. This generates a lack of continuity in the project and impacts the informal communication network between project members. Beyond this threshhold more formal communication mechanisms must be installed to protect the project

itself and this adds additional time to the project. The author uses eighteen months as the threshhold time length. A project longer than this threshhold must be run differently than one that is shorter. A one-year software project is different than a ten-year NASA project.

2. A major project part that exceeds six people carries increased risk. This is largely due to the increased number of people-to-people communication interfaces that must be managed.

3. A project must reach "critical mass" to succeed. The operational definition of "critical mass" is that sufficient resources and momentum exist in support of the project to allow it to survive a few serious setbacks. Examples would be the loss of a key contributor or the failure of an expected technical approach to develop out as hoped.

4. The most successful projects run on the edge of being in control. High technology businesses such as those surrounding software projects are quite competitive. Calculated risks must be taken in order to get a product to market quickly. An overly conservative approach may produce a quality product but too late; an overly aggressive approach may produce a poor quality product in time. There is a fine line to walk to maintain a balance between aggressiveness and insanity.

5. The most successful projects typically are made up of people and groups who know the overall objective and their role in accomplishing it. Many key decisions are made by those "in the trenches" without seeking approval of someone else on every score. Design and code reviews by peers can totally succeed only if these people understand where the project is headed and how it is supposed to get there.

6. One cannot change a plan if there is no plan. Flexibility is important. On a high technology project things happen such that plans must be changed. It is acceptable to change a plan occasionally. It should be sufficiently simple to do this so that interactions are known and so that the

changes can be quickly and effectively communicated to all others on the project. Managing the interfaces is essential.

7. Small teams are more effective than larger teams. The implication of this conviction is that quality people go a long way toward contributing to project success.

8. The value of having a plan is in forcing people to go through the discipline and mental exercise required to create it. The author has occasionally told an individual that if s/he creates a plan and then throws it away immediately upon completion (so it is seen by no one else) then half of the reason for doing it has been accomplished. The other half, of course, lies in communicating it to the others participating in the project.

THE "YELLOW-TAG" METHOD"

The method is as simple as planning in a way that includes the team members in the process. Focus is on identifying as many of the interfaces between the parts of the project as possible and on recording these interfaces in a form that can be communicated, discussed and easily changed as soon as new data suggests that such is warranted. Since it is desirable to deal with as much detail as possible, a large working area (e.g. piece of paper) is required.

The name "Yellow-tag" comes from the use of those little yellow stickers that have become ubiquitous in today's office. They come in several sizes, are readily available and are easily moved during the initial planning phases or when a change is necessary later. The idea, of course, is that there are some really simple mechanisms that can be employed to build and massage plans and schedules that allow viewing and interactive participation by many. The key to success is that they are kept visible throughout the project. Progress can be tracked by marking off tags with a colored marker. Precedence charts can be built and changed quickly. Tags recording tasks can be laid out in time. If an item needs to move from one person to another it is easy to do. Simple. Flexible. Communicates.

AN APPLICATION

The philosophy of "Yellow-Tag Management" was used by the firmware engineering team that designed and implemented the code for the 4115B high resolution color graphic terminal. Figure 1 describes the flow of a typical (successful) software project. The first two steps do not involve the use of yellow-tags but are essential to success. The objective must be known and communicated to all throughout the life of the project.

The preparation of the Process Definition, Problem Partitioning and Determination of Responsibilities was done using large sheets of paper, writing potential solutions on tags, and seeing how workable the solution was.

The next step was to map out the details of the precedence relationships. The objective of this process is to produce as detailed a view of the problem as possible and to determine the sequencing of the events. For example, certain data structures must be designed before a module that uses that data structure can be designed. The method used was to have the project manager and the project leader produce a reasonable first guess and then bring in each person one by one to add detail to the chart and identify new dependencies that exist. The chart was also explained to each person and s/he was asked to comment on any aspects that seemed improper or out of sequence. Frequently it was determined that something "in the future" was needed sooner than thought at first. These relationships were noted by drawing lines from one tag for an event to another. This, of course, is the same sort of process that one must go through to construct a plan for PERT or CPM and similar automated planning tools. The result of this effort was a large chart of tags and precedence lines. Each tag contained a brief task description (often just a module name) and the name of the individual responsible (as determined to this point). Identifying an individual, by name, with each task is essential. This is an early "reality check."

Any event that took time or produced something needed for the firmware was recorded. This meant that all interfaces with other related projects were also recorded. In the end, most of the early hardware and microcode events were also part of the chart since these were typically necessary events to firmware related activities. This is THE PLAN.

A new (or revised) plan was generated at each major milestone. Each plan attempted to provide great detail from the current point in time until the next milestone and as much as possible

for the remaining parts of the project. Ideally, at each milestone one would review the plan and make no changes except those which add detail. In reality, changes are made as new data surfaces. This is especially true up until the project reaches its Design Completion milestone.

The reader should note that the plan does not deal with the time dimension and only touches on resource allocation. People are assigned tasks prior to (and during) the plan generation in order to make sure someone is covering each item. When estimates and detailed insights are needed then the planners know who should be involved.

The next step is to apply resources and time to the knowledge provided by the plan. To do this, each person was asked to make time and resource estimates for the assigned tasks. It is a fairly simple task to generate a schedule from this data. The first one produced was way off the target. Some people had too much to do and others too little. Also, some tasks were completed too late based on precedence needs. Anyone who has generated a PERT or similar chart is familar with this phenomenon.

Another artifact of this step is that some tasks spanned several weeks resulting in a situation for which no progress (or problems) could be measured for some people for more than one week. This violated one of the planning objectives which was to be able to report accurate status each week for each person. When this occurred, the task was subdivided if possible. The final result was a large chart containing the names of each engineer on the project and the deliverable results for each week. This is THE SCHEDULE. It was kept in plain view of all and (along with the PLAN) were the focus of the project manager's view of the project.

During the final drive, a daily meeting was held to identify and communicate all known problems. Each problem was assigned to someone and recorded on a tag. A priority list of tags was kept and reviewed each day. All project engineers attended and all knew the priorities. Communication was key to keeping the team functioning at maximum efficiency. Problems found in one area sometimes were a sign of something wrong in another. Chasing dead ends had to be minimized.

RESULTS

The plan was executed and changed as necessary. Problems were visible early. Resource shortages were found early and accommodated in order to get the product code tested, evaluated and delivered on time.

GUIDELINES FOR SUCCESS

There are a few simple rules to remember to help insure a successful application of this method.

1. Make the plan visible and understood. Everyone must recognize this as a significant controlling document.

2. Build the first plan without regard to resources and time. This helps focus on the identification of tasks, interfaces and their sequencing. Applying time and resources too early can result in unintentional hiding of necessary information that determines success.

3. Make the schedule visible and keep it that way. Take a hard line on marking off an item as complete. Strive for measurable tasks. For example, "to have a design review" is an improper objective. Rather, the objective should be "to have a SUCCESSFUL design review." This implies that the reviewers have signed off on a particular module. Record this data for future reference.

4. The schedule should account for anything that takes time. This includes vacations, holidays, classes and side projects.

5. Make sure the schedule is sufficiently granular so that the project can be tracked at small enough intervals to give early warning. Brooks notes, "How does a project get to be a year late? ... One day at a time." [1] A project manager cannot bury his head in the sand. Problems must be known and faced.

6. Know the strengths and weaknesses of the plan and the

team. Look for trends and adjust continually. This is one of the prime reasons for keeping the data in a flexible form.

WATCH FOR FAILURE SIGNS

There are a few things to watch for that can nearly guarantee failure.

1. The classical one: "No time to plan." Do not let this get you.

2. GANTT charts (schedules) without a clear understanding of the precedence relationships to a significant level of detail. The value of understanding the interactions between the various tasks, support groups and "who needs what when" cannot be overstressed.

3. More than one person responsible for an item. Shared responsibility rarely works. It results in confusion, at best.

4. 100% of resources committed too early in the project. When involved in fast track or risky technologies one must plan on something going wrong. There is a balance between risk and foolhardiness.

SUMMARY

This paper has dealt with the idea that software quality is directly influenced by how well a project is planned and executed. An effectively managed project is a necessary condition for high quality software. A few informal methods for dealing with planning, scheduling and tracking/controlling a complex software project have been discussed. These have been successfully employed on recently completed software projects. "Yellow-tag management" focuses on participative planning of detailed interfaces and communication of this information to all.

Generalized Software Project Flow

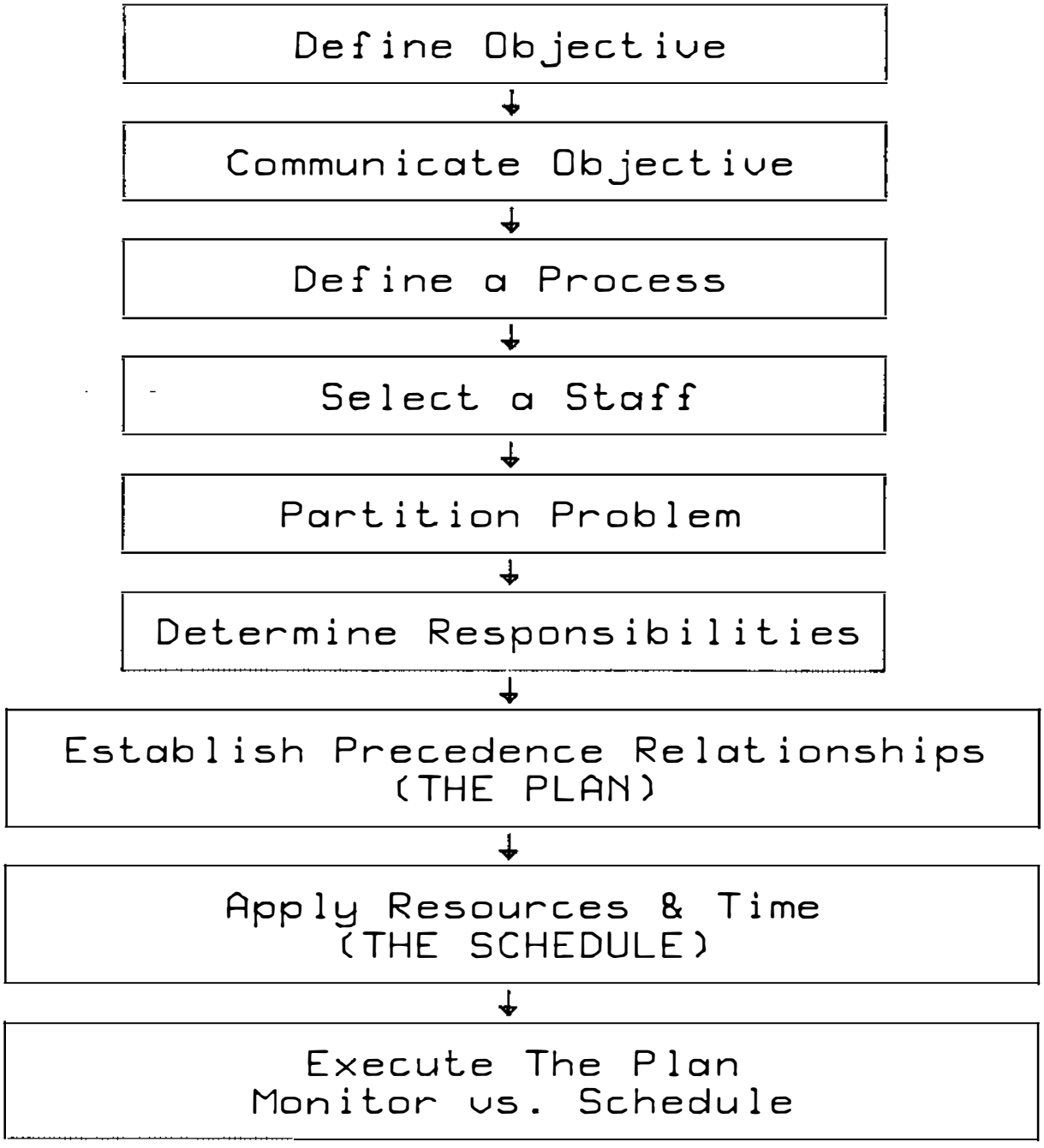


Figure 1.

REFERENCES

1. Brooks, Frederick P., Jr., The Mythical Man-Month Essays on Software Engineering, Addison-Wesley (1972)

INTRODUCTION

- High Quality Software requires project control
- Complexity implications
- Manage Interfaces
- Flexibility, Simplicity
- "Yellow Tags"

DTC
24 Sep 1984

DEFINITIONS

- Software
- Software Quality
- Control
- Communication
- "Informally Formal"

DTC
24 Sep 1984

CAUSES OF QUALITY FAILURE

- Incompetence
- Poor definition
- Conflicting objectives
- Carelesness

DTC
24 Sep 1984

SOME CONVICTIONS

- Maximum time length
- Number of people
- Critical mass
- "Edge of Control"

DTC
24 Sep 1984

MORE CONVICTIONS

- Independent people processes
- Plans and Changed plans
- Small teams
- (Hidden) Value of plan

DTC
24 Sep 1984

THE "YELLOW-TAG" METHOD

- Simple
- Flexible
- Communicates:
  - Interfaces
  - Expectations
  - Visibility

DTC
24 Sep 1984

AN APPLICATION

- Tektronix 4115B project
- Evolved
- Plan at each milestone
  - Revise
  - Add detail
- Final drive "Hot List"

DTC
24 Sep 1984

GUIDELINES FOR SUCCESS

- Visible; Understood
- First: PLAN ...
- ... then SCHEDULE
- If it takes time, put it on the schedule
- Granularity
- Consider strengths/weaknesses

DTC
24 Sep 1984

SUMMARY

- Software Quality requires Project Control
- Emphasize Substance, then Form
- Simplicity
- Flexibility

DTC
24 Sep 1984

BIOGRAPHY

David T. Cassing

David Cassing is an Engineering Manager in the Graphics Systems Products Division at Tektronix. He holds a B.S. in Mathematics and M.S. in Computer Science from Kansas State University. He has been involved with computers since 1965 and with computer graphics since 1972.

Mr. Cassing was the firmware project manager for the Tektronix 4115B high resolution color graphics display terminal. He has been responsible for the development of computer graphics software at the systems and applications levels at Tektronix since 1978. Previously, Mr. Cassing was a development engineer and software engineering manager for six years at Texas Instruments where he worked on the design and implementation of VLSI graphics layout systems.

3 Quality in the Design Phase —Second Session

Speakers and Titles:

Dr. Thomas J. Edwards/Mr. Thomas N. Anderson; Teltone
"Validation as a Part of Software Design"

Mr. Don Zocchi; Tektronix
"Adding Quality at the Analysis and Design Phases"

VALIDATION AS A PART OF SOFTWARE DESIGN

Thomas J. Edwards and Thomas N. Anderson
Teltone Corporation
10801 120th Avenue N.E.
Kirkland, Washington 98033

ABSTRACT

When designing a complex software(/hardware) product a means of validating the software should be developed. Because the product must be revalidated each time the software is updated, the validation procedure should be automated to insure that the necessary validation is carried out. To provide for this automation, one of the engineering tasks is to design test scripts which will stimulate the product in a directed manner to verify a design, with the resulting stimulus responses to be either verified directly or saved in a file for later verification. In this case, the interface between the product and the test script is a system capable of providing the real world stimulus and observations dictated by the test script. When the test scripts and any corresponding stimulus response files are completed for a given release level of a product, automatic validation is then possible by rerunning the tests when alterations are made to the product. This enables the software designer to fully verify a design during unit test while further enabling system wide testing during integration, validation, and final qualification. Furthermore, we have found that engineering design cycle productivity can be greatly enhanced when the validation is initially performed on a computer simulation of the product. In this case, the interface between the product and the test script is a system capable of simulating the real world stimulus and observations. In either case, the test script is the same.

1. VALIDATION PHILOSOPHY

When a complex software(/hardware) product is developed, the resulting product must be thoroughly validated that it functions as defined in its product definition. Furthermore, when fault reports or product enhancements require the product to be updated, it must be thoroughly revalidated before re-releasing to the market place. To reduce the time for validation and to insure that the validation is carried out, an automated validation system is required.

Validation actually occurs within the product development cycle at three levels: engineering design, quality assurance, and manufacturing. The most extensive testing is by quality assurance, followed closely by engineering, and the least amount by manufacturing. An automated validation system should attempt to satisfy all three levels.

The level relevant to this discussion is engineering and in particular software development. The development of software generally occurs in five phases: requirements, design, coding, unit test, and integration test. In many development efforts, validation of software is left to to someone else, to the end, or is an afterthought. However, at the requirements level, with the aid of product definitions and marketing reqruirement documents, a software engineer can specify externally the manner in which the product will function given a set of external stimuli and and conditions. And it is at this phase that the software engineer could specify a set of tests that would thoroughly verify the resulting design. These tests could then be applied to the engineering unit test(s), and integration test. By writing the tests in a test language suitable for an automated validation system, the software engineer could run the tests as often as required to produce software which matches the product definition. And with automation, testing would not be an afterthought or a burden, but could become part of the software engineering design.

A consequence of making the testing a part of design is that quality assurance is given a nearly complete set of test scripts upon which to qualify the product. As each engineer is independently providing a part of the system, the end result is a cumulative set of test scripts which cover the entire product. Of course, most of these test scripts will be limited to (1) testing the product with external stimuli as specified in a user's guide, and (2) testing the product with external stimuli which deviate from normal operation (gorilla tests). And the scope of these tests will be probably cover only a single feature in isolation. A third level of tests needs to be generated. This level will test the interaction of features, either serially or in parallel to provide a system wide suite of tests. With these three levels of testing completed, a fully automated qualification of the product can begin.

The benefits gained by making automated test development part of the design cycle should be many fold. The definition of unit tests and integration tests should better define the software design. Automating the unit tests should provide a very valuable debug tool which can later be used for integration testing. The resulting set of engineeriung tests should become a major part of the system wide qualification designs. And during system debugging, existing or new test scripts may be used to debug software in a directed and controlled manner. Furthermore, when the bug is fixed, not only can the single 'fix' be verified for correct operation, the revalidation of the entire system will be possible to insure that the fix has not affected the remainder of the system. And perhaps most importantly, a software engineer may perceive proper testing as a design aide, not a design burden.

2. INTEGRATED SOFTWARE DESIGN AND TEST CONCEPT

Our concept is to provide an integrated environment for all phases of software development and testing. The top level system architecture is depicted in Figure 1. The figure is a simplified data flow diagram with dotted lines partitioning the components into their respective physical entities. Each of these will be discussed in the following paragraphs.

2.1 Host

The host computer is the hub of the software development and test environment. It provides the resources for:

[1] Source Code Editing.

[2] Source Code Control.

[3] Compiling/Assembling.

[4] Linking.

[5] Simulation Execution and Control.

[6] File Download/Upload to the Test and Validation System (TVS) and Target System.

The goal is to provide a software environment that allows the software engineer to remain on the development host as far into the developmental process as possible. This will provide access to software development tools that may be lacking on the target system. To reach this goal, scope of the host is extended beyond the usual boundries with the addition of a simulator capable of executing and exercising the target system software. The boundry is further extended by allowing the test scripts that drive the simulator to also drive the test and validation system (TVS) which in turn drives the target system.

The simulator provides the environment for the initial testing of the target software. The tools of the host (symbolic debugger, regular expression searcher, etc) assist in this effort. The user specifies the stimuli to the target software (through the simulator) by developing a test script.

When the target system software is ready to install in the target system, the host provides the means to download the necessary target software to the target system and the test scripts to the test and validation system. The same test scripts that were used to drive the simulator can now be used to drive the target system. The result of running test scripts agsinst the target system, the test log file, is transferred back to the host where it can be reviewed and

compared to the test log file from the simulator run.

2.2 Test Scripts

Means must be provided to specify the target system stimulus and response necessary to verify correct system design and implementation. A test language appropriate to the nature of the target system must be devised. The Test Script is used as input to both the Simulator and the Test and Validation System (TVS) at various stages of development.

The test language may be a commercially available language with a special library of subroutines developed that are applicable to the user's target system. In choosing a language, ease of installation on both the host and the TVS should be considered.

2.3 Simulator

The Simulator wraps around the target system software and simulates the hardware of the target system. It also provides an interface to the Test Scripts which will specify the Target Software excitation. Figure 1 provides most of the data flow paths for the development environment. User interaction is not depicted but is generally significant. The simulator is generally executed under the control of a symbolic debugger enabling the user to interactively control the flow of execution and examine the state of target system software data structures. This satisfies the goal of providing maximum opportunity to debug the target software on the host where the best debugging tools exist and a multiuser environment is supported. Target Systems are generally a very limited resource that must be used efficiently. It is advantageous to do as much debugging as possible on the Host.

2.4 Test Log

Both the Simulator and the Test and Validation System provide output to a file describing the results of the test being performed. This file is referred to as the Test Log. The Test Log can include:

[1] Error Messages.

[2] Informational Messages from the Test Script.

[3] Debug Information.

[4] Test Results

The Simulator and the TVS are designed to use the same formats for output to the Test Log. This uniformity allows direct comparison of the results of test execution under either environment.

2.5 Target System

The approach we are suggesting is directed towards applications in which the target system is a

hardware/software product that exists separate from the software developmental system (Host). In this case, means must be provided to transfer the target software from the host to the target system for test and debug. This may require special bootstrap software to be installed in the target system, but this is generally preferrable to requiring manual intervention in the software installation process.

It also may be advantageous to design the architecture of the target system software so that a single process or module interacts with the hardware. This facilitates the design of the simulator, which can then replace this hardware/software interface module.

2.6 Test and Validation System

The test and validation system (TVS) is designed to interface to the target system to provide the same real world excitation and measurement as would be encountered in the target system's intended environment. The TVS is controlled by a test script downloaded from the host. The test script should execute on the TVS in the same manner as on the simulator and the output (test log) should be directly comparable, within a reasonable time window, to the test log from the simulator run.

The principal design of the TVS will be to accommodate the interfaces of the target system. In addition, software control of each input and monitoring of each output is required. The complexity of this task varies greatly with the nature of the target system.

3. IMPLEMENTATION

We have sought to realize the above concept in support of a digital Private Automatic Branch Exchange (PABX) project. A brief description will be given of the components of our implementation. The overall system hardware architecture is depicted in Figure 2.

3.1 Host

The Host is a VAX 11/750 running Berkeley UNIX. The target system software is written in C. This is a synergistic combination in that UNIX provides a rich set of tools in support of C. The target software can be executed on the host (under control of the simulator) without the need for emulators. The debugging tools of the host can aid in the debugging effort.

The link between the host and the TVS is a standard asynchronous serial line (RS232) with a protocol to insure the integrity of file transfers. The link between the host

and the target system is also a standard asynchronous line with a file transfer protocol. It is supported at the PABX side by a PROM based bootstrap which participates in the transfer and initiates realtime execution of the target

3.2 Target System (PABX)

The target system in our application is a digital PABX (Cascade 400, by Teltone). The systems interfaces and their dimensions are as follows:

[1] 400 Stations

[2] 64 Trunks

[3] 2 Consoles

[4] 1 Serial Maintenance Port

[5] 1 Serial Call Detail Recording Port

Other interfaces do exist (paging, fault display, status lamps, etc.), but will not be discussed for brevity sake.

Our goal is to exercise the above interfaces to validate our design and implementation. In this validation, we must perform sequences of events and observations that would exercise the features of the system. The system features range from the very fundamental (like getting dial tone when you take the receiver off hook) to the very complex (like automatic route selection). These features are largely a function of software and to test them is to test the target software.

3.3 Test and Validation System

The TVS is controlled by a Hewlett Packard Series 200 computer running HP BASIC. This choice was motivated by the desire to use off the shelf instruments where ever possible and tie them to the controlling computer via the IEEE-488 bus (HPIB). The serial links to both the host and the PABX use standard RS232 interfaces available from HP. The in-house designed sub-assemblies (switch matrix, loop holders, console interfaces, digit sender, digit receiver, etc.) are all controlled via standard parallel digital I/O cards available from HP.

The TVS software consists of a menu driven control program with a library of subroutines to be used in writing test scripts. The test scripts are written in HP BASIC but consist primarily of a series of calls to the library routines. The choice of HP BASIC was made because nothing resembling UNIX/C was available for the Series 200 computer at that time. This was unfortunate, but unavoidable. It means that at present we must convert our simulator test scripts from C to HP BASIC before they can be downloaded and

executed on the TVS. This process is partially automated and since the simulator and TVS libraries are functionally identical, the conversion is not as painful as it might seem. This situation will be remedied as we work towards

available for the Series 200 family. This conversion will greatly aid in tightening the coupling between the Host and the TVS.

The test function library consists of subroutines that enable one to perform telephonic excitation or observation. They provide the means to manipulate each of the interfaces in a manner similar to what would occur in the real world. For example, in support of stations, subroutines are provided to (1) take a station off hook, (2) put a station on hook, (3) dial a digit, (4) listen for call progress tones, et cetra. The test script is made up largely of calls to these subroutines to perform the desired sequence of events and observations.

In order to facilitate convenient reference to the target system ports without explicit knowledge of its configuration, a logical numbering scheme is employed. Logical numbers, by definition, start at one and go up to the maximum number of that particular type of resource. The classes of logical numbers are 1) logical station numbers (lsn), 2) logical trunk numbers (ltn) and 3) logical console numbers (lcn). The mapping between the logical numbers and the physical equipment is specified by the user in a configuration file. This enables the user to explicitly control the extent of the hardware being tested.

The general philosophy of the library routines is to require the user to specify the expected results either implicitly or in the calling parameters. If any unexpected result occurs, error messages are automatically generated. For example, if the test script calls the routine to listen for dial tone, implicit in that call is the fact that dial tone is expected at that port. If it is not detected, an error message will be logged in the test log file. The advantage of this approach is that the test scripts are very straight forward and linear.

3.4 Simulator

The implementation of the simulator was facilitated by the fact that the software architecture of the target system was such that all interaction with the hardware occurred through a single software process. This process would communicate with the rest of the target system software by sending or receiving interprocess messages to or from the other processes. The simulator takes the place of this process (and the hardware) by generating and receiving the interprocess messages appropriate for a given scenario under test.

Input to the simulator comes from the test script. On the TVS the test script would command some telephonic event to occur (e.g. go offhook or dial a digit). On the simulator, the test script would cause an interprocess message to be generated that would represent the occurance of that same event. The simulator maintains data structures representing the state of the hardware interfaces. As interprocess

messages are issued from the target software to change the state of a port or interface, the simulator updates the appropriate data structure. Observational requests from the test script (e.g. listen for dial tone) causes the simulator to query these data structures to determine if the proper state does exist.

The simulator can also be run under the control of a symbolic debugger to better facilitate the analysis of the program execution.

4.

RESULTS AND CONCLUSIONS

Automated validation has had a pervasive and positive effect on product development at Teltone. New software can now be very thoroughly verified within a matter of minutes, hours, or days, depending on the tests, as compared with hours, days, and weeks without automated testing; or even worse, thorough testing would not occur. We can also now perform tests that would be impossible to generate manually, but are quite simple to generate through automated means. This latter facility allows the product to be tested under more severe conditions than may be encountered in the real world.

Another positive result is that the engineering staff can generate a majority of the test scripts required by quality assurance and manufacturing. The quality assurance staff can now dedicate more time in developing more complex test scripts for product qualification.

We have found that with a project of several software engineers, tight design schedules, and very limited access to a single automated tester, a simulator is indispensible. Without a simulator, the software engineers were not highly motivated to write the test scripts because of foreigness of the environment and very limited access (the simulator can be run from home over a modem link). Most of our early test scripts were therefore written by engineering personnel working on the validation system. With the completion of the simulator, design productivity has increased several fold. The simulator has provided an on-line non-realtime simulation unit and integration test capabilities to each software engineer in a familiar environment with the resulting test scripts available for real-time validation of the product

Even with a simulator on the host, we have found at times that a single TVS/target system to be insufficient. The architecture of the system, however, easily accommodates multiple TVS/target system stations supported from a single host.

An additional value for automated testing is that a software system can more easily be ported without fear of being unable to quickly resolve complex software errors. We have

actually translated a very large and complex system written in one language into another language. As we had already developed a large set of test scripts which characterized the original system, the resulting system was more readily debugged through the aide of the automated tests. Without automated testing, the language translation would probably not have been attempted.

Another benefit of automated testing is in debugging system wide software faults. When a fault is reported, test scripts are generated which attempt to recreate the condition leading to the fault. The script can be modified an re-run as many times as required to duplicate the fault. The test can now be run in a directed manner to uncover and fix the fault. The resulting test script, of course, is added to the repertoire of system wide test scripts to catch the reappearance of the fault at a later date. This type of debugging provides a much more thorough and complete testing than would otherwise be possible without automated test capabilities.

And perhaps the most positive result is that validation has become a software design aide. The integrated design/test environment has increased productivity while at the same time provided a very complete validation of the product. The result is a quality product which will continue to maintain its quality throughout the development cycle.

FIGURE 1. INTEGRATED SOFTWARE DESIGN AND TEST ARCHITECTURE

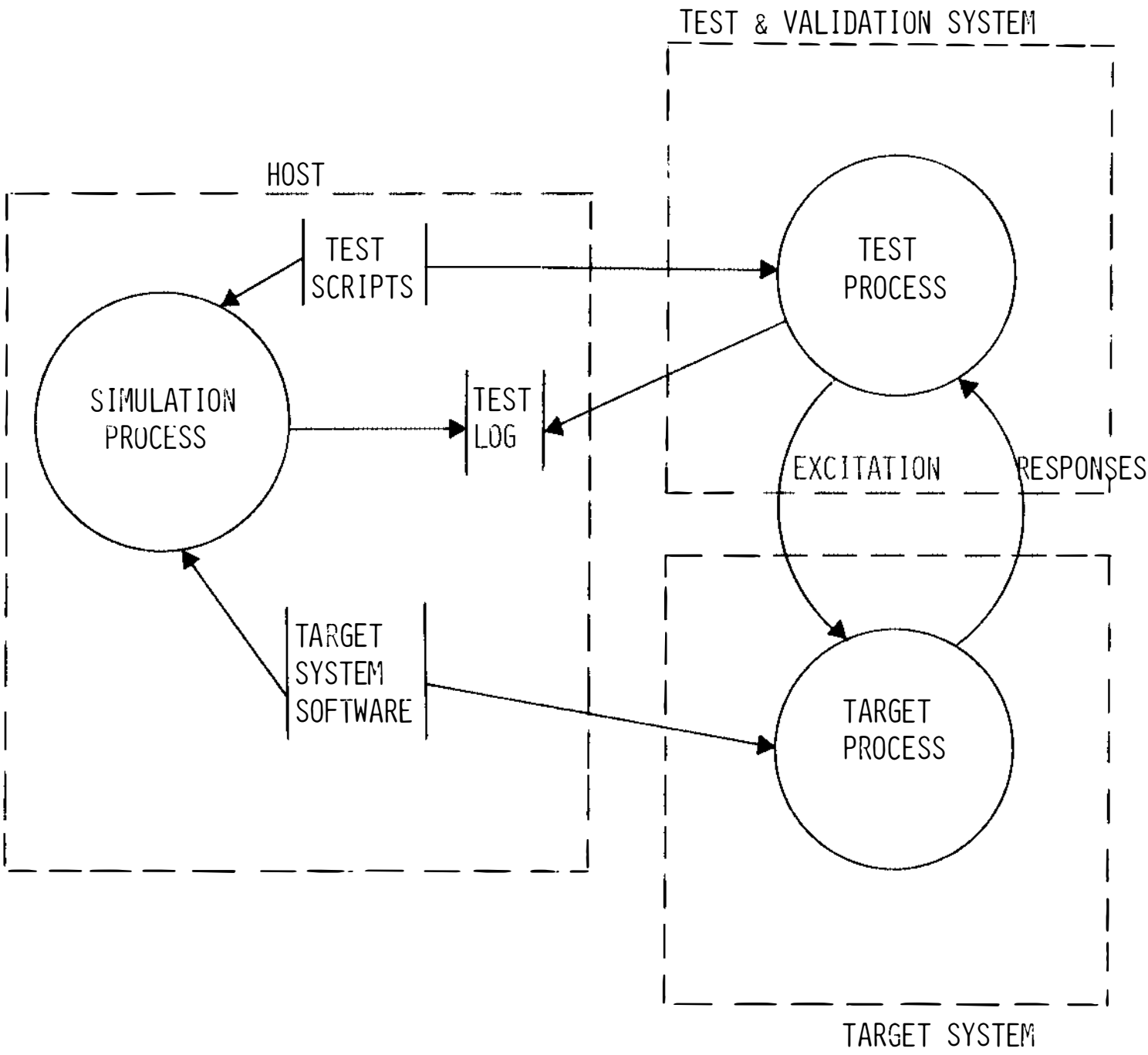
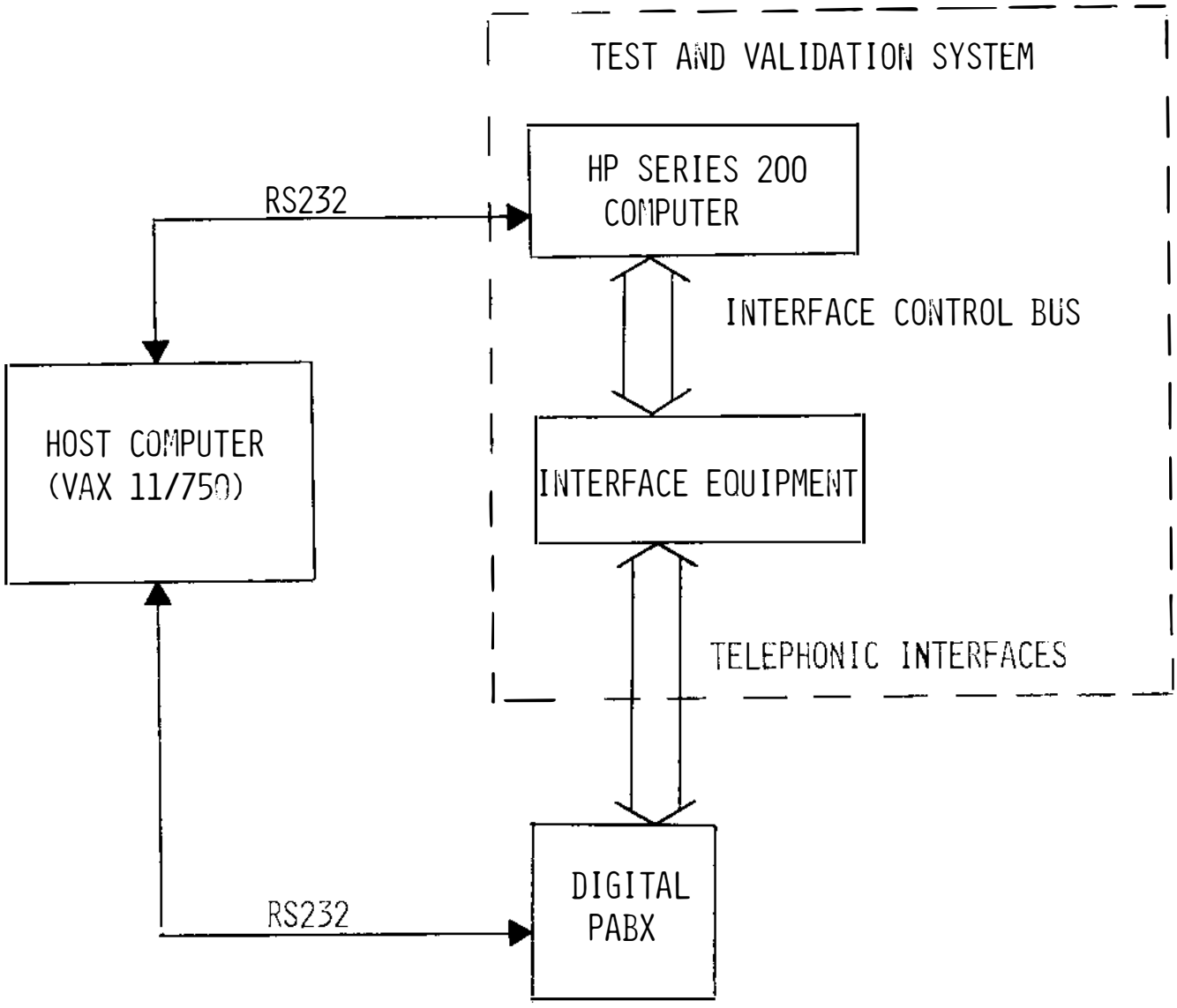
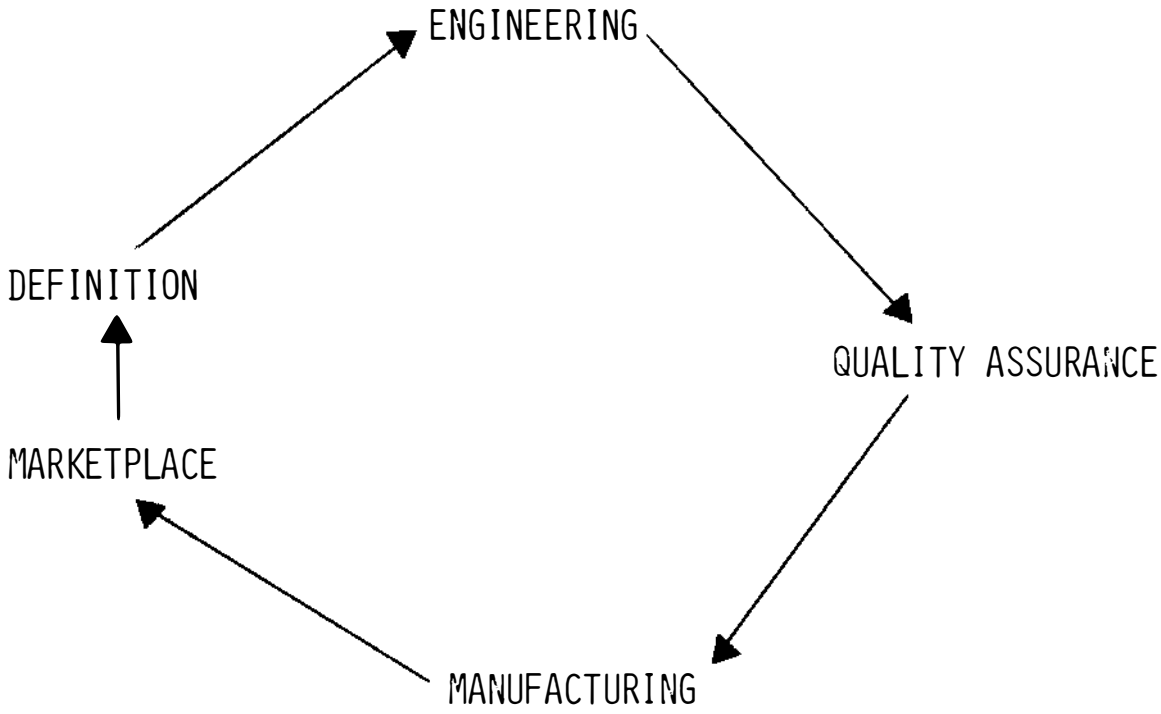


FIGURE 2. INTEGRATED SOFTWARE DESIGN AND TEST HARDWARE ARCHITECTURE




ENGINEERING
QUALITY ASSURANCE
MANUFACTURING
MARKETPLACE
DEFINITION

- REQUIREMENTS
- DESIGN
- CODING
- UNIT TEST
- INTEGRATION TEST

- TEST DESIGN LIMITS
- TEST INDIVIDUAL FEATURES PER A TYPICAL USER'S GUIDE
- TEST INDIVIDUAL FEATURES WITH ERRONEOUS OR NON-STANDARD INTEGRATION
- TEST FEATURES IN SERIAL AND PARALLEL
- REAL-WORLD TESTS

- DESIGN
- DEBUGGING
- PRODUCT QUALIFICATION
- PRODUCTIVITY

INTEGRATED SOFTWARE DESIGN AND VALIDATION METHODOLOGY

- THREE HARDWARE ENTITIES INVOLVED
  - HOST COMPUTER
  - TARGET SYSTEM
  - TEST AND VALIDATION SYSTEM
- ALL ARE COUPLED TOGETHER TO FORM AN INTEGRATED ENVIRONMENT

HOST COMPUTER

- HUB OF THE DEVELOPMENT ENVIRONMENT
- PROVIDES:
  - SOURCE CODE EDITING
  - SOURCE CODE CONTROL
  - COMPILING ASSEMBLING
  - LINKING
  - SIMULATION EXECUTION AND CONTROL
  - FILE DOWNLOAD/UPLOAD TO TARGET SYSTEM & TVS

SIMULATOR

- PROVIDES MEANS TO EXECUTE TARGET SOFTWARE ON HOST
- DRIVEN BY TEST SCRIPTS
- EXECUTED UNDER CONTROL OF SYMBOLIC DEBUGGER

TEST SCRIPTS

- SPECIFIES EXCITATION OF TARGET SYSTEM
- SPECIFIES RESPONSES OF TARGET SYSTEM
- DRIVES BOTH SIMULATOR AND TVS

TEST LOG

- ERROR MESSAGES
- INFORMATIONAL MESSAGES
- DEBUG INFORMATION
- TEST RESULTS

TARGET SYSTEM

- HARDWARE/SOFTWARE PRODUCT
- DOWNLOAD CAPABILITY
- ARCHITECTURE ADAPTABLE TO RUN UNDER SIMULATOR

TEST AND VALIDATION SYSTEM

- INTERFACES TO TARGET SYSTEM INTERFACES
- PROVIDES REAL WORLD STIMULUS
- DRIVEN BY TEST SCRIPTS
- AUTOMATED TESTING

IMPLEMENTATION

HOST

- VAX 11/750
- BERKELEY UNIX
- SERIAL LINES TO TVS AND TARGET SYSTEM

TARGET SYSTEM

- DIGITAL PABX
- SYSTEM DIMENSIONS AND INTERFACES
  - 400 STATIONS
  - 64 TRUNKS
  - 2 CONSOLES
  - 1 SERIAL PORT

TEST AND VALIDATION SYSTEM

- CONTROLLED BY HP SERIES 200 COMPUTER
- INTERFACE EQUIPMENT LARGELY OFF THE SHELF
- IN-HOUSE DESIGNED INTERFACE EQUIPMENT CONFORMS TO A BUS STANDARD FOR EASE OF MAINTENANCE AND EASE OF EXPANSION
- INTERFACE EQUIPMENT PROVIDES EXCITATION OR MONITORING OF PORTS

TEST SOFTWARE

- HP BASIC OPERATING SYSTEM
- MENU DRIVEN CONTROL SOFTWARE
- TEST LIBRARY TO PROVIDE EXCITATION AND MONITORING FUNCTIONS

- MORE THOROUGH DESIGN
- MORE COMPLETE VALIDATION
- INCREASED PRODUCTIVITY
- MORE DEVELOPMENT OPTIONS

- LEVELS OF VALIDATION
- VALIDATION NOT AVAILABLE OTHERWISE
- ASYMPTOTICALLY APPROACH COMPLETE VALIDATION

- DECREASES TIME FOR UNIT AND INTEGRATION TESTS
- DECREASES TIME FOR INITIAL QUALIFICATION
- DECREASES TIME FOR DEBUGGING
- DECREASES TIME FOR RE-QUALIFICATION
- QUALITY ASSURANCE PROVIDES MORE COMPLEX TESTS
- COOPERATIVE EFFORT AMONG DEPARTMENTS
- DESIGN ACCOMMODATES NEW PRODUCTS

BIOGRAPHICAL SKETCH

Thomas J. Edwards is software manager in the Local Area Communications business unit at Teltone corporation and an Auxiliary professor at the University of Washington in Seattle. His interests include speech recognition by machine, database management systems, and telecommunications. He received a BSEE from the University of Washington in 1970, an MSEE from the University of Southern California in 1973, and PhD from the University of Washington in 1977.

Thomas N. Anderson is the project engineer overseeing the development of an automated test and validation system at Teltone. His interests include speech recognition by machine, speech synthesis, and aides to the blind. One of his recent developments is a very compact diagnostic aide for blind diabetics. He received a BS in Physics from Walla Walla College in 1977.

Adding Quality in the Analysis and Design Phases

Don Zocchi

Tektronix, Inc.

Don Zocchi
Tektronix, Inc.

1. Introduction

The Analysis and Design Phases contain the most leverage in the Product Life Cycle. Output from these phases is:

- refined
- expanded
- built on

in the later phases. The Analysis Phase Output is refined and expanded during Design. The Design Phase Output is refined and expanded during Coding. Since each phase builds on the earlier phases:

> Problems created, or solved early drastically affect product quality later.

2. Product Life Cycle

Software generally goes through the following phases in its lifetime:

- Problem
- Analysis
- Design
- Code and Test
- Evaluation
- Release
- Maintenance (Problem Phase).

3. Analysis

3.1 Objectives

Describe the user's problem, and plan how to solve it.

3.1.1 Problem Description

The Problem Description models the problem to be solved.

3.1.2 Project Plan

The Project Plan includes the resources (cost) and time needed to solve the problem.

3.2 Analysis Involvement

3.2.1 User

The User describes the problem to the analyst. The User also approves the Problem Description and plan.

3.2.2 Analyst

The Analyst models the problem for the designer.

3.2.3 Project Manager

The Project Manager plans the implementation.

3.3 Analysis Roadblocks

Describing the user's problem is not easy. Listed below are some roadblocks that hinder the analysis process.

- Problem Complexity
- Language and Communication Problems
- Misconceptions and Biases
- Wrong people on the problem
- Time
- Budget

3.4 To Improve Analysis Quality

To improve analysis quality, the analysis roadblocks must be overcome, or reduced. The following list suggests general areas for improvement.

- Problem Complexity - improved methods.
- Language and Communication Problems - increase iteration between analyst and user, rigorous models, technical writing courses.

- Misconceptions and Biases - rigorous models.
- Wrong people on the problem - find and keep good people.
- Time - schedule longer analysis phases.
- Budget - attack smaller problems.

People are the key to successful analysis.

4. Design

4.1 Design Objectives

Describe a realistic solution to the user's problem.

4.1.1 Design Specification

The Design Specification is a model of the solution. It describes the software architecture (how the modules are connected), and module descriptions.

4.1.2 Refined Project Plan

The Refined Project Plan describes a work breakdown structure based on the design specification.

4.2 Design Involvement

4.2.1 Designer

The Designer describes the solution for the programmer.

4.2.2 Manager (Opt.)

The Manager refines the plan.

4.3 Design Roadblocks

Once the problem has been modeled new roadblocks are encountered. The design roadblocks contain roadblocks encountered in analysis plus:

- Poor Analysis
- Technology
- New Constraints and Expectations

4.4 To Improve Design Quality

The following list suggests ways to reduce design roadblocks.

- Poor Analysis - improve analysis quality.
- Technology - continuing education for designers.
- New Constraints and Expectations - enter analysis phase again.

5. Summary

The analysis and design phases contain the most leverage in the project cycle. The analysis phase models the user's problem. The design phase derives the problem's solution. Product quality can be improved by reducing analysis and design roadblocks.

Adding Quality in the Analysis and Design Phases

Don Zocchi

Tektronix, Inc.

The Analysis

and

Design Phases

contain the most leverage
in the Product Life Cycle.

Output from earlier phases is

refined

expanded

built on

in the later phases.

The Analysis Phase Output
is refined and expanded
during Design.

The Design Phase Output
is refined and expanded
during Coding.

Since each phase builds on the earlier phases:

Problems created, or solved early drastically affect product quality later.

Product Life Cycle

Problem

Analysis

Design

Code and Test

Evaluation

Release

Maintenance
(Problem Phase)

Analysis Objectives:

Describe the user's problem,

and plan how to solve it.

Analysis Output:

Problem Description that:

models the problem to be solved.

Project Plan that:

describes the time and resources needed to solve the problem.

Analysis Involvement:

User - describe the problem
to the Analyst.

Analyst - model the problem
for the designer.

Manager - plan
the implementation.

Analysis Roadblocks:

Complex Problems.

Language and Communications.

Misconceptions and Biases.

Wrong people on the problem.

Time.

Budget.

To Improve

Quality in Analysis:

Reduce or Remove

Analysis ROADBLOCKS!

Complex Problems - Improve Methods

Language and Communications -
Increase User Interaction

Misconceptions and Biases - use
rigorous models

Wrong People on the Problem - Find
and keep good people

Time - Lengthen Analysis Phases.

Budget - Attack Smaller Problems.

Design Objectives:

Describe a realistic solution to the user's problem.

Design Output:

Design Specification that:

describes the solution shape (how the modules are connected), and module descriptions.

Refined Project Plan that:

has a work breakdown structure based on design specification.

Design Involvement:

Designer - describe the solution for the programmer.

Manager (Opt.) - refine the plan.

Design Roadblocks:

The roadblocks encountered in analysis plus:

Poor Analysis.

Technology.

New constraints and expectations.

To Improve

Quality in Design

Reduce or Remove

Design ROADBLOCKS!

Poor Analysis – improve analysis quality.

Technology Barriers – continuing education helps.

New Constraints and Expectations – Enter analysis phase again.

Summary

Highest leverage.

- Analysis
- Design

Analysis Phase models problem.

Design Phase derives solution.

4 Panel Discussion

Panel Members:

Mr. George D. Tice, Jr.
Ms. Karen Ryer Cunningham
Mr. Michael Garvey
Dr. Michael C. Mulder

MINI-SESSION 4 - Professional Society Panel Discussion

The software community in the Northwest is served by three major professional societies - ACM, DPMA and the IEEE Computer Society. Each panelist will discuss activities of interest to the local software specialist.

Panel Members:

George D. Tice, Jr. (Moderator/Panel Member)

Mr. Tice is a Senior Software Engineer implementing software quality and productivity improvement projects in the Microcomputer Development Product Division of Tektronix, Inc. He is a member of the Tektronix Engineering Activities Council.

As Chairperson of the 2,000 member IEEE Computer Society Software Engineering Standards Subcommittee he is responsible for the development of software engineering standards and the conduct of seminars and workshops that support published IEEE software engineering standards. He was Chairperson of the Task Group that developed the Guide for Software Quality Assurance Plans to support IEEE Std 730-1984, Standard for Software Quality Assurance Plans. He is the 1984 International President of the Society of Reliability Engineers.

Mr. Tice is a CDP with over fifteen years experience in software quality and reliability and data processing including the teaching of San Diego Community Colleges. He has presented software quality related papers at ASQC, IEEE, and SRE conferences and is preparing a software quality textbook for publication by Prentice-Hall.

Mr. Tice received his B.S. from the Pennsylvania State University and his MPA from the San Diego State University. He is a member of AICCP, ASQC, IEEE and SRE.

Karen Ryer Cunningham

Ms. Cunningham received her Bachelor of Science Degree with High Honors in 1976 from the College of William and Mary in Virginia. She majored in Mathematics with a Computer Science option. In 1977, she received a Master of Science degree in Computer Science from Purdue University and completed all course work for her Ph.D. Her research involved Storage Reclamation in high level languages.

In 1979, Ms. Cunningham joined the faculty of Computer Science at Portland State University as an Assistant Professor. During summer breaks, she has done software engineering at Tektronix. Her current research interests include compiler construction and programming language design.

Ms. Cunningham is a member of ACM and the IEEE Computer Society. She is currently Executive Vice President for the Willamette Valley Chapter of the ACM. She is also faculty advisor for the Portland State University student ACM chapter.

Michael Garvey

Mr. Garvey is Vice President and Manager of Systems Research and Development at First Interstate Services Company. He is responsible for planning and coordinating the implementation of major data processing requests made by First Interstate Bank of Oregon.

Mr. Garvey graduated with a BBA in Personnel Management from the University of Oregon in 1968, and spent until 1970 in the U.S. Army Air Defense Artillery. From 1970-1981 he worked for the First Interstate Bank of Oregon, then transferred to his present position at First Interstate Services Company.

Michael C. Mulder

Dr. Mulder is the director of the Applied Research Center at the University of Portland. He is a member of the Board of Directors of the IEEE Computer Society.

5 Case Studies and Techniques

Speakers and Titles:

Mr. Allen Sampson; Tektronix
"The Design of Testable Menu-Based Systems"

Ms. Susan Wechsler; Hewlett-Packard
"Ingredients for a Quality Product"

Mr. Russell R. Sprunger; Graphic Software Systems
"Managing Independent Testing Organizations"

THE DESIGN OF TESTABLE MENU-BASED SYSTEMS

Allen Sampson
Tektronix, Inc.
Beaverton, Oregon 97075

ABSTRACT

Menu displays are gaining popularity as a human interface to electronic instrumentation and to personal computer applications software. The increasing sophistication of instrumentation and applications is resulting in menu-based systems that are complex and difficult to verify. The need for testable menus is becoming critical in ensuring high software quality and feasible software development times. The effect of menu testability on software quality is examined using the menu-based systems of Tektronix logic analyzers as a case study. An overview of the software design and evaluation process is presented that describes methods for the functional specification, functional analysis, and automated testing of menus. Emphasis is placed on detailing those design methods that improve the testability of menus and ultimately the quality of the product software.

INTRODUCTION

Menu-based systems, consisting of menu-driven interfaces and supporting software, are being used in applications ranging from electronic instrumentation to personal computer software. The task of testing menu-based systems is increasing as the underlying applications become more complex. The effect of menu testability on software quality is becoming critical to the design and evaluation of menus.

Logic Analyzers (instruments used to acquire and display digital signals occurring in electronic systems) are one form of electronic instrumentation utilizing a menu-driven interface. Logic analyzers typically use a hierarchy of menus to model multiple levels of user interface. Each menu consists of identifying nomenclature, data entry fields, a screen cursor, and other features that allow the instrument to be set up and operated. A large amount of interdependency exists between the menus and fields of logic analyzers, causing such menu-based systems to be very complex.

Tektronix logic analyzers [1,5] employ menu-based systems that exhibit the menu characteristics described above. The testability of menu-based systems of this type is examined by using the Tektronix DAS 9100 Series Logic Analyzer [2] for a case study. The functional specification, functional analysis, and automated testing of menus are described using the DAS 9100 as an example. Testability requirements are presented with

respect to their effect on the specification, analysis, and automated testing methods of menus. In conclusion, techniques for improving the testability of menus are examined.

Tektronix logic analyzers are used as examples in the methods prescribed here, but the techniques are general enough to be applied to a wide range of menu-based systems. The methods may be used to enhance the testability of menu-based systems and to improve the overall quality of the product software.

FUNCTIONAL SPECIFICATION

A functional specification is a description of a system that identifies and defines the functions the system must accomplish. The primary goals of a functional specification [7] include:

- The specification should provide the designer with all of the information necessary to create a software design for the system.
- The specification should provide the evaluator with all of the information necessary to create functional tests for the system.
- The specification should discuss the program in terms normally used by both the designer and evaluator.

A number of software design and program specification techniques have been developed [8,9,10] that may be applied to the functional specifications of menus. The techniques support the goals described above and are important for overcoming the problems associated with menu specifications. They include:

- **Top-down Decomposition:** A technique that breaks up complex topics into small pieces. The result of a top-down decomposition is a hierarchy of elements that can be used to substantially reduce the large number of test cases associated with some menus.
- **Abstraction:** A technique that defines how a complex topic is to be broken up into small pieces. Abstraction forces the detail of a complex topic to lower levels in the hierarchy of a top-down decomposition.
- **Self-contained Modules:** A self-contained module has a definition and operation that is complete within itself. A self-contained module restricts the amount of information that is needed for both designing and testing a menu module.
- **Module Content:** Sufficient information must be contained in a module description to design and test the module. The information should specify the states and state transformations that are valid and invalid for the module.

A method for the functional specification of menus has been developed for logic analyzers that utilizes these techniques. The method uses

abstraction to perform a top-down decomposition of a menu down to the field level (the base component of a menu). The top node of the decomposition hierarchy for the menu specifies the entire menu. The second level of elements in the decomposition hierarchy specifies the basic functions which comprise the menu. Each of the succeeding levels break down the menu functions until all of the menu fields have been reached. Figure 1 shows an abbreviated menu and its associated decomposition hierarchy.

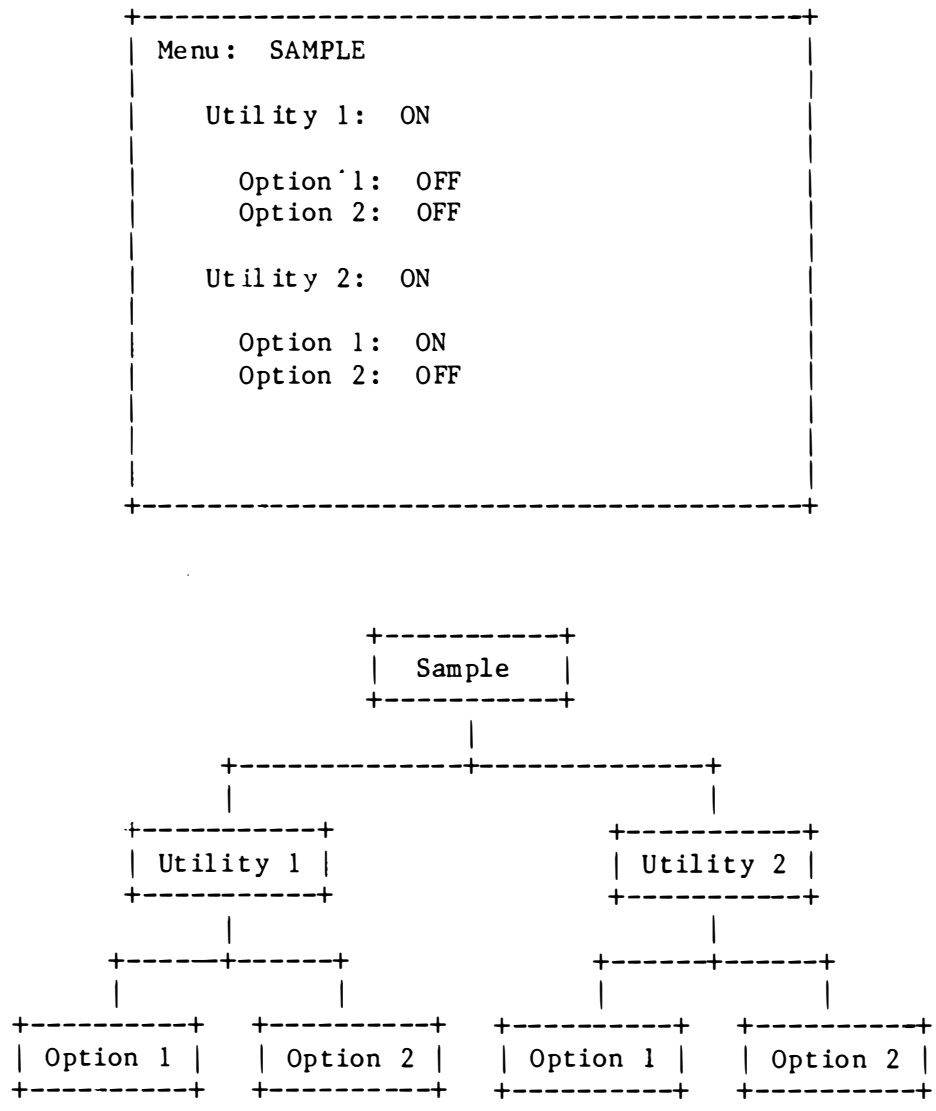


Figure 1. A sample menu with its associated decomposition hierarchy.

The elements in the decomposition hierarchy for the menu are called functions. A function identifies and defines a specific area on a menu. For each function the following information is specified:

- **Description:** Presents a concise English language description of the operation and content of the menu area covered by the function.
- **Operations and Values:** Specifies the keyboard keys that operate within the function and specifies the data values and nomenclature that are valid for the function.
- **Default Value:** Identifies the default values associated with power up, reset, or initial creation of the function.
- **Error Conditions:** Identifies all error conditions associated with the function and specifies how those conditions are resolved. Error conditions may result from invalid keys, entry of invalid values, or changes external to the function.
- **External Functions Affecting the Present Function:** Identifies all external functions (exclusive of parent functions) that may modify the structure or meaning of the present function in any way. Specifies the effect of the external function on the present function.
- **Subfunctions:** Identifies each of the subfunctions directly supporting the present function.

An example of a menu and its associated decomposition hierarchy, taken from the Tektronix DAS 9100 logic analyzer, is shown in Figures 2 and 3 respectively. The menu specifies the trigger conditions for a 91A24 data acquisition module. Inverse video is used to display data entry fields and a blinking screen cursor marks the menu field that currently may be modified. A special purpose keyboard is used on the DAS 9100 that supports the following menu operations:

- **Menu Selection:** A set of menu keys is available for selecting the desired menu.
- **Scrolling:** Data may be scrolled either vertically or horizontally.
- **Data Selection:** A "select" key may be used to scroll through predefined field entries.
- **Data Entry:** A set of data entry keys may be used to enter numeric and alphanumeric characters.
- **Special Function Keys:** A set of special function keys is available for performing specific functions within menus.

An example of a function specification for a function of the 91A24 Trigger Specification menu is shown in Figure 4.

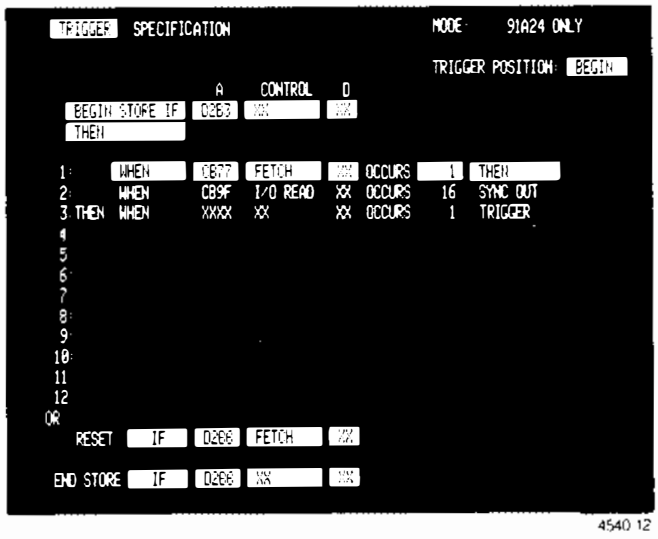


Figure 2. A snapshot of the 91A24 Trigger Specification menu found on the Tektronix DAS 9100 logic analyzer.

FUNCTIONAL ANALYSIS

Functional analysis consists of a group of problem-discovery methods that exercise the functions specified in a functional specification. These methods are both formal, such as proofs of correctness, and informal, such as document inspections. Together, the problem-discovery methods form a comprehensive and integrated approach for evaluating the functions of a system.

The problem-discovery methods may be divided into two subcategories [6]:

- **Static Analysis:** Pencil and paper or automated methods which analyze the system functional specification against a checklist of special properties.

- **Dynamic Analysis:** Actual operation methods that exercise the system functions under carefully controlled conditions.

Static Analysis

One form of static analysis is document inspection. Document inspection involves inspecting a functional specification against a checklist of desired properties. Howden [4] has specified a checklist of general properties which are suitable for the analysis of menus:

- **Consistency:** Each of the functions of the functional specification should be consistent in definition and operation.

- **Necessity:** Functions supporting goals that conflict with the intended goals of the system should be removed.

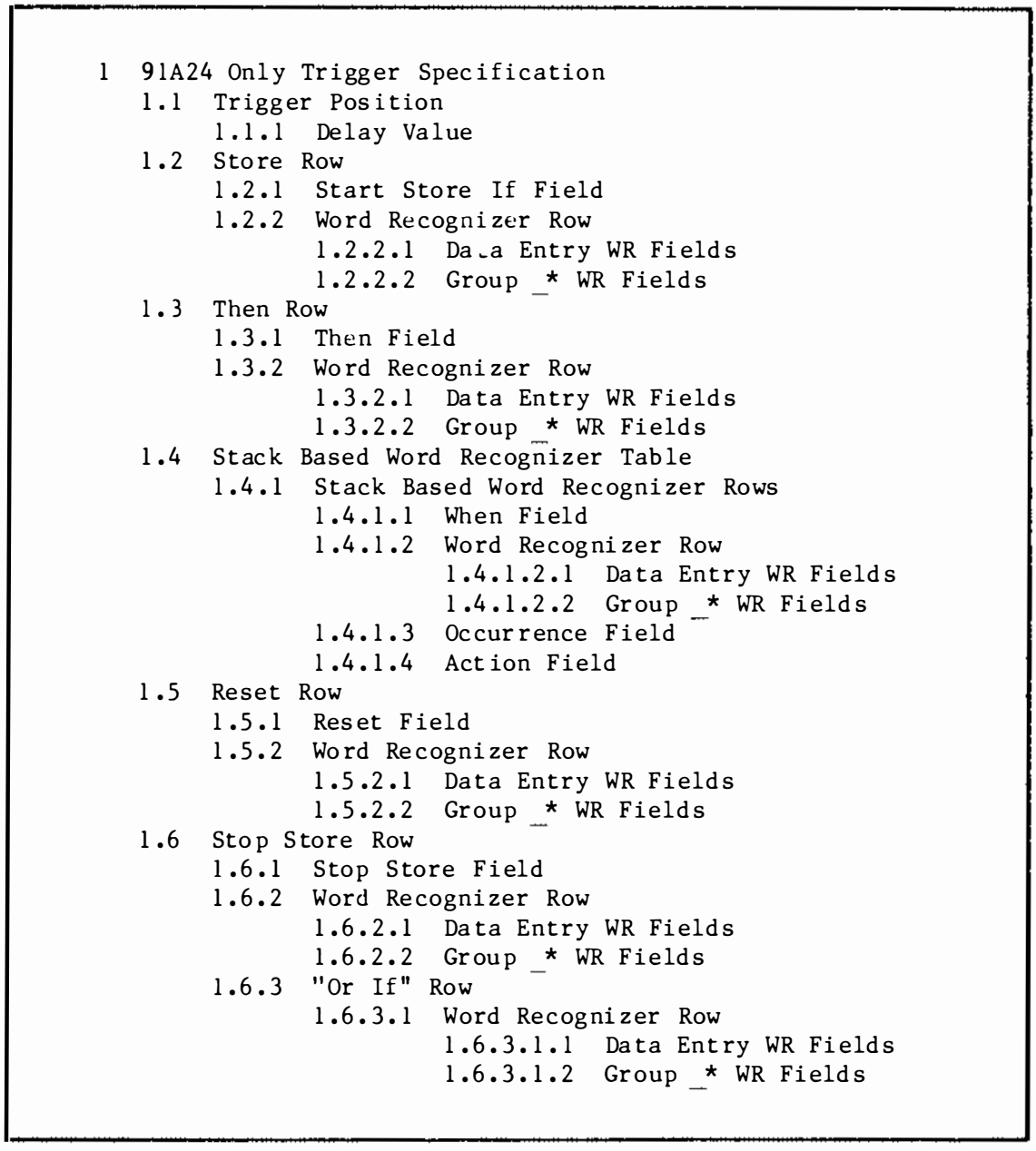


Figure 3. Decomposition hierarchy for the 91A24 Trigger Specification menu.

1.1.4 Stack Based Word Recognizer Table

1. Description

   Specifies the screen area that displays the 16 levels of the word recognizer stack. The display window displays up to 12 levels of the stack at a time.

2. Operations and Values

   A. SCROLL UP,SCROLL DOWN: Causes the vertical scrolling of the Stack Based WR rows.
   B. ADD LINE: Causes a new WR row to be added on the line below the line that the cursor is presently sitting on. All the lines below the cursor are then moved down one row. When 16 rows have been specified, the Add Line key is disabled.
   C. DEL LINE: Causes the line that the cursor is on to be deleted and all lower lines to be moved up.

3. Default Value

   A. Power Up: "WHEN XX XX XX OCCURS 1 TRIGGER"
      (Only one row is present in the table.)

4. Error Conditions

   A. Table Full: The message "TABLE FULL" is displayed when all 16 levels of the table have been defined and the Add Line key is pressed.

5. External Functions Affecting Present Function

   None.

6. Subfunctions

   A. Stack Based Word Recognizer Rows

Figure 4. Function specification for a 91A24 Trigger Specification menu function.

- **Sufficiency:** The set of functions defined for a system should fully meet the system goals.

- **Clarity:** The functions specified in the functional specification should be clearly defined.

- **Feasibility:** It should be possible to design and implement the functions that are desired for the system.

- **Testability:** It should be possible to test all of the functions that are specified for a system.

- **Correctness:** The functions for a system should operate coherently with each other.

Document inspection of a functional specification for a menu is performed on a function basis. Each function defined within the decomposition hierarchy for the menu is inspected against a checklist. A set of desirable menu attributes may be derived from the checklist of general properties specific to menu-based systems. An example of a checklist for menu functions is given in Figure 5. For any of the questions in the checklist except 1D, a "no" answer indicates that a problem may be present in either the human interface or the operation of the system.

Dynamic Analysis

Dynamic analysis of system functions involves selecting and establishing important data values [3]. Selection criteria for menu test data should include:

- **Extreme Values:** Values should be selected that are at the endpoints of the range of valid values for a function.

- **Nonextreme Values:** Values should be selected that are within the range of valid values for a function.

- **Special Values:** Values should be selected that have a specified importance to a function.

- **Combinations of Values:** Combinations of extreme, nonextreme, and special values should be selected to exercise a function.

Extreme, nonextreme, special, and combinational value test cases are constructed for each function in the system. The information for generating the test cases is easily taken from the menu functional specification described earlier. The format used in specifying a function identifies the test data and combinations of test data for the function, greatly simplifying the data selection process.

1. Description

   A. Is the function nomenclature consistent within the system, within the family of systems and within the industry?
   B. Does the nomenclature accurately define the function it labels?
   C. Does this function directly support its parent function?
   D. Are the effects of this function duplicated elsewhere in the system?

2. Operations and Values

   A. Is the operation performed consistently within itself, within the system, within the family of systems, and within the industry?
   B. Has the range of values for the function been defined?

3. Default Value

   A. Does the default value allow easy system operation for a majority of users?

4. Error Conditions

   A. Have all error conditions resulting from either local or external function operations been identified and resolved?

5. External Functions Affecting Present Function

   A. Have all external functions affecting the operations or values of this function been identified?

6. Subfunctions

   A. Is this function fully supported by its subfunctions?

Figure 5. A static analysis checklist for menu functions.

The structure of the functional specification for menus helps reduce the large number of tests that occur when different combinations of values are considered. The hierarchy and modularity of the menu decomposition breaks the menu down into functionally related subsets from which test data may be drawn. Functional tests for the bottom elements of the hierarchy exercise the individual fields of a menu. Higher elements in the hierarchy have tests that exercise combinations of fields and menu areas. As the hierarchy is ascended, functional areas are combined to reduce the number and size of tests.

The manual application of a functional dynamic test involves pressing a series of keys and visually verifying that the correct menu setups are attained. An example of a dynamic test is shown in Figure 6. The dynamic test format consists of a "Display Setup" section and a "Tests to Perform" section. The "Display Setup" section defines a series of setups that are applied to the menu. The "Tests to Perform" section defines tests that are applied to each display setup that is attained and tests that are applied independently of display setups.

AUTOMATED TEST EXECUTION

Menu-based systems may have functional tests that take days and weeks to execute. During system development, regression testing can further extend test times by weeks and months. Manually applying functional tests under such circumstances is long and tedious, resulting in low productivity and reduced system quality. The automation of functional test execution will significantly reduce the problems of productivity and quality and promote system development in a timely and cost-effective manner.

The automatic execution of functional tests for a menu may be performed by a program called an automatic menu test driver. The goals of an automatic menu test driver are:

- Simulate a user as closely as possible with respect to keyboard input and screen read back.
- Provide a standard notation for specifying the functional tests for a menu.
- Automate the verification of test results.
- Automate the error recovery process.

An automatic menu test driver called the Automated Instrument Tester (AIT) has been developed at Tektronix for use on the DAS 9100 logic analyzer. A block diagram of the AIT is shown in Figure 7. The AIT system is composed of a host (PDP 11/35), an instrument under test, and a user terminal. The instrument under test is connected to the host by either RS-232 or GPIB interfaces.

1.1.4 Stack Based Word Recognizer Table

Display Setup:

1. Stack Based Word Recognizer Table:

   A. Place one entry in the table.
   B. Place eight unique entries in the table.
   C. Place sixteen unique entries in the table.

Tests to perform:

1. Confirm each Display Setup:

   A. Place the inverse video field at the top of the screen.
   B. Scroll to both ends of the table.
   C. Place the inverse video field in the center of the screen.
   D. Scroll to both ends of the table.
   E. Place the inverse video field at the bottom of the screen.
   F. Scroll to both ends of the table.

2. Confirm the operation of the ADD LINE and DEL LINE keys (applied independently of display setup):

   A. Press ADD LINE and DEL LINE at the top of the screen.
   B. Press ADD LINE and DEL LINE in the center of the screen.
   C. Press ADD LINE and DEL LINE at the bottom of the screen.

3. Verify the table full message by trying to place more that 16 entries in the table.
   (Applied independently of display setup.)

Figure 6. A dynamic test for a 91A24 Trigger Specification function.

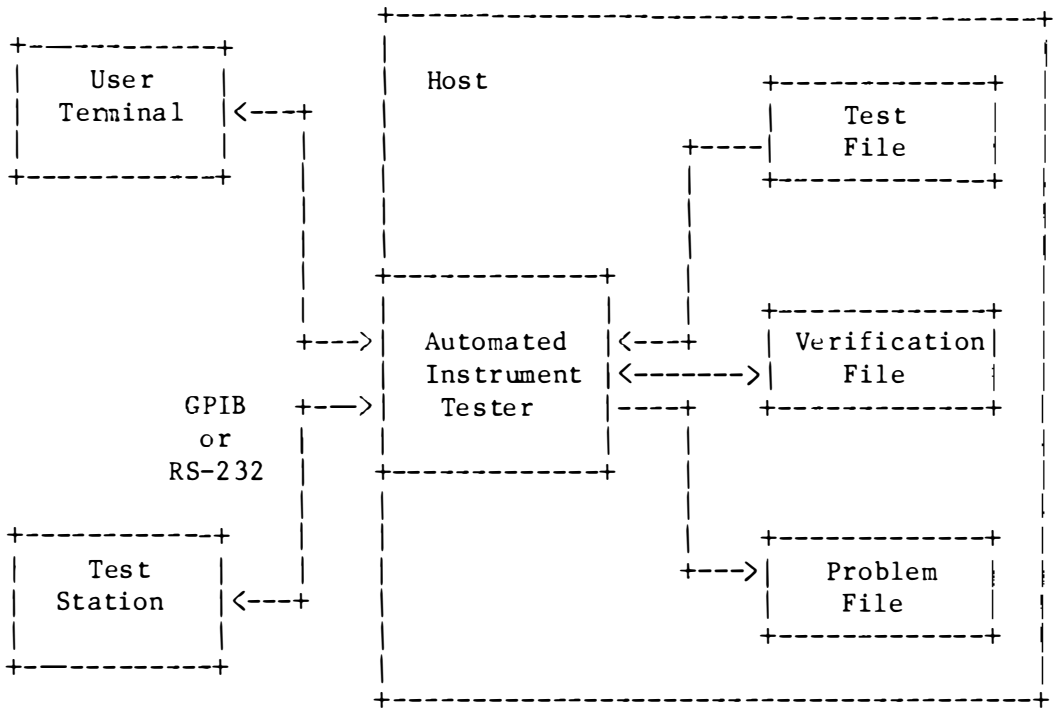


Figure 7. Block diagram of the Automated Instrument Tester structure.

Three files are used by the AIT during test execution: A test file, a verification file, and a problem file. The test file contains AIT commands and key stroke commands sent to the instrument under test. Each test file is created manually with a text editor. The verification file contains menu information used in verifying test results. Verification files may be created either manually with a text editor or automatically by the AIT. The problem file maintains a list of problems encountered during test execution and is generated as output.

The process of executing an AIT test consists of several steps. Key stroke commands are retrieved from the test file and sent to the instrument under test. The instrument processes the key stroke commands as if they were entered via the instrument keyboard. Menu information is then received back from the instrument and compared against menu information contained in the verification file. If a problem is recognized, an entry is recorded into the problem file for later analysis.

The user terminal is used to monitor and control test execution. Test status and verification information is automatically displayed on the terminal during test execution. The user has the capability of stopping and resuming test execution and performing a variety of other operations from the terminal keyboard.

A sample test file is shown in Figure 8. Two commands, ".Function" and ".Test", are used for breaking the test into modules. A ".Function" is specified for each function defined in the functional specification for a menu. One or more ".Test" sections may be created for each function. A ".Verify" command is used for reading a portion of the menu and comparing it against menu information contained in the verification file. Slashes (/) are used to indicate the beginning of comments.

```
.Function:  1.1.4  Stack Based Word Recognizer
 Init;             /  Initialize the DAS 9100
 Trigger;          /  Display the 91A24 Trigger Spec menu

.Test: 1
 /  Verify the "TABLE FULL" message

 Down 4;                    /  Move the screen cursor to the table
 Addline 11;                /  Add 11 entries to the table
 Scroll up, 4;              /  Scroll the table up 4 entries
 Addline 4;                 /  Finish filling the table
 Addline;                   /  Try adding an entry to the full table
 .Verify: CRT,1,1,0,40      /  Verify that the "TABLE FULL" message
                               has been displayed
```

Figure 8. A sample automated test for a 91A24 Trigger Specification menu function.

Automated testing results in shorter test execution times, better test coverage, and higher software quality. For example, manually testing the 91A24 Trigger Specification menu requires about 50 hours, the AIT performs the same job in only 8 hours. Automation improves test accuracy, as well, by avoiding human error during test execution. Only one problem has been identified in the 91A24 Trigger Specification menu in the year since its release. The low problem count is a good indicator of high software quality.

TESTABILITY REQUIREMENTS

The ease with which the specification, analysis, and automated testing methods presented here can be applied and utilized is largely dependent on the structure, content, and operation of the system menus. Specifically, the following design aspects affect the testability of menu-based systems:

- **Functional Specification Content:** The functional specification for a menu-based system should be complete. The static analysis of an incomplete specification can fail to identify problems. The cost of fixing a problem in code is much higher than fixing the problem in a functional specification.

- **Menu Organization:** A menu should be designed so that it can be easily decomposed into the hierarchy required for a functional specification. Menus not easily decomposed are usually difficult to test and often represent a poor human interface.

  The number of cross-dependencies within the hierarchy for a menu also affects the testability of the menu. A function that has a high number of other functions affecting its operation takes longer to test and is more prone to errors.

- **Menu Cursor Operation:** It should be possible to move the menu screen cursor from an unknown menu location to a known location by sending cursor key stroke commands. One way to accomplish this is to cause the screen cursor to stop when it reaches the end of a line or the top or bottom of the screen. The reason for this design constraint is that in some automated tests the exact location of the screen cursor is not known. A series of cursor key stroke commands must be sent to place the cursor in a known location before continuing the test. If a wrap-around cursor implementation scheme is used (where the cursor can go to the next line by sending a left or right cursor command), it would not be possible to place the cursor in a known location. Such a scheme would limit the automated testing capabilities.

- **Key Stroke Input and Menu Output:** A menu-based system should have the capability of accepting key strokes and sending menu text over an interface to a host computer. The input of key strokes and the output of menu information are both necessary to support the automated testing of menus. It is important that the processing of key stroke information from a host be handled in the same manner as the processing of normal keyboard input. One of the objectives of automated testing is to simulate a user as closely as possible. A wide disparity in the processing of host and keyboard key strokes will not meet this objective.

- **Active and Passive Design Constraints:** The software design of a menu-based system should identify and isolate design constructs where errors are difficult to recognize through automated testing. An example would be placing the port of a programmable CRT controller next to the CRT screen buffer. If the CRT screen buffer is accidently overwritten, the CRT controller could be reprogrammed without being detected by automated testing. Solutions to this problem might be to move the port to a less volatile area or to implement screen interface routines which prohibit writing outside of the screen buffer.

CONCLUSION

Methods for the functional specification, functional analysis, and automated testing of menus have been presented that yield high software quality in menu-based systems. Testability requirements have been identified and described that are critical to the use and success of the methods. The results of the case study on the 91A24 Trigger

Specification menu indicate that the methods are viable and indeed necessary to achieve a high quality system within a feasible time period. The methods are also general enough to be applied to a wide range of menu-based systems.

New techniques are being explored for improving the specification and analysis methods. One technique involves the use of a formal menu specification process to reduce problems found through static analysis. A second technique concerns automated test generation. The two techniques complement each other and offer the possibility for further improvements in the software quality of menu-based systems.

A formal menu specification process is being developed that defines the valid node types allowed in a menu hierarchy. When a menu is created, a designer selects predefined node types to build a hierarchy and define the menu. The process avoids the user interface problems and testability problems encouraged by informal methods.

Automated test generation is being examined as a method for creating tests executed by an automatic menu test driver. A generic automated test is defined for each of the node types used in building a menu hierarchy. An automated test generator then processes a menu specification by combining the information from machine-readable function specifications and generic tests to create fully operational automated tests. The potential exists for large reductions in test development time.

The ultimate goal for a testable menu-based system is the full automation of all testing activities. Automated static analysis, automated dynamic test creation, and automated test execution will greatly reduce testing time and increase software quality.

ACKNOWLEDGEMENTS

The author wishes to thank Larry Larison, Allen Cicrich, Steve Kronschnabel, and Jim Fenton for their valuable suggestions in revising several drafts of this paper. The author also acknowledges Doug Warren and Steve Kronschnabel for their work in developing the Automated Instrument Tester.

REFERENCES

1. Broughton, Robert S., "Structured Logic Analysis for Manufacturing Testing", IEEE 1983 International Test Conference, pp. 538-543.

2. "DAS 9100 Series Operator's Manuals", c1982, Tektronix, Inc., part number 070-3264-00, Beaverton, Oregon.

3. Howden, William E., " Applicability of Software Validation Techniques to Scientific Programs", ACM TOPLAS, Vol. 2, No. 3, July 1980, pp. 307-320.

4. Howden, William E., "Validation of Scientific Programs", ACM Computing Surveys, Vol. 14, No. 2, June 1982, pp. 193-227.

5. "Logic Analyzer Concepts", c1984, Tektronix UK Limited, Harpenden, Herts, England.

6. Miller, Edward, "Introduction to Software Testing Technology", Software Testing and Validation Techniques, E. Miller and W. E. Howden (Eds.), IEEE Computer Society, Long Beach, 1981, pp. 4-16.

7. Parnas, D. L., "A Technique for Software Module Specification with Examples", Communications of the ACM, Vol. 15, No. 5, May 1972, pp. 330-336.

8. Robinson, L. and Levitt, K. N., "Proof Techniques for Hierarchically Structured Programs", Communications of the ACM, Vol. 20, No. 4, April 1977, pp. 271-283.

9. Ross, Douglas T. and Schoman, Kenneth E., "Structured Analysis for Requirements Definition", Software Design Techniques, P. Freeman and A. I. Wasserman (Eds.), Long Beach, 1980, pp. 97-125.

10. Savage, Ricky E., Habinck, James K., and Barnhart, Thomas W., "The Design, Simulation, and Evaluation of a Menu Driven User Interface", Proc. Human Factors in Computer Systems, 1982, pp. 36-40.

THE DESIGN OF TESTABLE MENU-BASED SYSTEMS

ALLEN SAMPSON

TEKTRONIX, INC.

BEAVERTON, OREGON 97075

OVERVIEW

INTRODUCTION

FUNCTIONAL SPECIFICATION

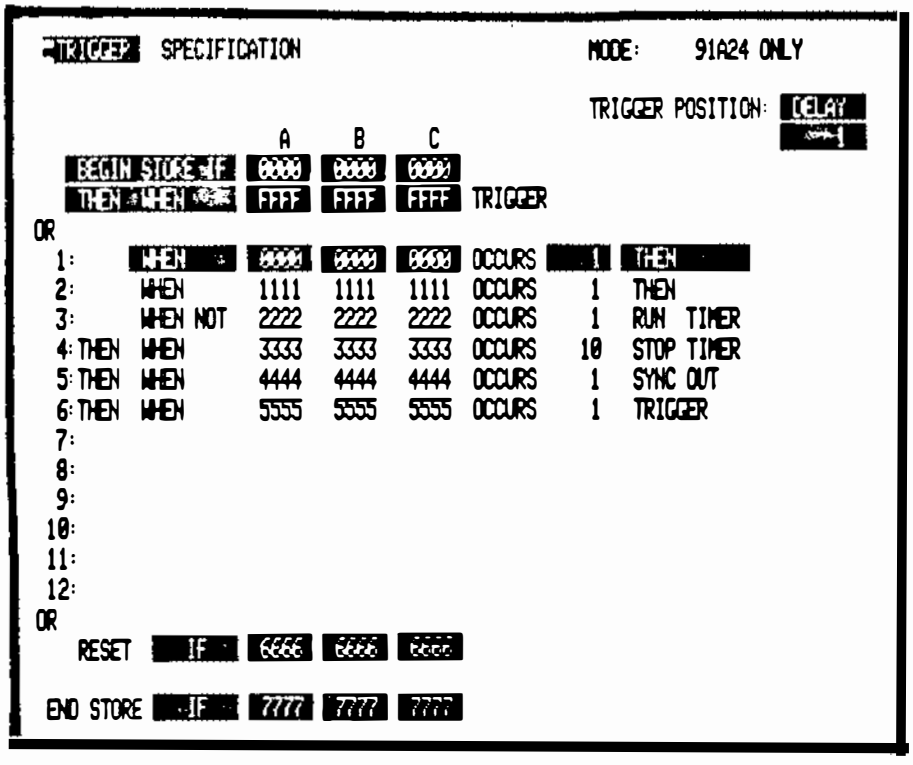
FUNCTIONAL ANALYSIS

AUTOMATED TEST EXECUTION

TESTABILITY REQUIREMENTS

CONCLUSION

DAS 9100 SERIES LOGIC ANALYZER



91A24 TRIGGER SPECIFICATION MENU

MENU CHARACTERISTICS:

- **HIERARCHY OF MENUS**
- **IDENTIFYING NOMENCLATURE**
- **DATA ENTRY FIELDS**
- **SCREEN CURSOR**
- **MENU EDIT FUNCTIONS**

FUNCTIONAL SPECIFICATION

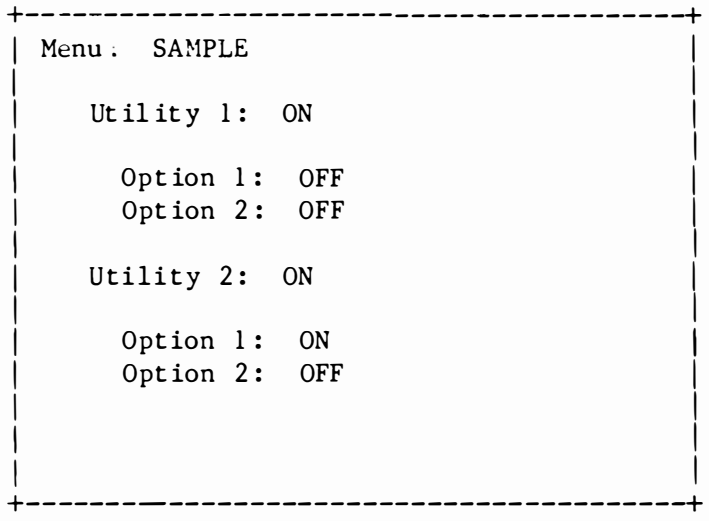
GOALS:

- **PROVIDE DESIGNER WITH INFORMATION TO CREATE SOFTWARE DESIGN**
- **PROVIDE EVALUATOR WITH INFORMATION TO CREATE FUNCTIONAL TESTS**
- **WRITTEN IN TERMS USED BY BOTH**

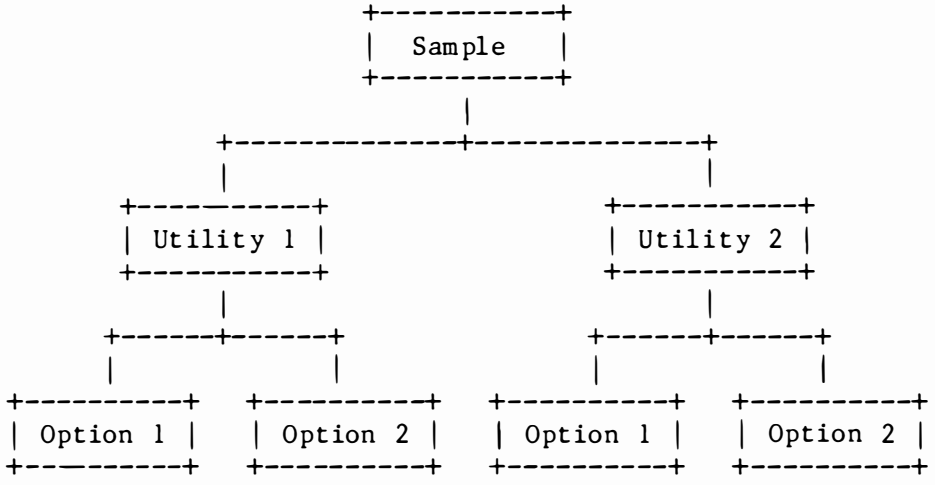
TECHNIQUES:

- **SELF-CONTAINED MODULES**
- **ABSTRACTION**
- **TOP-DOWN DECOMPOSITION**
- **MODULE CONTENT**

SAMPLE MENU



MENU

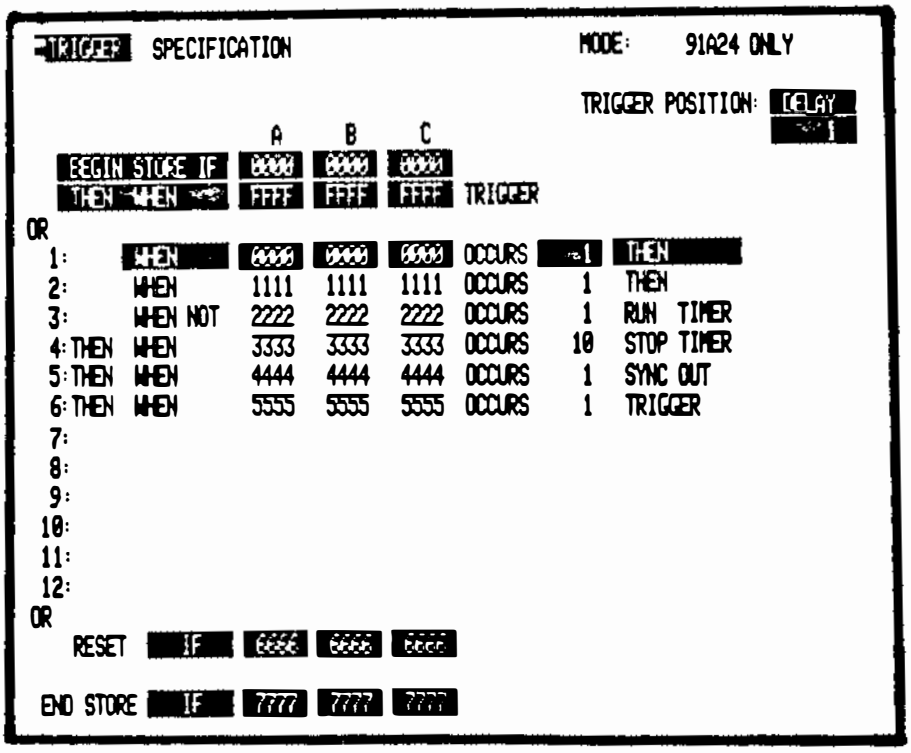


DECOMPOSITION HIERARCHY

FUNCTION SPECIFICATION FORMAT

1. DESCRIPTION
2. OPERATIONS AND VALUES
3. DEFAULT VALUE
4. ERROR CONDITIONS
5. EXTERNAL FUNCTIONS AFFECTING THE PRESENT FUNCTION
6. SUBFUNCTIONS

91A24 TRIGGER SPECIFICATION



MENU

1 91A24 Only Trigger Specification
- 1.1 Trigger Position
  - 1.1.1 Delay Value
- 1.2 Store Row
  - 1.2.1 Start Store If Field
  - 1.2.2 Word Recognizer Row
    - 1.2.2.1 Data Entry WR Fields
    - 1.2.2.2 Group _* WR Fields
- 1.3 Then Row
  - 1.3.1 Then Field
  - 1.3.2 Word Recognizer Row
    - 1.3.2.1 Data Entry WR Fields
    - 1.3.2.2 Group _* WR Fields
- 1.4 Stack Based Word Recognizer Table
  - 1.4.1 Stack Based Word Recognizer Rows
    - 1.4.1.1 When Field
    - 1.4.1.2 Word Recognizer Row
      - 1.4.1.2.1 Data Entry WR Fields
      - 1.4.1.2.2 Group _* WR Fields
    - 1.4.1.3 Occurrence Field
    - 1.4.1.4 Action Field
- 1.5 Reset Row

⋮

DECOMPOSITION HIERARCHY

91A24 TRIGGER SPECIFICATION

1.1.4 Stack Based Word Recognizer Table

1. Description

Specifies the screen area that displays the 16 levels of the word recognizer stack. The display window displays up to 12 levels of the stack at a time.

2. Operations and Values

A. SCROLL UP,SCROLL DOWN: Causes the vertical scrolling of the Stack Based WR rows.
B. ADD LINE: Causes a new WR row to be added on the line below the line that the cursor is presently sitting on. All the lines below the cursor are then moved down one row. When 16 rows have been specified, the Add Line key is disabled.
C. DEL LINE: Causes the line that the cursor is on to be deleted and all lower lines to be moved up.

3. Default Value

A. Power Up: "WHEN XX XX XX OCCURS 1 TRIGGER"
(Only one row is present in the table.)

4. Error Conditions

A. Table Full: The message "TABLE FULL" is displayed when all 16 levels of the table have been defined and the Add Line key is pressed.

5. External Functions Affecting Present Function

None.

6. Subfunctions

A. Stack Based Word Recognizer Rows

FUNCTION SPECIFICATION

FUNCTIONAL ANALYSIS

GROUP OF PROBLEM-DISCOVERY METHODS THAT ANALYZE SYSTEM FUNCTIONS.

- **STATIC ANALYSIS**
- **DYNAMIC ANALYSIS**

STATIC ANALYSIS

GENERAL CHECKLIST:

- **CONSISTENCY**
- **NECESSITY**
- **SUFFICIENCY**
- **CLARITY**
- **FEASIBILITY**
- **TESTABILITY**
- **CORRECTNESS**

DYNAMIC ANALYSIS

TEST DATA SELECTION:

- **EXTREME VALUES**
- **NONEXTREME VALUES**
- **SPECIAL VALUES**
- **COMBINATIONS OF VALUES**

91A24 TRIGGER SPECIFICATION

1. Description

   A. Is the function nomenclature consistent within the system, within the family of systems, and within the industry?

   B. Does the nomenclature accurately define the function it labels?

   C. Does this function directly support its parent function?

   D. Are the effects of this function duplicated elsewhere in the system?

2. Operations and Values

   A. Is the operation performed consistently within itself, within the system, within the family of systems, and within the industry?

   B. Has the range of values for the function been defined?

3. Default Value

   A. Does the default value allow easy system operation for a majority of users?

4. Error Conditions

   A. Have all error conditions resulting from either local or external function operations been identified and resolved?

5. External Functions Affecting Present Function

   A. Have all external functions affecting the operations or values of this function been identified?

6. Subfunctions

   A. Is this function fully supported by its subfunctions?

STATIC TEST

91A24 TRIGGER SPECIFICATION

1.1.4 Stack Based Word Recognizer Table

Display Setup:

1. Stack Based Word Recognizer Table:

   A. Place one entry in the table.
   B. Place eight unique entries in the table.
   C. Place sixteen unique entries in the table.

Tests to perform:

1. Confirm each Display Setup:

   A. Place the inverse video field at the top of the screen.
   B. Scroll to both ends of the table.
   C. Place the inverse video field in the center of the screen.
   D. Scroll to both ends of the table.
   E. Place the inverse video field at the bottom of the screen.
   F. Scroll to both ends of the table.

2. Confirm the operation of the ADD LINE and DEL LINE keys:

   A. Press ADD LINE and DEL LINE at the top of the screen.
   B. Press ADD LINE and DEL LINE in the center of the screen.
   C. Press ADD LINE and DEL LINE at the bottom of the screen.

3. Verify the table full message by trying to place more that 16 entries in the table.

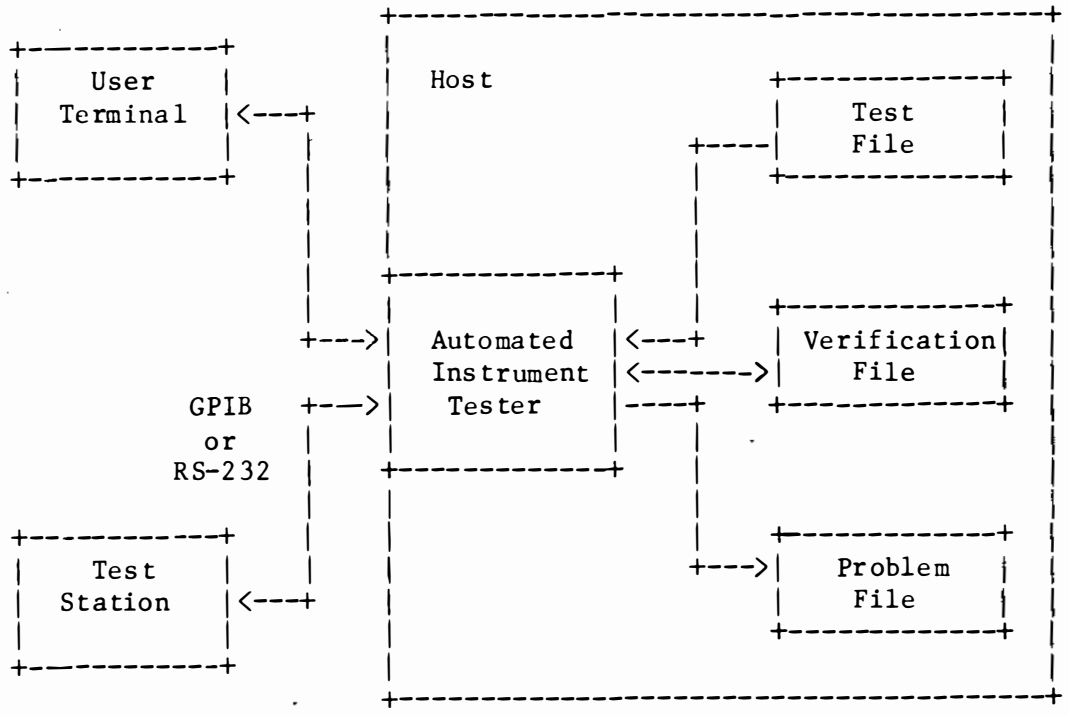
DYNAMIC TEST

AUTOMATED TEST EXECUTION

GOALS:

- **SIMULATE A USER**
- **PROVIDE STANDARD NOTATION FOR TESTS**
- **AUTOMATE VERIFICATION OF TEST RESULTS**
- **AUTOMATE ERROR RECOVERY PROCESS**

AUTOMATED INSTRUMENT TESTER



BLOCK DIAGRAM

```
.Function:  1.1.4  Stack Based Word Recognizer
 Init;          /  Initialize the DAS 9100
 Trigger;       /  Display the 91A24 TriggerSpec menu

.Test: 1
 /  Verify the "TABLE FULL" message

 Down 4;                  /  Move the screen cursor to the table
 Addline 11,              /  Add 11 entries to the table
 Scroll up, 4;            /  Scroll the table up 4 entries
 Addline 4;               /  Finish filling the table
 Addline;                 /  Try adding an entry to the full table
 .Verify: CRT,1,1,0,40    /  Verify that the "TABLE FULL" message
                             has been displayed
```

AUTOMATED TEST

TESTABILITY REQUIREMENTS

- FUNCTIONAL SPECIFICATION CONTENT
- MENU ORGANIZATION
- MENU CURSOR OPERATION
- KEY STROKE INPUT AND MENU OUTPUT
- ACTIVE AND PASSIVE DESIGN CONSTRAINTS

CONCLUSION

- METHODS YIELD HIGH SOFTWARE QUALITY
- TESTABILITY REQUIREMENTS ARE CRITICAL TO THE USE AND SUCCESS OF METHODS

NEW TECHNIQUES BEING EXPLORED:

- AUTOMATED TEST GENERATION
- FORMAL MENU SPECIFICATION PROCESS

TESTABILITY GOAL

FULL AUTOMATION OF TESTING ACTIVITIES:

- **AUTOMATED STATIC ANALYSIS**
- **AUTOMATED DYNAMIC TEST CREATION**
- **AUTOMATED DYNAMIC TEST EXECUTION**

BIOGRAPHICAL SKETCH

Allen Sampson received B.S. and M.S. degrees in Computer Science from the University of Utah in 1977 and 1980. He currently leads the software evaluation effort for logic analyzer new product introduction at Tektronix. He has 4 years experience in developing and refining evaluation methods for menu-based systems and has also defined and implemented an automated menu test driver currently used on the DAS 9100 Series Logic Analyzer. Mr. Sampson is a member of the Association for Computing Machinery and the Society of Reliability Engineers.

Ingredients for a Quality Product

by,
Susan Wechsler

Often, quality is referred to only as an end, such as a bug rate to satisfy a completion criteria. But the real nature of quality is in bringing maximum value to the customer, and this implies keeping costs down, as well as building a superior product. When focusing on quality from this perspective, it becomes clear that quality is also a process concerned with minimizing waste and maximizing productivity. Therefore, quality begins with product definition, and continues through final testing.

Hewlett-Packard used this philosophy to develop a revolutionary new handheld computer, the HP-71B. Its 64K byte operating system, written in assembler, was designed to encourage third party software. The enhanced BASIC operating system includes built-in statistics and math conforming to the proposed IEEE floating point math standard, a sophisticated file system, and the capability to be customized for almost any task. The proprietary CPU was developed and debugged concurrently with the software, meaning that for the initial design and development phases there were no available tools for ascertaining actual system performance. Thus, the HP-71 presented a formidable challenge for the design and development of a quality product.

Eleven software engineers contributed to the product, requiring a significant coordination effort. The effort paid off in the successful introduction of the HP-71. At the time the code was finished, the documentation was complete, which facilitated testing, third party software, and product support. And, although developed by so many engineers, the HP-71 software is well integrated, providing a consistent, easy to use interface to a powerful technical tool.

The HP-71 quality process falls into three major categories: Tools, Documentation, and Evaluation. Several hardware and software tools were an asset to productivity in the design, development, and debugging phases. The emphasis on documentation ensured that quality was built into the machine from the onset since it reduced ambiguity in the product design and misunderstanding among team members. Evaluation refers to testing, and to measuring and improving system quality.

TOOLS

The tools challenge for the HP-71 was to provide an environment in which eleven engineers could develop software for a 64K system, minimizing any negative impact on each other's productivity. The

linker, software and hardware simulators, portable RAM boxes, and debugger all contributed to the productivity of the design team.

The number of development engineers, the size of the operating system, and the time required to assemble long source files made it necessary to partition the operating system into many modules. The linker that was developed could load as many as 80 object modules; it also provided a comprehensive cross reference table that was an indispensible debugging tool and facilitated the use of common symbolics and subroutines.

The software simulator allowed unlimited break point setting, single stepping of machine-level instructions, viewing of all CPU registers simultaneously, and the capability to change the contents of CPU registers. The software simulator was a powerful tool, but as code was initially written and tested, it became clear that the operating system's actual friendliness and response time could only be guessed at -- interaction with a handheld operating system running at 1/200 actual speed, through a terminal connected to a mainframe certainly left a lot to the imagination.

The hardware simulator that later became available ran at the proper CPU speed, provided numerous other enhancements, and gave initial insight into the personality of the final product. Unfortunately, this first hardware simulator had to be shared among all the development engineers, so to facilitate productivity, the work day was divided into 1/2 hour time slots that had to be reserved in advance. Because of the scarcity of available time, the hardware simulator was primarily used for tracking down the cause of specific bugs.

Initially, the hardware simulator consisted of several breadboards. As the actual chips became available, they were substituted into the hardware simulator for the various breadboard components, simulating the machine's ultimate operation as much as possible, while providing a test station for the chips themselves.

The next available hardware tool revolutionized the quality effort. This was a portable RAM box with 80K of memory that could be configured in a variety of ways, including as built-in ROM or plug-in RAM. RAM boxes were produced in large enough quantities that, with the handset containing CPU and display driver chips, each software team member had a dedicated testing station. Experimental versions of the operating system could be downloaded into the RAM box in a matter of minutes. The design team was finally able to use the product as extensively as would an end user. Flaws in the user interface that were not seen before quickly became obvious, and enhancements were made to add consistency and to improve ease-of-use. Also, the availability of RAM boxes allowed application engineers to start testing their interfaces to the operating system, adding a whole new staff of functional testers.

The software debugger resident in a portable RAM box practically eliminated the need for a hardware simulator. The debugger provided most of the functionality of the hardware simulator, including some enhancements. The two biggest advantages gained by the debugger/portable RAM box were that productivity was no longer dependent on the availability of a heavily utilized mainframe, and that because of the portable nature of the RAM boxes, engineers could take work home, thereby also increasing productivity.

DOCUMENTATION

In any group effort, one of the most important ingredients for success is clear, unambiguous communication. We found that the most effective way to avoid misunderstandings was to keep ideas thoroughly documented. The importance of documentation is often downplayed because of tight schedules. Yet ironically, the fast pace necessitated by a tight schedule makes communication even more crucial, since so much more can go awry in a shorter period of time.

For the HP-71, there were additional reasons for wanting a well documented system. For the first time in the history of Hewlett-Packard calculator development, the decision was made to provide all the necessary tools to encourage third party software and hardware companies to customize the product.

System documentation included the External Reference Specification (ERS), the Internal Design Specification (IDS), a weekly newsletter, and informational headers preceding every piece of code in the system. Each of these was instrumental in promoting a team effort.

The ERS grew with the HP-71 definition. As with most products, the initial activity for the HP-71 was setting product objectives. Product objectives are largely based on the target market, and for calculators, target market determines size, price window, and core functionality. Once this market was identified, the next step was preliminary product definition. Looking at the company's strategic plans, as well as at the target market, the product went through a refinement period in which some of the more major design issues were resolved. From this period emerged a precursor to the ERS. This preliminary ERS was expanded and fine-tuned throughout product design and development.

A Hewlett-Packard R&D project goes through a formal transition, taking it from the investigation phase to the development phase. The transition is complete when the product, as defined in its preliminary ERS, gains the support and approval of all the necessary functional areas. Marketing must be confident the product can be sold into its target market, Manufacturing must be confident the product can be built, Quality Assurance must develop the evaluation plan, and the Lab must allocate the necessary resources for product development. At

this crucial stage, projects are often cancelled or modified. Some are perceived by marketing as being too complicated to explain to the target market; others have hidden costs exposed that would price them out of the intended market; and still others address a market too small to justify the product. In any case, once a product gets the necessary commitment, its product objectives are considered more or less fixed.

As product design is translated into product development, a project is vulnerable to one of the biggest threats known: 'creeping featurism'. 'Creeping featurism' is characterized by a product's functionality growing beyond what was originally conceptualized in an ERS. It causes a product's cost to rise beyond what was anticipated, and it is a major obstacle to quality. Features that were not originally included in the design have a tendency to be added like bandaids, causing problems, which in turn require additional bandaids to fix. A certain amount of creeping featurism is inevitable in every product, since it is neither feasible nor reasonable to conceptualize an entire product in a preliminary ERS; but as a general rule, creeping featurism is something to avoid. The HP-71 was no exception. The areas of the machine that caused the most problems were related to features added later that were not essential to the product, but that were perceived as making a major contribution. Although we were confident such features could be added with minimal effort and ROM, in each case we had underestimated on both counts. That is not to say that the enhancements did not make a significant contribution to the finished product -- they did; but the cost in time and effort cannot be overlooked. An ERS can be a useful yardstick during product development to keep creeping featurism in check.

The three volume Internal Design Specification (IDS) provided an in-depth, thorough view of the operating system implementation. Volume I contained information about internal data structures, algorithms for the more complicated code, the CPU instruction set, and instructions on everything from resource allocation to writing a LEX (Language Extension) file to extend the machine's capability. Volume II discussed the interface to every supported entry point. Volume III contained the source code at release time.

The ERS and IDS were more than just an effective means of minimizing ambiguity. They proved to be valuable time savers for the development team. The final ERS contained a keyword dictionary, supplying detailed information on the purpose, syntax, operation, and error conditions of each BASIC keyword, so that while our Technical Publications Department was writing the HP-71 Reference Manual, designers were able to continue testing; because the information was already in place, little additional time was spent working with Technical Publications on this manual. Also, the IDS freed us from continually being interrupted to provide details to applications programmers or third party software vendors.

A newsletter was circulated among the software engineers to convey system changes, code packing tips, documentation conventions, and useful subroutines. The importance of the newsletter became increasingly obvious as we got further into the product development. Publishing existing debugged subroutines led to the elimination of routines scattered throughout the operating system that had similar functionality, but different implementation. Not only was code safely packed in this manner, but more significantly, a core set of well known low level utilities emerged, ultimately reducing the inherent QA headache when similar functionality is accomplished with different pieces of code scattered throughout the machine.

A documentation header preceded every routine, providing the name, functional category, purpose, entry and exit conditions, names of called routines, register use, subroutine level use, and history of any modifications. The empty header shell was in an on-line file, allowing a designer to merge the shell directly into the source code file and provide the appropriate information; every effort was made to clarify and simplify the code documentation process.

Engineering hours spent on documentation headers were well worth it. Before the documentation was in place, an engineer could spend long moments staring at a piece of his/her own code, attempting to figure out how it worked, what it did, and what it used. After the documentation was in place, we could peruse each other's routines, making possible the code review process that came later.

In a group software effort, it is of vital importance that a standard for documenting utilities be established at the outset. Our documentation header evolved as the HP-71 was developed, and the cost incurred as a result was considerable. By the time code was revisited to properly document, the engineer was no longer as familiar with it as when it was first written. Consequently, mistakes were made. And mistakes made when documenting widely used utilities can have big implications. Also, for every low level utility revisited, the calling routines had to be carefully examined to ensure that the new information did not uncover a potential problem; then the calling routine itself had to be redocumented, and any references to it, checked out in turn. Sometimes, this effect rippled through dozens of routines. This methodical and time consuming effort to improve documentation headers became our earliest form of code review. However, many of the bugs found with this exercise could have been avoided if the documentation had been in place from the beginning. It is impossible to overemphasize the importance of timely, detailed, and accurate documentation of routines, especially for low level utilities.

The benefits of documenting utilities are well known; however, we took the benefits a step further. A program was written to extract the documentation headers from the source code, and sort the utilities by category. This resulted in a utility library for each area of the

operating system, facilitating the use of common code. In addition, extraction could be restricted to supported utilities, automatically generating IDS Volume II.

EVALUATION

When HP-71 software was nearing completion, emphasis shifted from code generation and bug fixing to a concentrated focus on product evaluation. A bug rate completion criteria was established, and maximum effort went into finding, prioritizing, and categorizing bugs. Evaluation involved design and code reviews, functional testing, comparison of the documentation with actual operation, a bug tracking database, an innovative incentive program for finding bugs, and bug fixing.

Design reviews were done on complex code that seemed particularly vulnerable to bugs. During a design review, a small subset of the software team looked at the designer's algorithm and listened to the designer's explanation of his/her thought processes and assumptions. Frequently, during the detailed explanation, the designer suddenly realized a problem that he/she had not previously conceptualized; and occasionally, members of the design review team found omissions or misconceptions in the algorithm.

As alluded to earlier, code review on any meaningful level would not have been possible without documentation headers. During the code review process, a designer reviewed source code and the accompanying documentation unassisted by the author. Optimally, the reviewer was at least vaguely familiar with that area of the machine. Code review uncovered bugs as well as documentation problems. Although the intention was to cover the entire operating system with code review, time constraints limited it to only about 20% coverage. Code review, however, did cover all critical areas.

Functional testing placed a premium on thoroughness, and the dynamic nature of the operating system (variable memory configuration, 128 system flags, optional peripherals, plug-in software, etc) added to the challenge of thorough testing. Engineers on the design team did as much 'white box' testing as possible, especially to ensure code coverage in regression tests.

Given all the parameters that required consideration, regression testing would have been inconceivable without an automated test system. Two methods of automated testing were employed. The first involved BASIC programs that were written to test the extended precision math. The other test system, known as ATS (Automated Test System), consisted of a FORTRAN program running on the HP-3000, and hardware tied directly into the input lines of an HP-71 handset. In addition to electronically pressing keys and reading the HP-71 display

buffer, ATS could be used to either create a file of test results, or to generate a comparison file showing discrepancies between current test results and a file containing expected results. ATS 'scripts' written by the designers extensively tested parse, decompile, and execution for all 240 BASIC keywords. Regression tests included bounds, equivalence, and error testing. When tests were written, the relevant code was examined to ensure coverage; and when a bug was discovered and fixed, a test was added to the appropriate BASIC program or ATS script to guarantee that it did not reappear unnoticed.

Although each designer was responsible for testing his/her own code, the testing binge was a functional testing technique that produced results. There were two types of binges. The first involved selecting a specific part of the operating system for intensive testing by all team members. The second involved partitioning a subset of the operating system among the team, and giving individual assignments so that no one was testing their own code. A binge assignment lasted anywhere from a day to a week, depending on the number and severity of the bugs found. The former type was reserved for areas that were considered high risk; the latter was used to ensure that the entire operating system was adequately covered by functional testing.

One extremely valuable source of functional testing was our application engineers. To ensure early availability of plug-in software, they were writing software concurrently with the mainframe development. This group of engineers provided an objective perspective on product quality (one likely shared by third party software vendors) that the design team could not provide. They were also among the major testers of 'polls', hooks in the operating system software for allowing the machine's functionality to be enhanced, or overridden. Although their primary interest was not in testing the operating system, the application engineers did discover bugs in the software and in the documentation.

Additional product testing was accomplished through documentation review. The Quality Assurance, Technical Publications, and Marketing departments read the ERS to acquaint themselves with details of the HP-71 function set. Application programmers read the IDS to interface to the operating system. And the design team reviewed every page of the HP-71 Reference Manual and the HP-71 Owner's Manual to ensure their accuracy. In each case, discrepancies were uncovered between the documented and actual operation. Most of these discrepancies were errors in documentation, but several were implementation problems.

The Quality Assurance Department created a bug tracking database that used a program called STARS (Software Tracking and Reporting System) for bug submittal, prioritization, categorization, and monitoring. STARS also provided report generation so that bug rates and problem areas in the operating system were readily identified. STARS was a

valuable source of information. Since it handled enhancement requests as well as bug reports, explanations in a STARS report could vary from the detailed text of a bug fix to the reason why an enhancement request was denied.

Whenever a bug report did result in a code change, a formal procedure was followed to guarantee that each version of the operating system was better than the previous one:

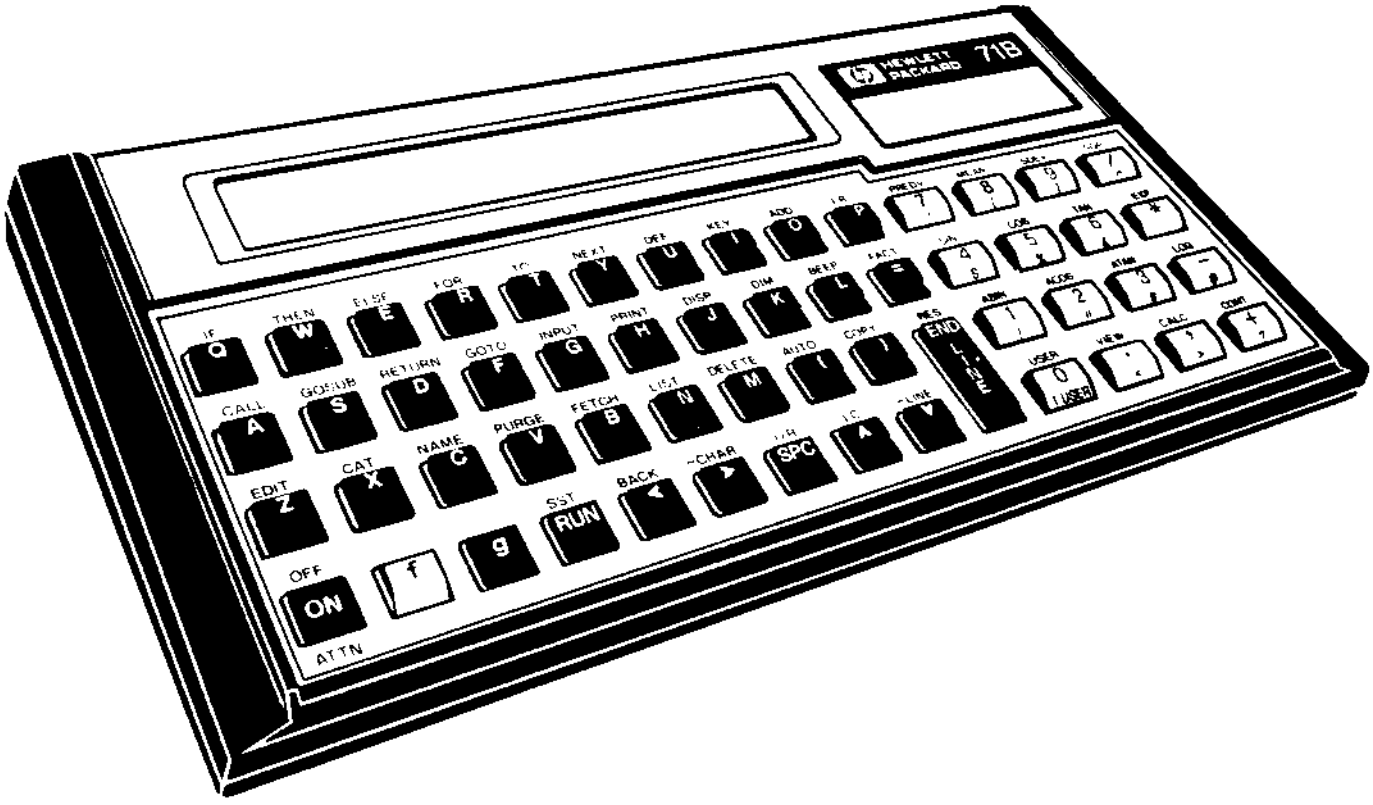
1) The code change was documented by the author so that in the event that it caused other problems, the code change could be modified, or at least undone.
2) The author created a test version of the operating system, consisting of the current version with the code change.
3) Another member of the design team verified that the reported bug was fixed with the test version of the code, that all other affected paths were also tested, and no new bugs were created.
   (This was facilitated by any available regression tests.)
4) The code change was signed off by this verifier.
5) The code change was added to the current version.

Initial testing uncovered bugs at a faster rate than they could be fixed. Ultimately, this was found to be a positive phenomenon since it was discovered that fixing individual bugs as fast as they were reported, frequently led to bandaiding problems. Often, letting bug reports accumulate revealed a more complete characterization of the problem, leading to a more educated and cohesive solution.

Of all the activities to ensure quality, one program stands out as the most effective for maintaining the morale of the design team during the potentially stressful period of functional testing. When the design team had fixed all known bugs, and felt confident about the operating system software, a program was implemented whose purpose was to increase the number of functional testers, provide additional motivation for finding bugs, and to make bug finding a celebration, even for the responsible author; after all, for every bug found pre-release, there would be one less bug our customers would find for us. The HP-71's code name was TITAN, and the program name was TIPSY: TITAN Incentive Program for a Perfect Software Yield. TIPSY raised a challenge to the Lab and Quality Assurance departments, that for every bug report submitted requiring a code change, the submitter would receive a credit. Credits were redeemable only in the company of a design team member, and were good for everything from after work drinks at the local bar to HP cafeteria items. (Submitters choosing the first option were also eligible for a ride home). The program maintained comraderie among the team members, and between the team and other functional areas. By promoting a healthy sense of competition, TIPSY expanded the circle of functional testers for the product, and at the same time kept the bug finding process light. Morale was high, while functional testing and bug finding was maximized prior to product release.

Throughout product evaluation, the Quality Department functioned in a support role, not as an adversary. They worked closely with the lab engineers, establishing evaluation tools and assisting in testing. Before final code was released for ROMs, engineers from both departments attended a Risk Assessment Meeting to review the testing effort and compare product performance against the completion criteria. The decision to release code was a mutual one.

The HP-71 achieved its quality end, but only because of a quality process: Various development tools increased productivity. The detailed internal documentation helped everyone work effectively toward the same goal. And, evaluation techniques made the adopted completion criteria achievable and measurable. Ensuring quality in the HP-71 certainly was a learning experience, not only in terms of what to repeat in the future, but also in terms of pitfalls to avoid in future products.


HEWLETT PACKARD 71B
ON
ATTN
RUN
SPC
END LINE

```
**********************************************************************
**********************************************************************
**
** Name:     xxxxxx  -  title
**
** Category:   xxxxxx
**      CONFIG =  System Configuration Utilities
**      DCMUTL =  Decompile Utilities
**      EXCUTL =  Execute Utilities
**      FILUTL =  File Utilities
**      FNEXEC =  Function Execute
**      MATH   =  System Math Functions
**      MTHUTL =  System Level Math Utilities
**      PARUTL =  Parse Utilities
**      POLL   =  Poll interface description
**      STDCMP =  Statement Decompile
**      STEXEC =  Statement Execute
**      STPARS =  Statement Parse
**      TIME   =  Time and Date Utilities
**      VARMGT =  Variable Management
**
** Purpose:
**      xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
**
** Entry:      xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
**
** Exit:       xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
**
** Calls:      xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
**
** Uses.......
**  Exclusive: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
**  Inclusive: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
**
** Stk lvls:   xxxxxx   (max used by this routine and its calls)
**
** NOTE:          (remove any item below this point if not needed)
**      xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
**
** Detail:
**      xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
**
** Algorithm:
**      xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
**
** History:
**
**    Date      Programmer              Modification
**  --------   ----------   ----------------------------------
**  MM/DD/YY   xxxx         xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
**
**********************************************************************
**********************************************************************
```

```
**********************************************************************
**********************************************************************
**
** Name:     POKEP   -  POKE Statement Parse
** Name:(S) STRNGP   -  Parse of Mandatory String Expr
**
** Category: STPARS
**
** Purpose:  POKEP parses POKE statement.
**
**           STRNGP parses a mandatory string expression
**
** Entry:    D(A) = (AVMEME)
**           D1 points to input stream
**           D0 points into output buffer
**           2 entry points:
**           1) POKEP  - D1 past POKE keyword
**                       D0 past tPOKE
**           2) STRNGP - D1 pts to alleged string expr
**
** Exit:
**           Valid parse =>
**            Return with carry clear
**            P=0
**            POKEP entry:
**             D1 points past syntactically correct stmt.
**             POKE tokenization written to ouput buffer.
**             D0 points past POKE tokenization.
**            STRNGP entry:
**             D1 points past string expression.
**             String expr tokenization in output buffer.
**             D0 points past string expr tokenization.
**
**           Else error exit
**
** Calls:    OUT1TK, STRGCK, COMCK+
**
** Uses:     A-C,D(15-5),D0,D1,R0,R1,R3,S0-S3,S7,S11,
**           FUNCD0
**
** Stk lvls: 5
**
** Detail:   POKE <string expr>,<string expr>
**
** History:
**
**    Date      Programmer   Modifications
**  --------    ----------   -------------
**  05/11/83    S.W.         Replaced call to COMCK1
**                            with call to COMCK+
**
**********************************************************************
**********************************************************************
```

EVALUATION TECHNIQUES

- **Design Review**

  Essential for complex algorithms

- **Code Review**

  Essential for code difficult to cover with functional testing

- **Functional Testing**
  - **Binges for concentrated focus**
  - **Regression tests for code coverage and *fixed* bugs testing**

- **Documentation / Manual Review**

- **Beta Test Site**

Sample STARS Report

SR# 0039000182 **Pri:** 3

Classification: Known Problem **Status:** Signed Off

Display delay doesn't always work after programmatic BYE.

DELAY BYE/OFF FUNCTIONAL TESTING

Submitter: B.S. **Submitted:** mm/dd/yy
Lab engr : N.M. **SR updated:** mm/dd/yy
Release : mm/dd/yy

SUBMITTER TEXT:

REPEATABLE? : Y
OPER. MODE : RUNNING
DESCRIPTION : Display delay does not work after ATTN wakeup from programmatic BYE. To repeat:
10 BYE / 20 DISP "A" @ DISP "B"
RUN, then wake up with ATTN. Sometimes also true with timer wakeup.

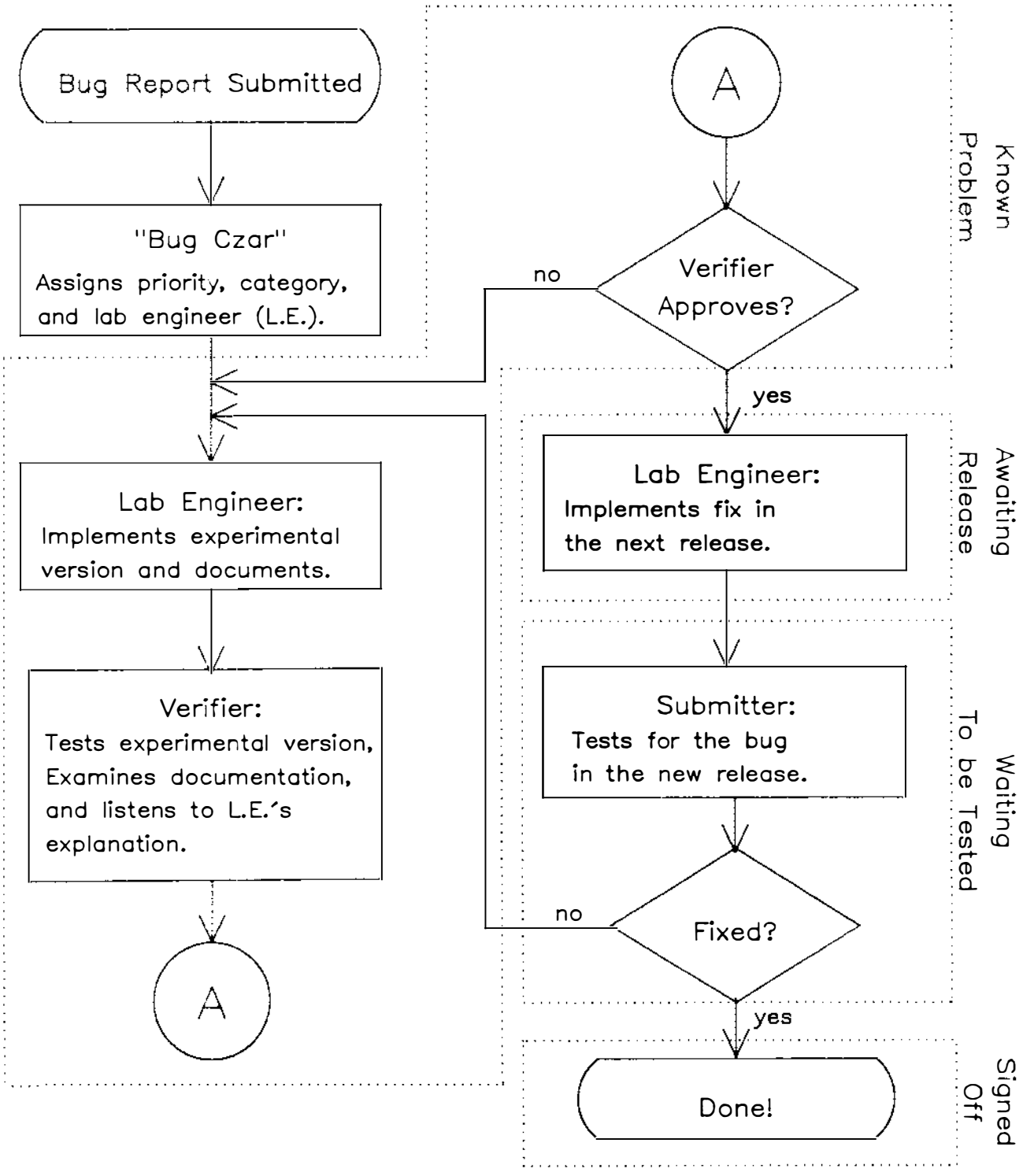
FIX INFORMATION TEXT:

Problem was with execution of linefeed delay. Linefeed delay did not occur if first key in keybuffer was ATTN, even if buffer pointer said keybuffer was empty. Solution was to look at ATNFLG instead of buffer.

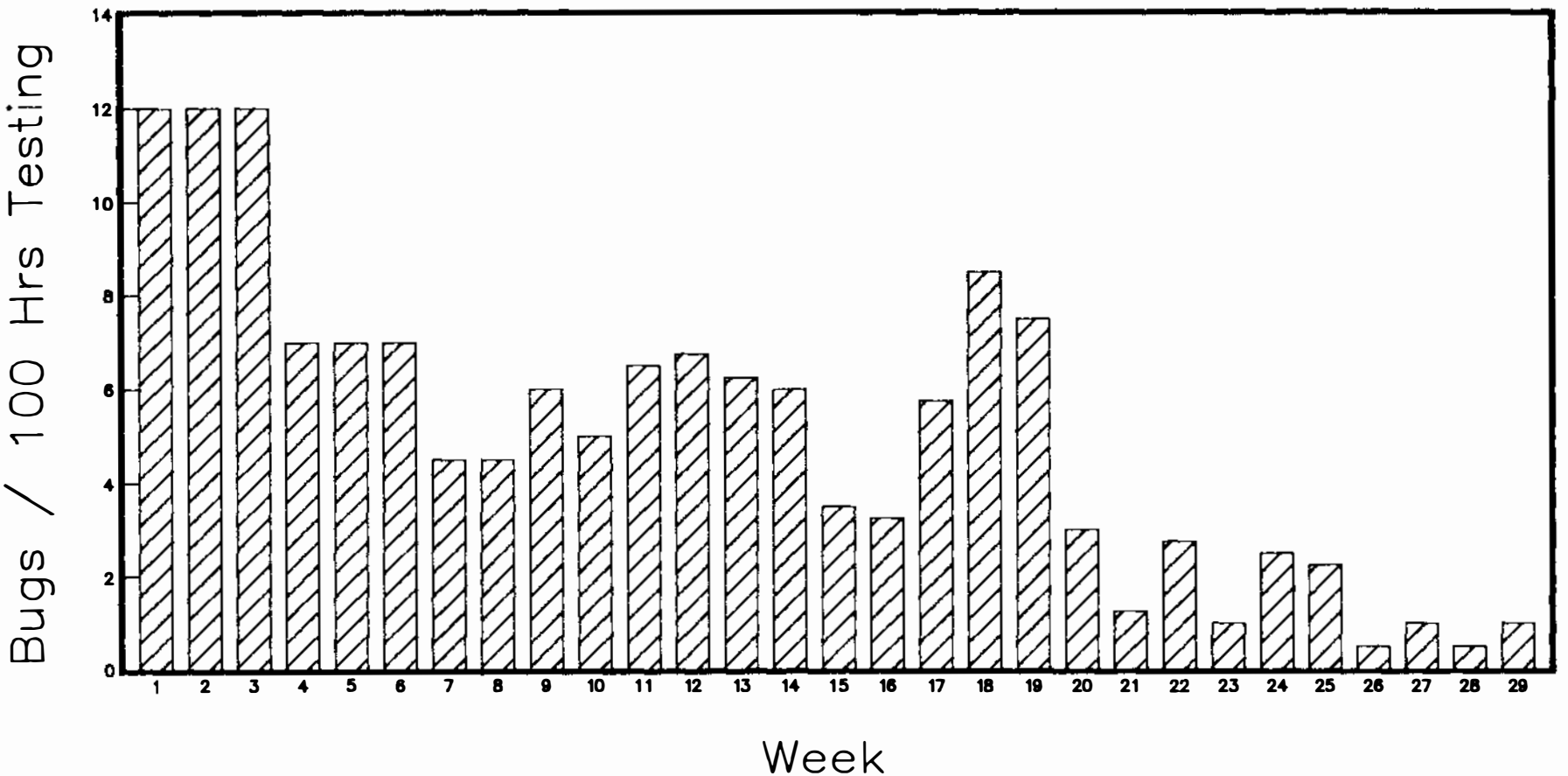
PRIORITIZING BUGS

Priority		Description
1	Critical	Machine Crashes
2	Serious	No workaround, but needed
3	Guarded	Workaround exists, but needs fix
4	Cosmetic	Nice feature, but may not install
5	No Value	User misunderstanding

Bug Life Cycle / Code Control



TITAN BUG RATE



Susan Wechsler graduated from California State University at Long Beach with a BA degree in mathematics in 1979. At Hewlett-Packard, she was a major contributor to the design and implementation of the operating system for the HP-71B Computer. She recently authored a paper in the July 1984 *Hewlett-Packard Journal*, "A New Handheld Computer for Technical Professionals".

Managing Independent Test Organizations

by

Russell R. Sprunger, Vice President of Production
Graphic Software Systems, Inc.
25117 SW Parkway
Wilsonville, OR 97070

In today's fast paced microcomputer world, getting a quality product to market in a timely fashion can make the difference between long term success of a company and mere survival. How to adequately test such products is a crucial consideration in the development process. Independent hardware testing labs have existed for many years. However, independent software testing labs have become a reality only in the past few years. Independent testing laboratories can offer professional services, enable effective project cost management, and provide experts not normally available in developing organizations. This article will outline the benefits of using outside independent test organizations and describe effective utilization of their resources.

In the summer of 1983, Graphic Software Systems, Inc. (GSS) was preparing for the Fall introduction of a series of new graphic software products. At that time, GSS had devoted most of its resources to developing software products and had not established a formal testing organization. The challenge then was to develop and implement a plan for testing these new products. The testing had to be prompt, comprehensive, inexpensive and reliable under repetitive use.

At the base of this product series to be tested was a product called **GSS-DRIVERS**, a device-independent graphics extension to MS-DOS (tm) and Unix (tm). Therefore, the proposed tests needed to exercise a wide variety of functions taking into account the device-dependent attributes of each device. Additionally, this variety of test functions needed to address both the devices then supported by GSS, as well as software developed in the future to support additional graphics devices. Since GSS products are designed to be usable on multiple operating systems, the tests also had to be portable between operating systems.

Having outlined the goals and constraints for software testing, GSS began investigating ways to implement the tests. One possible choice was to hire full time employees, thereby establishing the foundation for a Quality Assurance group within the company. Other possibilities included using qualified customers, or outside contractors.

Finding qualified people to immediately staff a new department was a task that presented more difficulties than advantages. Using qualified customers had provided valuable usability feedback, but historically failed to result in comprehensive, well documented, and portable testing. Therefore, GSS began searching for an established group capable of satisfying our requirements. We began our search with several organizations referred to us by our colleagues.

The next step was to determine how to qualify and select the right organization. The initial qualification process consisted of contacting each organization, requesting company background, specialization, and general business terms. This process resulted in two organizations whose advertised business was product testing and/or installation, each of which had some expertise in testing graphics products.

Next, each of these organizations was visited to evaluate the personnel, previous work, and the work environment itself. This activity also involved discussing the product(s) to be tested and providing the test organizations with enough product material to allow them to provide GSS with a formal quote. From this experience, we determined that the formal quote should contain at least the following items:

1. Detailed description of work

    a. Overview
    b. Number of tests
    c. Language of implementation
    d. Test methodology

2. Schedule

    a. Start and end date of project
    b. Deliverables by developer by date
    c. Deliverables by contractor by date
    d. Number and location of meetings

3. Cost

    a. Personnel
    b. Travel
    c. Equipment (hardware and software)

4. Business terms.

    a. Payment terms
    b. Ownership of deliverables.

Following on-site evaluation and receipt of quote, GSS selected the International Bureau of Software Test (IBST) of Sunnyvale, California.

Once the contract had been awarded, technical and management representatives were designated and regular meetings were scheduled. Weekly meetings were determined to be appropriate on projects of up to three months duration. Due to the physical distance between GSS and IBST (approximately 600 miles), many of these meetings were conducted over the phone. On-site visits to the test location were conducted every three weeks.

As with any testing effort, it was important to define an effective means of addressing each product error and of updating the test organization with new releases of the product. Therefore, IBST provided weekly reports, in addition to the meetings, which included project status and detailed software error accounting. Detailing each error and then following up to insure that each error had been addressed in subsequent releases of software, was a significant task both in its importance and in the resources expended.

Following completion of the project, GSS had a complete documented set of tests that could be run regularly as new installations occurred.

Using outside testing resource has revealed several benefits:

1. Easy management of the highly variable demand for human resources

2. Well documented testing of "first time user" situation

3. Complete, documented test suites

4. Good user interface feedback

5. Verification of product conformance to stated documentation

6. Verification of product conformance to standards

The earlier in the product's development that the testing organization can be involved in the project, the greater the potential that design errors can be found before they become ingrained product deficiencies. Simple economics dictate that

correcting problems at the design stage is much less expensive than having to rebuild the product because testing was relegated to the final stages of product development.

An additional advantage of earlier involvement of the testing organization is that they may have the time to learn the product thus enabling them to be more expansive in the design of their testing procedures, rather than simply insuring that the product conforms to the primary level of usage suggested by the documentation. Providing the organization with the opportunity to perform in-depth testing permits optimization of the testing process.

The following steps outline the process of using outside organizations to perform product testing:

Step 0 Designate a project leader

Step 1 Determine what you hope to obtain by using an outside organization

Step 2 Locate available outside testing organizations

Step 3 Determine the business procedures of these organizations

Step 4 Specify what you need to have tested

Step 5 Obtain quotes for services

Step 6 Evaluate test proposals on the basis of:

a. Cost
b. Manpower applied to task
c. Types of testing performed (validation, stress, destructive, ease of learning, ease of use)
d. Length of test cycle
e. Amount of retest
f. Test deliverables
g. Completeness of test cases

Step 7 Negotiate contract

Step 8 Begin contract

Step 9 Monitor project by:

a. Setting weekly schedule for multi-month projects; monthly for multi-year projects
b. Evaluating test procedures
c. Conferring on the acceptance of project deliverables
d. Determining whether to conduct face-to-face vs. long distance monitoring. Face-to-face is preferred on at least these occassions during the contract:

    1. at the start of the project,
    2. at the acceptance of the detailed test plan,
    3. at the delivery of each major test component, and
    4. at the presentation of final test findings.

Step 10 Study test deliverables

To summarize, outside test organizations can provide many benefits. To effectively use these valuable services, the product developing organization should accurately assess their testing needs, involve the testing organization as early as practical, and demonstrate proficient project management skills, particularly the ability to manage projects via long distance.

- END -

BIOGRAPHY

Mr. Sprunger is a co-founder and Vice President of Graphic Software Systems, Inc. He has been involved in all aspects of the company and has specialized in the areas of product testing, documentation, and production.

Mr. Sprunger also has broad experience in interactive computer graphics. While at Tektronix, he was a principal developer of PLOT 10 IGL (a SIGGRAPH '79 Core Compatible interactive graphics package). He has served as the Tektronix consultant for graphics software to international customers. While working at Argonne National Laboratory, he was responsible for graphics programming on numerous environmental modeling and monitoring projects.

Mr. Sprunger is also a member of the National Computer Graphics Association (NCGA), Association for Computing Machinery (ACM), and SIGGRAPH.

6 Automated Systems and Communication Techniques

Mr. Martin Kennedy/Ms. Cary J. Lamoureux; Tektronix
"Communication Techniques for Better Design"

Mr. Pete Fratus; Hewlett-Packard
"Automated System Testing"

Ms. Donna J. Wengrowski; Mentor Graphics
"An Automated Problem Reporting System"

Communication Techniques for Better Design

Martin Kennedy
Cary J. Lamoureux

Microprocessor Development Products Group
Tektronix

Communication Techniques for Better Design

ABSTRACT

This paper will show that a key to designing quality into a software product, is the early use of a controlled software requirements communication technique, to transmit design detail and functionality, to all the groups involved throughout the design cycle of a product. From design to evaluation, manual writers, marketing, and manufacturing, the need for correct, timely and up-to-date information, usable by each group is critical to the smooth development and quality of the final product.

Assuming a correlation between better quality in the final product, with more complete and timely software requirements specifications, then a common controllable design documentation system becomes critical to the product development success.

A deterrent to communication between groups, is that each group requires varying details, and functionality from design, to prepare and perform their product development tasks.

Currently the most common form of this communication is the Software Requirements Specification (SRS), a text document of the product detail description, written parallel to the design encoding.

More efficient and productive is a tool which would be a part of the early design, aiding in design software analysis. And producing a more common and complete communication of the product, for each development groups usage, not as a parallel effort, but as part of the product design.

Communication Techniques for Better Design

Martin Kennedy

Cary J. Lamoureux

Microprocessor Development Products Group
Tektronix

One of the greatest deterrents in developing a quality software product is effective communication of product information during the project.

Throughout the development of a product, many "groups" must communicate varied degrees and amounts of data, in as concise a manner as possible, for the project to complete with the highest degree of success.

For the duration of the paper, the definition of a "group" is those persons who are highly specialized in one area of a product's development phase.

These groups may be generally divided into the following titles or areas of responsibility depending on the project team organization. (See Figure 1) These include:

Design which coordinates the preliminary specifications, produce the source code of the product, schedule project resources, and archive the finished product.

Evaluation which verifies the product package, possibly review and/or report on the functionality, the customer interface, and the product manuals.

Technical Manuals group which produces documentation for the installation, user command syntax, maintenance, and service or sales literature for the product.

Additional groups are Manufacturing, Project Management, and Marketing and/or Field Support.

All of these groups depend on the ability to communicate product information and functional description to carry out their

tasks in a timely, complete, and efficient manner. The key to this communication is to accurately describe the product and its attributes early in the design phase and in terms that are common to as many of the groups as possible.

During the last ten years much effort has been invested to produce tools which better communicate system design abstractions and program models. Examples of such tools are compilers and assemblers, which allow textual descriptions to be transformed into actual source code. They allow "readability" of the code and its intended function, for later analysis by other groups. They, also, permit separation of processes and their descriptions by "function" or "procedure" units.

Still, the most general tools used today to model systems and functional descriptions are textual tools, which create and maintain the product documentation. Some examples of these are special formatting packages which can include "standard" titling, and forms manipulation and electronic mail systems which are used to send evaluation reports, design updates, and project information. In addition special programs exist that enforce a version control of textual documentation and source code during a project. Documents such as the SRS, Software Design Descriptions (SDD) and User Interface Syntax, are all written using tools that perform primarily textual media manipulation and control.

However, a natural difficulty exists in describing a dynamic system model or abstract process in a concise manner using exclusively narrative text as the communication media. Too often the product specification created by these text tools has created "Victorian novels" , which must be read in their entirety to understand any point in the functionality. Because large volumes of text are difficult to digest, this has led to incomplete, ambiguous, and even self-contradictory specifications.

The practical problems that arise from the use of the textual media are varied, depending on the particular groups that are attempting to inject in or extract needed information from the product documentation. A few of the more common outcomes are:

A lack of a common language between the product groups. Group buzz words, new definitions, and assumptions of functionality begin to invade the early modeling and coding. The confusion of terms can and has led to portions of code to interfacing improperly at implementation time.

The lack of an early, usable model in the design cycle leads to programers and designers becoming very "tunnel visioned" in their view of the project and product. Knowledge of the entire project, or a functional overview, becomes fragmented quickly because a good overall picture or model of the system does not exist.

The effort placed into the creation and updating of the project design documentation becomes the most time consuming task of the project. Typically, the true description of the product functionality is done as a parallel effort to the coding phase of the product, not during the design phase. This leads to major revisions of the documents as the product comes closer to completion, which results in more money spent and schedules not met. The later an error is found in the life cycle, the more it costs to fix.

It follows that other product groups that have relied on the preliminary specifications lose some degree of the effort placed into test cases, drivers, manual information, and possibly even scheduled resources. This leads to the ever popular "frozen specifications", for which there is no way to truly stop the needed updates as the project progresses. Reduce them YES; stop them NO!

Considering these points, the critical limitation in using the current tools is the difficulty in translating system models of a product, which are dynamic, with tools and a communication media that tend to be static.

Recently, a number of methodologies have been developed to improve the process of system modeling and front-end design. Some of the better known are the Yourdon and DeMarco methodology for Structured Design and Structured Analysis. However, efficient tools that offset the textual problems needed to implement and maintain the methods have been lacking. Such tools, when implemented, should:

1) show "logical system models" and system parameters in a media form usable across a number of product groups.

2) permit creation of an early version of the requirements, thereby providing other groups with the opportunity to interact with the specifications documentation at design time, not during the coding period.

3) reduce the time spent on producing documentation, by making it an active creative portion of the design cycle, instead of a parallel effort during the coding cycle.

4) allow easy maintenance of the project documents and models thus enhancing the methods of tracking the development progress and the configuration management of the product.

5) model large, as well as medium and small

product or system models.

6) contain automated functions which will update support documentation as refinements are made to the design model.

7) use graphic representations.

Such tools do exist. One is an automated structured analysis (SA) package currently being used and productized by Microprocessor Development Product Division at Tektronix, Inc.

The SA tool is a set of programs that allow the user to create, evaluate modify, and display system models as documents. Each SA document permits access and modification using tools, such as text and graphic editors.

The SA tools will create and modify three types of documents: Data Flow Diagrams (DFD), Data Dictionaries (DD), and Mini-Specifications documents (MS).

Data Flow Diagrams are a graphic representation of all or part of the system being modeled.

Figure 2 shows the conventions of data flow diagrams. The DFD consists of a title block and any number of five types of data elements.

A title block contains the figure number and the title of the DFD. The title is the name of the parent process, except in the case of the top-level DFD, which is named "DFD0". In addition, the title block contains the user ID of the last person to modify the DFD and the date of the last modification. The editor will automatically update the date and user ID, whenever a DFD is modified.

The DFD five types of data elements are:

Process - Processes are represented by the bubbles. They indicate a transformation of input data into output data. Each process is automatically assigned a number and a name. The process name and a number are then used as the title and figure number of a child DFD.

File - Files are represented by one or two horizontal lines and a name. Files store the data produced or used by processes. Two lines indicate that the file originates at this "level" of the DFD. A single lined file indicates that it is used at the current level, but is defined by a parent or higher DFD.

External - Externals are data origins or destinations that are completely outside of the system being modeled. They are represented by rectangles. Externals can, also, be used to

represent data gates, by substituting "-<" or ">-" for the name. The data gates show parallel decomposition or synthesis of data from a single data flow. Note the data flow decomposition of the "product_input" data flow, into two other distinct data flows.

Boundary Point - The boundary point is used to identify a point where data flows enter or leave the current boundaries of a DFD. Thus, a boundary point is the data interface with a parent or child DFD.

Data Flows - The Data Flows are represented by lines with arrows. The data flow has one origin and one destination. They can be named, but when used with a File, the name is redundant. Data Flows are the pipe lines along which information travels.

The Data Dictionary contains the definitions of all of the named data flows and data stores used in the Data Flow Diagrams.

It consists of individual definitions of data flow and data store names used in the DFDs, in the MSs and in the DD. The definition syntax combines primitive data items and operators to denote equivalence, sequence, selection, and iteration. Other symbols are used to identify comments and literal strings. The data name "TO-BE-DEFINED" is reserved by the SA tool. This data name indicates that the item has been added to the data dictionary by its appearance in a DFD, but has not been defined. The definition can be completed later as more project data is found or verified. Figure 3 is the data dictionary of the data flow diagram in figure 1.

Mini-Specifications describe how input data is transformed into output data at a particular point in the system. That point is a functional primitive - a process that cannot be broken down further into other data flow diagrams.

It contains a title block and the description of a process. The description is the body of the mini-spec and can be written in any format. Typically, the format is structured English, decision trees, or decision tables. For documentation control, the title block contains the title of the MS, which is the title of the parent process, the user ID, the creation date, and the figure number. Figure 4 is the mini-specification of the process "Implementation", in figure 2.

The SA tool set consists of the graphics editor, document evaluation tools, and document update or fix tools.

An interactive graphics editor, is used to produce the data flow diagrams. It understands the unique character of each of the data elements within the data flow diagrams, and prompts for the needed information as the diagram is built.

The "evaluation" programs of the SA tool help find certain types of errors, depending on the SA documents type being

evaluated. For example the evaluation programs will check the DFDs for; valid file types, existing parent, read-only or write-only processes or files, disjointed data items, correct file levels, and data conversion with the parent. For the Data Dictionary file, such aspects as; Valid syntax, TO-BE-DEFINED entries, Entries used in DFDs, Entries defined more than once, are all evaluated for.

The "fix" tools in the package, can be used to create SA documents and correct errors that relate directly to the SA methodology. If there is only one way to correct an error, the fix tools can correct it automatically. Like the "evaluation tools", the fix tools operation depends on the type of document being fixed.

Additionally there are special "display" tools that allow the conversion of the DFDs to a screen or print formats.

With the SA tools, preliminary system models can be created earlier for group reviews and feed-back. Actual design reviews can be held long before the "code reviews" are brought into play, thus work out designs and bugs much earlier in the product development cycle.

Also the SRS document is created by "pasting" together the Mini- Specifications of each of the process primitives. The documentation is now truly a part of the product design cycle.

As the project continues, the finalized SA tool documents will serve to reenforce an overview of the functions and data transformations of the product. This will help when the SDD and coding portion of the project begin, and common structures, or call parameters are needed to produce "integratable" software.

Lastly, the areas of a design which are still undefined, are quite apparent to any group referencing the initial design SA documents. The data dictionary entries explicitly state the need for definition. This helps to prevent the loss of early resource usage, and keeps management well informed on the advancement of the project in the early modeling stages.

Granted that the tools that have been described here do not answer all of the needs for better quality software designs, the initial results obtained at Tektronix, are very exciting. The automation of the Structured Analysis methodology has created a whole new approach of how to represent system and program models, resulting in a much clearer communication of early model problems. More project groups, are involved earlier in the design. And the better understanding of the design helps to to produce better quality software.

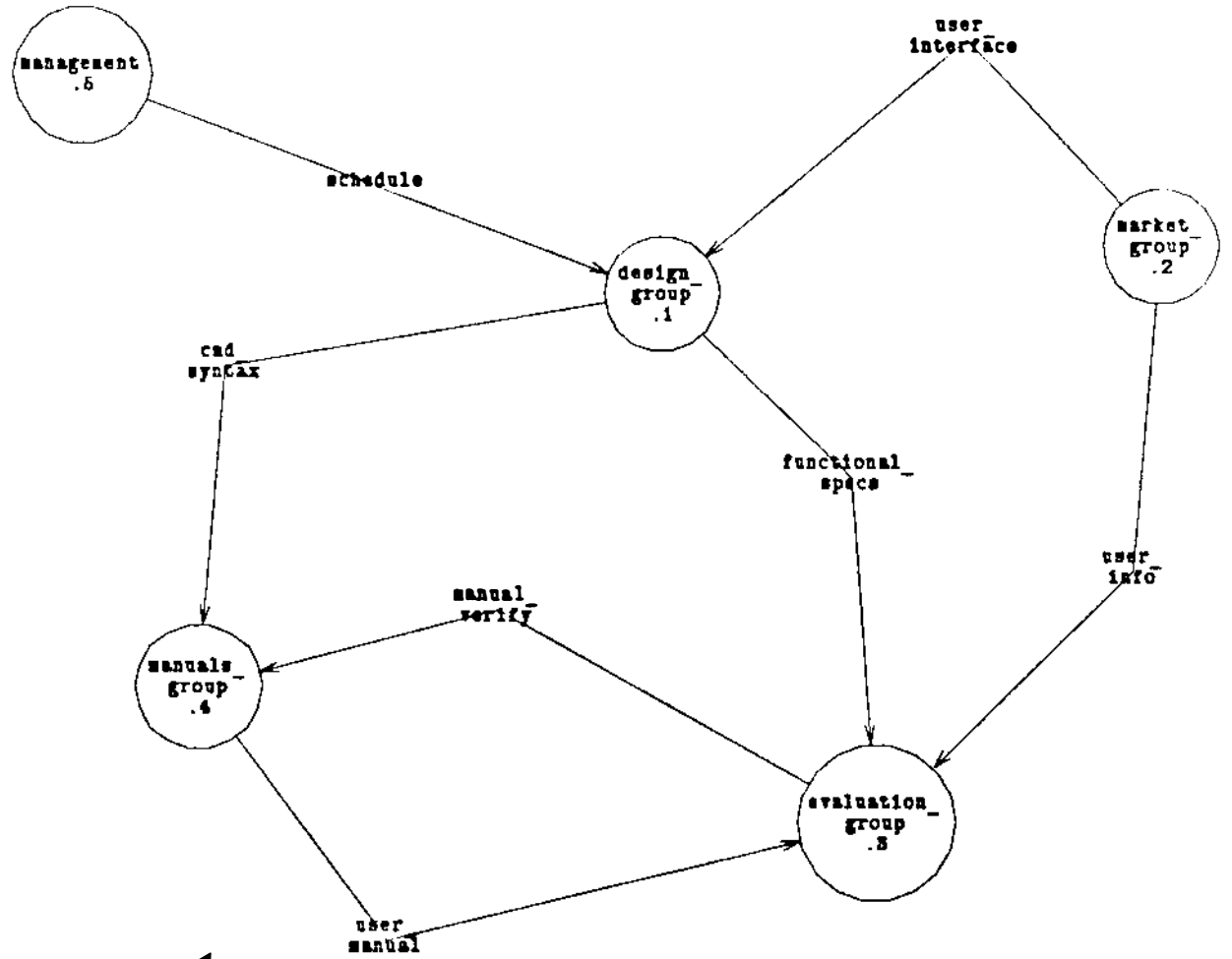


figure 1

DFD 0 - groups

7/ 9/84-martink

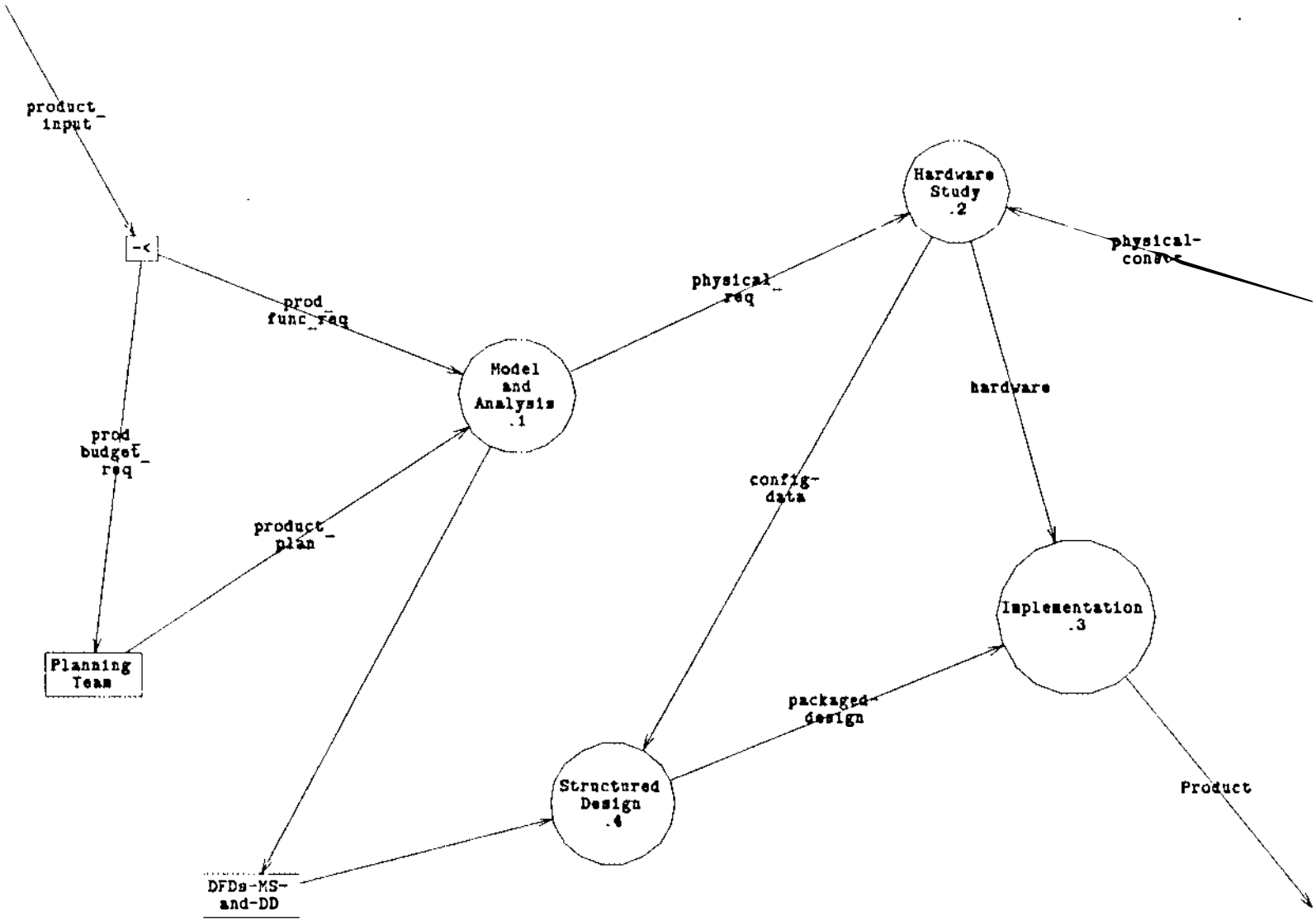


Figure 2

7/19/84-martink

DFD 1 - Early Part of Software Life Cycle

```
*
     Data Dictionary example
     Comments are bounded by asterisks
*

DFDs-MS-and-DD = [ DFD + MS + DD ]

Product = [ hardware | software | hardware + software] + {documentation}

config-data = TO-BE-DEFINED
hardware = TO-BE-DEFINED
physical-constr. = TO-BE-DEFINED

packaged-design = [ 1{structure-chart} + 1{module-description} + TO-BE-DEFINE

physical_req = TO-BE-DEFINED

prod_budget_req = $25,000.00

prod_func_req = TO-BE-DEFINED

product_input = prod_budget_req + prod_func_req

product_plan = prod_schedule_req + prod_physical_req
```

Figure 3

Mini-Spec #3 Title: Implementation
Designer:
martink
 7/ 9/84

Extract DFDs from the DFDs-MS-and-DD file.

Identify the central transform in the set of detailed design.

Produce a first-cut structure chart.

Revice first-cut structure chart.

Write module descriptions for all elements in structure chart

Using config-data, create a package-design

3

Figure 4

BIOGRAPHY

Cary J Lamoureux

Cary joined Tektronix in 1982, after graduating from Brigham Young University, starting in the Microprossor Development Product Group. Her work has included extensive evaluation experience with both Tektronix high level language compilers and language directed editors. She currently is with the Logic Design Systems Group designing simulation software.

Martin Kennedy

Martin joined Tektronix in 1982, after working with North American Rockwell, and receiving a second degree in computer science from the University of Wisconsin Madison. He started with the systems evaluation team of the Microprossesor Development Products Group, and is currently a project leader in hosted software interface designs.

AUTOMATED SYSTEM TESTING

Pete Fratus
Hewlett-Packard, Computer Systems Division

Overview

Software research and development at the Computer Systems Division (CSY) is done according to the Software Product Life Cycle (PLC). The PLC calls for a phased development with the five standard areas of Investigation, Design, Testing, Implementation, and Release.

During the Design Phase, a test plan is written to specify how the software modules and system will be tested. From this plan, test programs are written to test the reliability and functionality of the product. The two types of testing done are the Reliability Ring and the Certification Umbrella. Reliability testing is a stress test of software in which all types of functions are tested simultaneously over a period of time. Certification testing is a single pass through all functions.

On any release of the operating system, Multi-programming Executive (MPE), there are over 800 jobs containing over 10,000 tests. These are run against MPE and the subsystems. In the past, operators worked around the clock, running each job and checking the resulting output for errors. Although this method did get all the tests run, one was never sure that all the errors were found and reported. Therefore, the decision was made to develop a series of tools to automate the testing process, in the hope of catching more errors earlier in the system testing process.

The first program to be developed was the MONITOR. This program is script driven and controls the running of all test jobs. Error found during the testing process are automatically logged to a file. We have written a set of software procedures to allow the tests to report the code location of an error, supply explanatory messages, and provide 'soft' aborts. In addition, the system error traps are armed by the monitor program to catch any abnormal aborts of the tests.

With the controlling program written, the next step was to write a program to save the spooled output for later analysis. This program, called the ARCHIVER, is started by the MONITOR and copies all spool files to a tape. With the MONITOR and ARCHIVER in place, we began to catch a greater number of errors earlier in the testing process.

The next problem was who was going to read through 80,000 lines of spooled ouput looking for inconsistencies. To solve this, a spool file comparison program was written. Checksums are calculated on a known good run of the tests and recorded into a master checksum file. After the completion of each section of

the tests, the ARCHIVER starts the spool file comparison program, SPCOMP, and passes each completed spool file to it for analysis. A checksum is calculated for the spool file and compared to the checksum master. Errors such as missing lines, additional lines, and lines that do not match are reported in detail and summary reports.

The automated testing has allowed us to catch a large number of defects in software that would otherwise have gone to the customer. Projects are now in progress to enhance automated testing for more functionality.

The Certification Umbrella

Certification Monitor.

This is the program CM, which is the driver program for the Certification Umbrella. The program handles the following via script files:

- Initiation of tests
- Monitoring of tests for completion
- Checkpointing test runs
- Archival of test results to tape
- Initiation of interactive tests on terminal devices
- Starting and stopping of the logging process
- Checkpoint restarts of a test run that did not complete

Commands File.

This is the file COMMANDS, which contains the script commands used by the Certification Monitor. It usually contains the following:

- An ID command, which specifies the MPE version, date, and time of the Certification Monitor run.
- EXECUTE commands to launch pre- and post-processing jobs needed by the Certification Monitor.
- USE commands, which specify which script files to use for the certification test run.

Script Commands.

Script commands are the mechanism by which the tester has specified, to the Certification Monitor, what the flow of the test processing is to be.

Script Files.

These are files which are used to specify to the Certification Monitor the sequence of processing for each specific product test.

Archive Program.

This is the program ARCHIVE, which is started by the Certification Monitor. The program handles the following via commands from the Certification Monitor:

- o Starting the spool file comparison program, SPCOMP
- o Starting SPOOK for spool file access
- o Copying of spool files to tape
- o Copying of test result files to tape
- o Purging of processed spool files

Spool file Comparison Program.

This is the program SPCOMP, which is started by the archive program. The program handles the following via commands from the archive program:

- o Comparison of the current spool file against the master checksum file.
- o Reporting of errors such as line compare errors, missing lines, and additional lines.
- o Logging of the elapsed time and cpu time of each job, and the reporting of any job that has run significantly longer then the average times.

QA Log File.

This is the log file QASWFILE, which is created from the Certification Monitor. The log file is accessed through the use of the QA procedures and is used to log the start of tests, the end of tests, and any errors found during the execution of the tests. This log file is accessed at the end of the certification monitor run to print out an analysis report of the test results.

Master Checksum File.

This is the file CHECKSUM, which contains the spool file information of the complete test library in the form of byte checksums. When a spool file is passed to the spool file comparison program, it is run through an algorithm to create the same type of byte checksums that are in the master file. With this information, the comparison can be made between the current spool file and the spool file from a known good run.

QA Procedures.

Logging to the QA log file and reporting of test results is done with the QA procedures: START, QAERROR, FAILSOFT, and LOGEND. These procedures are used by both the certification umbrella programs and the user written tests to allow automated test result analysis. These procedures are in the system library.

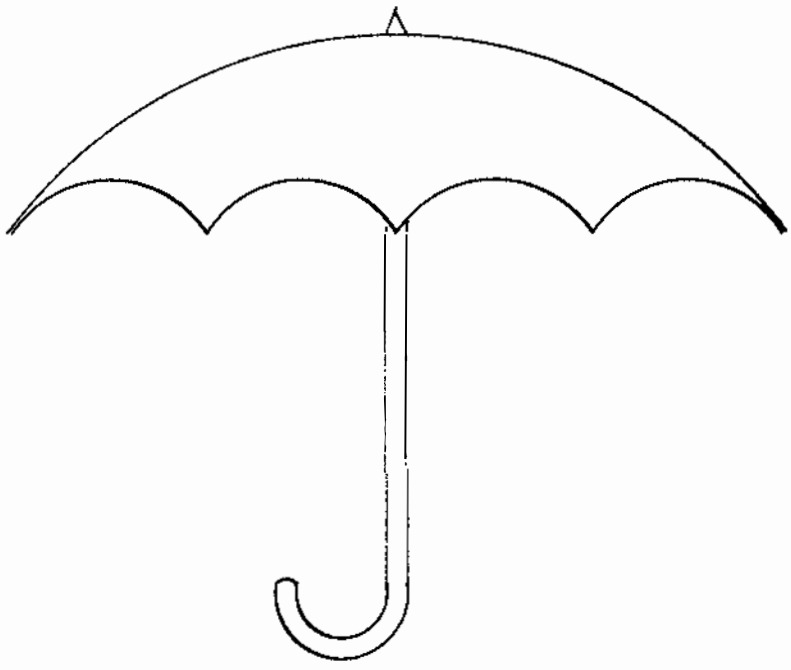
Bibliography

Software Product Life Cycle. Hewlett-Packard; SPLC.83.02 #5955-1756; February, 1983.

Customer Satisfaction Through Quality Software. Michael McCaffrey and Dan Coats, Hewlett-Packard; International HPIUG Conference, 1984.

Automated Certification Test Umbrella, External Specifications. Hewlett-Packard, 1983.

Certification Umbrella

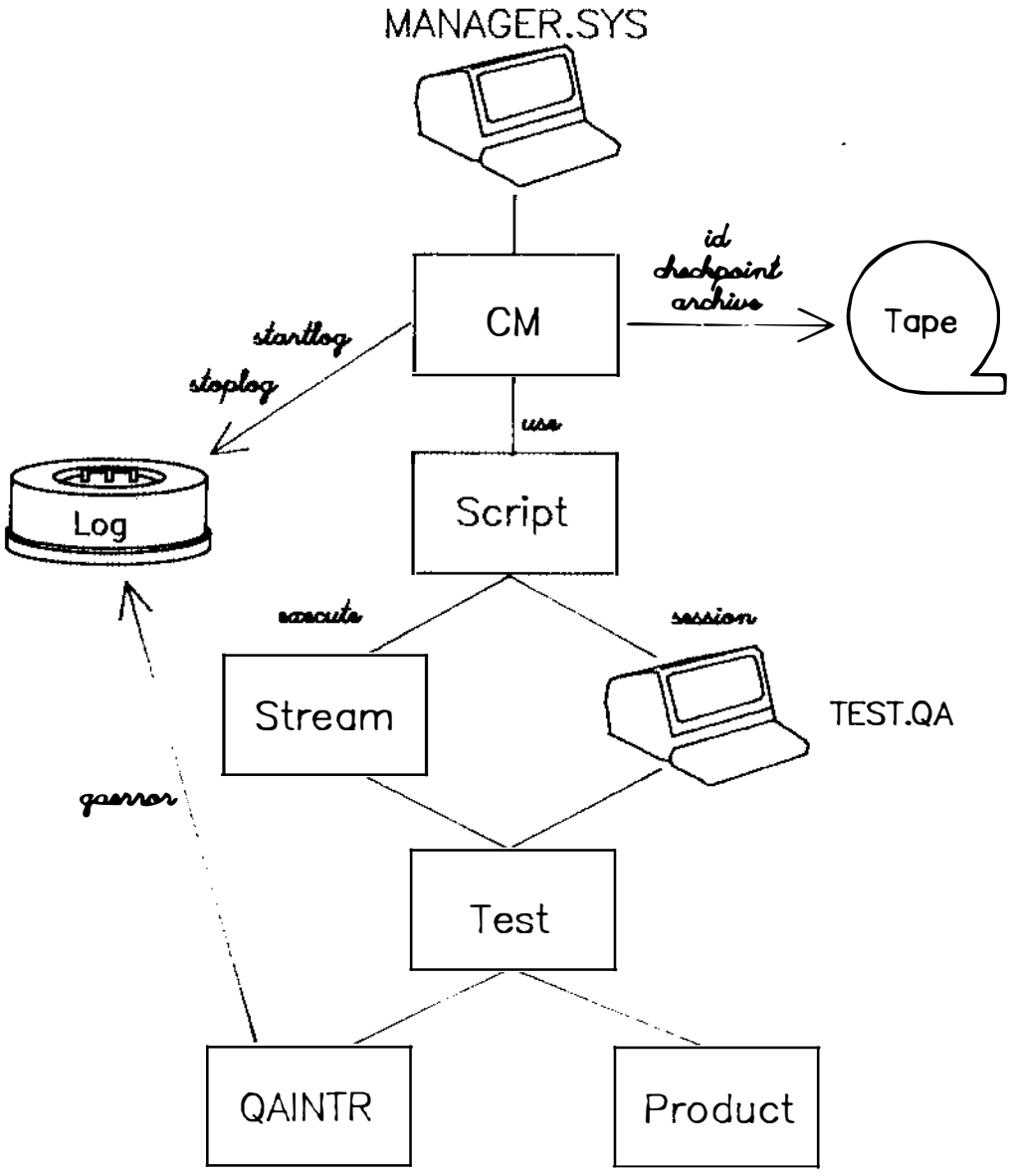


A testing tool for R&D engineers

for automated product testing.

cuhead

Umbrella Flow



cudlog

Umbrella Scripts

Control

USE	Switch script file
SESSION	Start interactive session
EXECUTE	Start test job stream

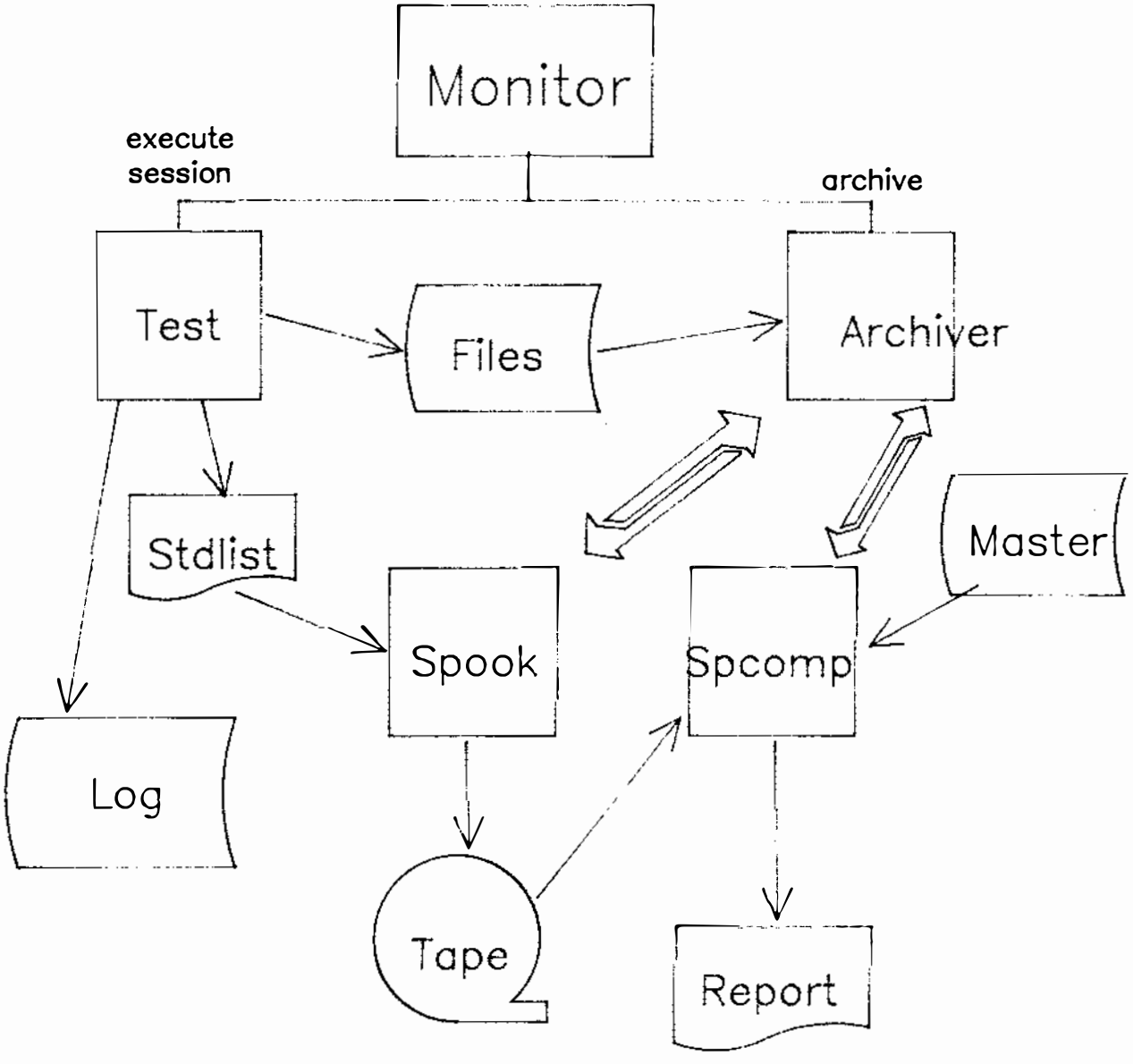
Logging

STARTLOG	Log beginning of test
STOPLOG	Log end of test

Archiving

ID	Get system identification
CHECKPOINT	Mark current test
ARCHIVE	Copy spool files to tape

Umbrella Processes



Biography - Pete Fratus

Mr. Fratus has worked at Hewlett Packard since 1966. His background consists of programming, applications design, computer assisted records management and data communications. His current area of responsibility is the production of tools that are used in software research and development.

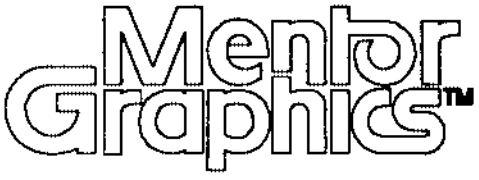
Mr. Fratus has had articles published in several computer magazines, and has spoken at several computer conferences.

An Automated Problem Reporting System

by

Donna J. Wengrowski

August 1984



Donna Wengrowski Mentor Graphics Corporation

An Automated Problem Reporting System

1. Abstract

In the real world of software and hardware there are bound to be lots of problems. What is the best way to handle the tracking of individual problems, from internal as well as external sources? How do we prevent reported bugs from "falling through the cracks"?

This paper will be a discription of the QA Problem Reporting Database Management System we have written and are currently using at Mentor Graphics Corporation. It will describe the process used in designing and implementing such a system to meet our needs. The life cycle of a problem report will also be explained.

Our Problem Reporting System consists of two Pascal programs that interface to an internal database management system. One program allows users to report, respond to and verify problems, as well as perform queries on the information stored in the database. The other program allows the QA department to control and monitor problem reports. The database may also be accessed directly through a friendly interface. This Problem Reporting System has proven to be a significant factor in producing good quality software in our organization. It is easily maintained, well organized and (best of all) automated.

Donna Wengrowski Mentor Graphics Corporation

2. Introduction to The Problem Reporting System

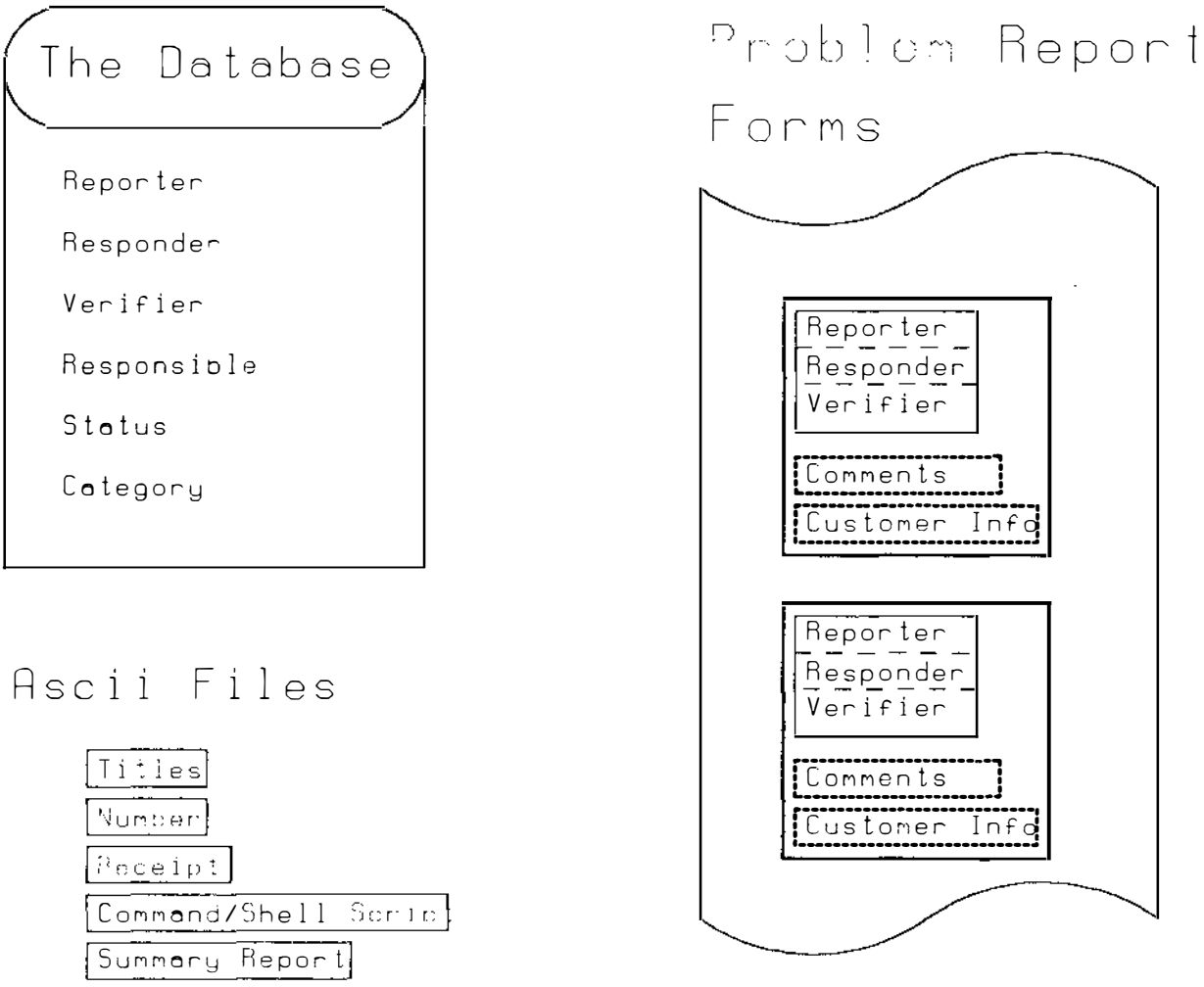
Software problems are reported by several different sources; Customers, Field and International Offices, and internally. Between the time it is reported and until it is resolved, each problem report must be accessable by The Customer Support and Engineering departments. Not only must they be able to see "what is happening" with the problem report, they may also want to know when it was reported, what version of the software will correct the problem, and what can be done now to prevent it. To assure quality software, each problem must be tracked and monitored by QA until it is resolved.

We run the problem reporting system on Apollo Network hardware configuration and have a Unix-like operating system called Aegis. The network gives all users the ability to access the system concurrently. We also have electronic mail, and a relational database system that was written for our software products. We designed a problem reporting system that would meet our needs given the hardware and software tools and utilities we had available.

The problem reporting system is a set of two pascal programs that interface with a relational database through a "shell script" or "command file" that is appended to by the programs. Ascii files are also created and accessed through the programs. Both the database information and the ascii files may be accessed directly by the QA department. We also make use of electronic mail to request responses from engineers and to send receipts to users reporting problems.

3. Storage and The Flow of Information

There are three parts to the storage system that is accessed by the problem report system, the relational database, the ascii files that are problem report forms, and other ascii files that are working files used by the problem report system programs.



Storage of Information

Donna Wengrowski Mentor Graphics Corporation

3.1 The Relational Database Schema

The relational database used by the problem reporting system will not be covered in detail here, only the schema used to define it. For more information on the database see "A Database Management System for Design Engineers" by Jack Bennett (DAC Proceedings 1982). The database schema is defined as follows. For each problem report in the system there exists, or will exist one entry in each of the following relations:

- o Reporter
- o Responder
- o Verifier
- o Responsible
- o Status
- o Category

The attributes of each relation will not be coverd by this paper, the relation names should be self-explanatory. For example, a new problem report is entered into the database, the following relations will have values assigned to their attributes: Reporter(= reporter information), Responsible (=QA), Status (=New), and Category (=Unknown). As the problem flows through the system, the Responder and Verifier relation attributes are also assigned values.

The problem reporting system is not a real time database interface, the information that is obtained from users is queued into a command file and is entered into the database by QA. This is called a database update. QA may update the database at any time. All other users have read only access to the database, so they may do scans for various information. For more information on database scans see Section 6.0 - Utilities.

3.2 Storage of Problem Report Forms

Problem reports are viewed as a form by most users. This form is actually one or more ascii files that are created and stored by the system. Each problem report is assigned a unique number when it enters the system, this number is then used to name the files that are associated with the problem report. Up to three ascii files are created for each problem report in the system.

The "form" file initially contains the reporter information, and the problem description. Later, information is appended to the "form" file

whenever any action is taken on the problem report. The history of a problem report is retained automatically.

The "customer" file is created if the problem report was originated by a customer. It contains customer information. The customer information is separated from the "form" file to 1) minimize storage space and 2) Give Customer Support Department the ability to obtain information and statistics from one location.

The the "comments" file is used for storage of misc. information. This is also an optional file. A comment file may be created or appended to at any time. The is also another way to document any action or history of a problem report.

Because the problem report form that the user sees is stored in ascii file(s), as well as in the database, the user may view the information in a problem report form immediately after reporting it. It is also much faster to access an ascii file (to view a problem report form) than it is to build a form from the information stored in the database.

3.3 More Files Used By The Problem Report System

There are several other working files used by the system, these are also ascii files. The "title" file contains each problem report and it's corresponding title, used to nicely format scan output, and to refer to a title when sending electronic mail. The "number" file contains the current problem report number, (it is incremented by 1 and restored each time a new report is entered into the system). The "receipt" file contains user names and problem numbers for sending receipts to users when the database is updated (receipts are sent by electronic mail). Most importantly there is the "shell script" or "command" file that is used to update the database. A "summary" file is also an ascii file that is created by QA (ususally after the database is updated), that has current information on all the reports in the database. For more on the summary file see Section 6.0 - Utilities.

The database, the file structure, and the electronic mail all are tools used in the problem reporting system. The next sections, will describe the programs that make the tools work for the system, and the flow of a problem report through the system, or how it all comes together.

4. The Programs - A Friendly Interface

There are two programs that make up the Problem Report System. The first is the user side the Control Program. The other side is the QA side, the Monitor Program. Each may be invoked and run on any terminal on the network.

4.1 The User Side - The Control Program

The Control Program brings up the following menu of commands:

```
-------------============-----------------------------------
WELCOME TO THE -->ALL NEW<-- QA PROBLEM REPORTING SYSTEM!
-------------============-----------------------------------

1.  Submit a New Problem Report.
2.  Respond to a Problem Report/Supply Additional Information Needed.
3.  Verify a Problem Report.
4.  View or Print a Problem Report.
5.  View or Print the Summary Report.
6.  Add Comments to a Problem Report.
7.  Return Misdirected Problem Report to QA.
8.  Search for Problem Reports.
9.  Print date/time of last Database Update.

Input choice (1..9, 0 when done): 0
```

Most of the choices in the menu are self-explanatory. All of the selections will either prompt the user for more information or bring up their own menus. The QA department felt the most important issue in designing a system was to make it easy to use -- fast and friendly.

Choice 8 from the menu enters the user into a scan menu. The scan menu allows read-only access to the database. See Section 6.0 - Utilities, for more information on scans.

Choices 1, 2, 3, and 7 all write to a "shell script" or "command" file. This file changes the contents of the database when QA does a database update. See section 4.2 The QA Side, for more information on updating the database.

Choices 4, 5 and 9 allow read-only access to ascii files. Choice 6 allows editing of ascii files.

4.2 The QA Side - The Monitor Program

The Monitor program is used only by QA, it has some of the same choices as the Control Program, but also the ability to directly change the values of database information. The Monitor Program brings up the following menu of commands:

```
-------------------------==================
QA PROBLEM REPORT SYSTEM MONITOR PROCESS:
-------------------------==================

1.  List NEW or RESPONDED TO Problem Reports.
2.  Send Report For Response ( ---> FORWARD).
3.  Send Report For Verification ( ---> VERIFY).
4.  View or Print a Problem Report.
5.  View or Print the Summary Report.
6.  Add comments to a Problem Report.
7.  List Late Reports. (status = FORWARD since <date>)
8.  Search for Problem Reports.
9.  Change Person Responsible for a Problem Report.
10. Change Category (TOOL/TOPIC) of a Problem Report.
11. Change Status of a Problem Report.
12. Update Database.
13. Create New Summary Report.
14. Delete a Report from the Database.
15. Assign Problem Reports to a QA person.
16. Check for a Response to a Problem Report.
17. Check for a Verification to a Problem Report.

Input choice: (1..17, 0 When Done): 0
```

Most of the choices are self-explanatory. The capital letters refer to status codes that may be assigned to a problem report. See Section 5.0 for more information on status codes and the life cycle of a problem report.

There is one person in our QA organization that monitors the entire process. That person uses choice 15 to alert the QA person responsible if any NEW problem reports are received in his/her area. Choices 2, 3, 12 and 15 all send electronic mail. Choice 8 brings up the same scan menu as the Control Program.

Choices 1, 7, 8, 13, 16 and 17 all allow read-only access to the database. Choices 2, 3, 9, 10, 11, 14 all write to a "shell script" or "command" file. Choice 12 actually runs the "command" file on the database, changing its contents. Choices 4 and 5 access ascii files as read-only. Choice 6 allows editing of ascii files.

5. The Life Cycle of a Problem Report

This section will bring together all the different parts of the problem reporting system that have been described. See Appendix B for a graphical view of the life cycle of a problem report represented by status codes.

5.1 A New Problem Report Enters the System

The person reporting the problem uses the Control program to enter the problem into the system. This causes the following actions to take place:

- The command file is appended with the following:
  1. Reporter information.
  2. Responsible = QA.
  3. Status = NEW.
  4. Category = Unassigned.
- An ascii file is created and stored that contains the problem "form".
- (Optional) A comment file is created with any comments the reporter wants to add.
- (Optional) A customer information file is created that contains information on the customer who it was reported by.
- A unique problem number is assigned to the problem report, and the number file is updated.
- The problem title(supplied by the reporter) and the problem number are stored in the title file.
- The reporter name, the problem number and the title are stored in the receipt file. When the database is updated with the information in the command file a receipt will be sent to the reporter via mail.

5.2 QA sends for a Response

The person in QA that monitors the system assigns the NEW problem report to a QA person to deal with using the Monitor program. Mail is sent to the QA person notifying them that a NEW problem has been submitted in their area. The QA person then uses the Monitor program to forward the problem report to the engineer who is responsible for the software. When a

problem report is forwarded the following actions take place:

- o The command file is appended with the following:
  1. Responsible = engineer it was forwarded to.
  2. Status = FORWARD.
  3. Category = assigned by QA. This is the tool or topic name that the problem report will appear under in the Summary Report.
- o Mail is sent to the engineer requesting a response.

5.3 The Engineer Responds

The engineer uses the Control program to view the problem report. The engineer may now do either of the following (using the Control program):

I. Send the report back to QA as misdirected, the problem is not in the engineers area of responsibility. If the problem report was misdirected the following actions take place:

- o The command file is appended with the following:
  1. Responsible = QA.
  2. Status = MISDIRECTED.
- o (Optional) Comments may be added to the comments file that is associated with this problem report.

When a misdirected problem report is returned to QA a QA person then re-forwards the problem report to the *correct* engineer using the Monitor program.

II. Respond to the problem report. When an engineer responds to a problem report the following actions take place:

- o The command file is appended with the following:
  1. Responder information.
  2. Responsible = QA.
  3. Status = RESPONDED TO.

- o The problem report form is appended with the responder information.
- o (Optional) Comments may be added to the comments file that is associated with this problem report.

5.4 QA Receives a Response

Once QA receives a response, the problem report is assigned a new status (using the Monitor program) depending on what the engineers response was. Below is a list of possible status codes and what they mean:

A. Can Not Dup - The engineer could not duplicate the problem. If more information is needed the problem will be forwarded back to the person who reported it. If more information is not needed and the problem just can not be duplicated then no further action will be taken on this problem report.

B. Same As <number> - This problem has already been reported, refer to problem report <number>. No further action will be taken on this "duplicate".

C. Can Not Fix/Will Not Fix - this problem is either a hardware or systems limitation (we have no control over it), or it is so obscure that it is deemed unworthy of fixing.

D. PER - this is not a problem, it is a Program Enhancement Request. This will be saved and reviewed during the planning stages of future releases of our software.

E. No Bug - this is not a problem, probably a pilot error. No further action will be taken on this problem report.

F. To Fix - this is a problem and will be fixed in a future software release.

5.5 QA sends for Verification

Once a problem has been fixed, the engineer notifies QA by mail. The report is now sent for verification. It is usually sent to the person who originally reported the problem, or (in the case of a customer reported problem) to a person that can verify that the problem has actually been fixed. QA uses the Monitor program to send for a verification, and the following actions take place:

- o The command file is appended with the following:

1. Responsible = person it was sent to for verification.

2. Status = VERIFY.

- Mail is sent to the person requesting a verification.

5.6 The Problem is Verified

The engineer uses the Control program to view the problem report. The engineer may now do either of the following (using the Control program):

I. The problem has been fixed. The following actions take place:

- The command file is appended with the following:

  1. Verification information.

  2. Responsible = QA.

  3. Status = RESOLVED.

- The problem report form is appended with the verifier information.

- (Optional) Comments may be added to the comments file that is associated with this problem report.

No further action will be taken on this problem report.

II. The problem has not been fixed. The following actions take place:

- The command file is appended with the following:

  1. Responsible = QA.

  2. Status = UNRESOLVED.

- The problem report form is appended with the reason that the problem could not be verified.

- (Optional) Comments may be added to the comments file that is associated with this problem report.

When a problem report can not be verified (status = UNRESOLVED), QA will use the Monitor program and forward it to the engineer who was responsible for fixing the problem.

Donna Wengrowski Mentor Graphics Corporation

6. Utilities

One of the best features of the problem reporting system are the utilities that are available for perusing the information in the system. These include viewing and printing problem reports, viewing and printing the summary report, and the ability to do various scans on the information in the database. Utilities used by QA include updating the database, and creating a new summary report, as well as the ability to change/view the contents of the database.

6.1 Scans

Scans are a powerful tool that allow all users to access the information in the problem report system database. In either the Control or Monitor programs the following menu is displayed:

```
-------------------------------------
Search for PRs (Problem Reports) Menu
-------------------------------------

1.  having status: {status}.
2.  reported on {tool}.
3.  reported since {date}.
4.  reported on {tool} since {date}.
5.  reported on {tool} having status: {status}.
6.  that {user} is responsible for.
7.  that {user} reported.
8.  report current status of PR {number}.
9.  report tool PR {number} is filed under.
10. report user responsible for PR {number}.

Input choice (1..10, 0 when done):
```

Each selection will prompt the user for the information needed, build a command file that will access the database (read-only), and display the results on the screen.

6.2 Updating the Database

The QA department should update the database periodically using the Monitor program. This causes the following actions to take place:

- Run a background process that "flushes" the contents of the command file, updating the database with this information. The contents of the command file is saved in a backup file before it is blanked out.

Donna Wengrowski Mentor Graphics Corporation

- Save and check the error output from the above "flush" process, making sure no errors occured on an update attempt. If the command file was blank, no update will occur.

- Save the date and time of the last database update (or attempted update) in an ascii file.

- Send receipts (via mail) to users who submitted NEW problems, indicating when the information was entered into the database.

6.3 Creating a Summary Report

Other than the scanning facilities, the summary report is the most important utility available. This files contatins problem report information listed by category (tool or topic name). The summary report is built directly from information stored in the database. If there is infomation that is missing or incomplete, the summary report will reflect it. In the case of lost information, the database may be adjusted by QA using the Monitor program. See Appendix A for a sample summary report.

7. Conclusion

This problem reporting system has worked well for our QA department. It is easy to use, our engineering and customer service departments have access to information on line, and it has several built in back up systems.

We have found that using this system, rather than the old manual method, we have increased our speed and efficiency. We have also eliminated the confusion of determining "what is going on" with each individual problem since the history of the problem become part of the report form. In the future we hope to make the system faster and also 'real-time', so no updating will be necessary.

Donna Wengrowski Mentor Graphics Corporation

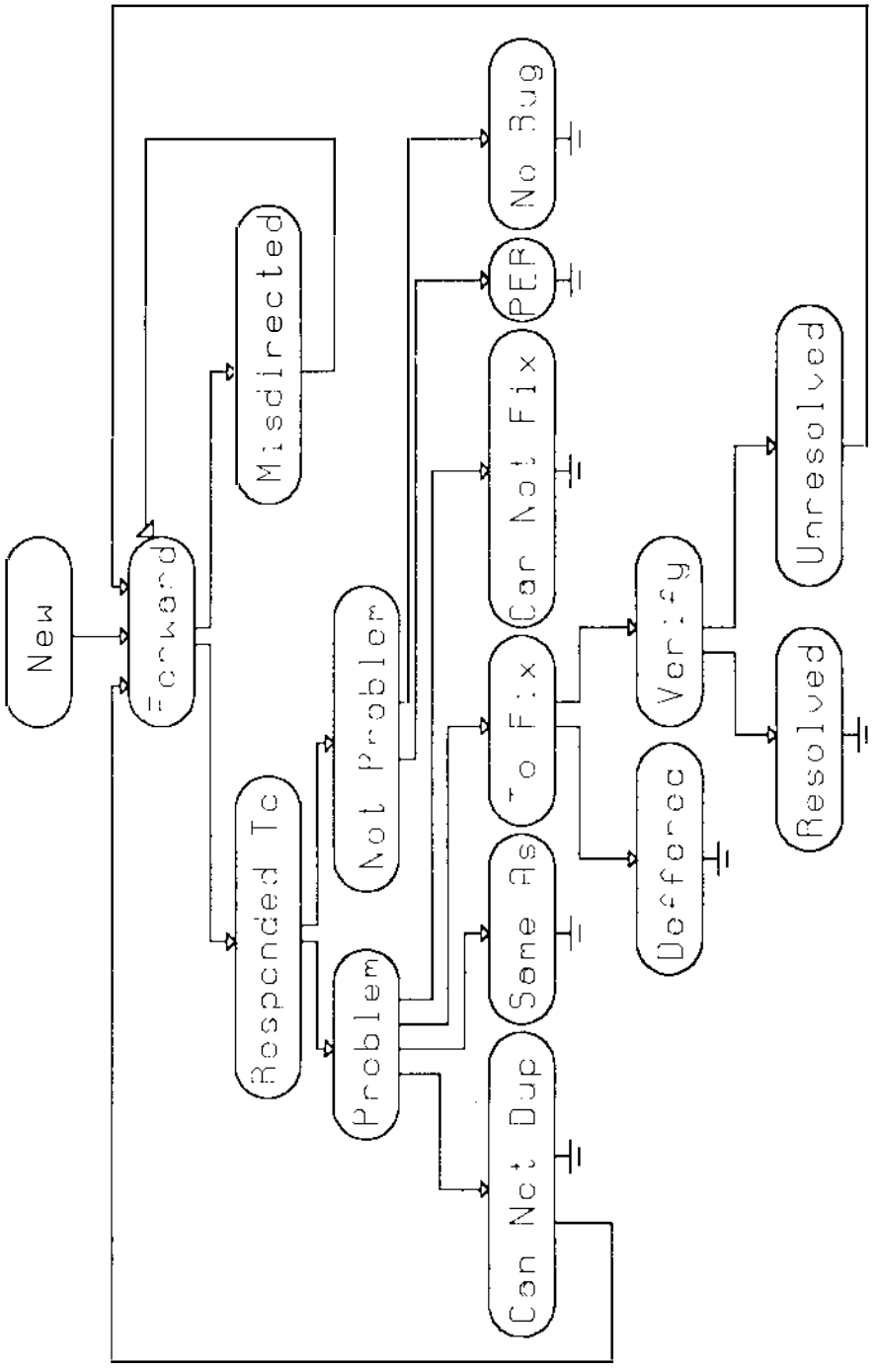
Appendix A

PROBLEM NUMBER	OR	TITLE	USER RESPONSIBLE	DATE	STATUS
TOPIC A					
220	C	Title of report 220	QA	840726	Resolved
TOPIC B					
123	I	Title of report 123	MarvinK	840129	Responded To
128	I	Title of report 128	MelodyS	840129	To Fix
175	C	Title of report 175	Berll	840129	Verify
277	C	Title of report 277	KittyC	840706	Forward
358	I	title of report 358	QA	840806	No Bug
TOPIC C					
8	I	title of report 8	QA	840806	Can Not Fix
58	I	title of report 58	QA	840806	Same As PR 8

Sample Summary Report

Donna Wengrowski Mentor Graphics Corporation

Appendix B



Status Code Life Cycle

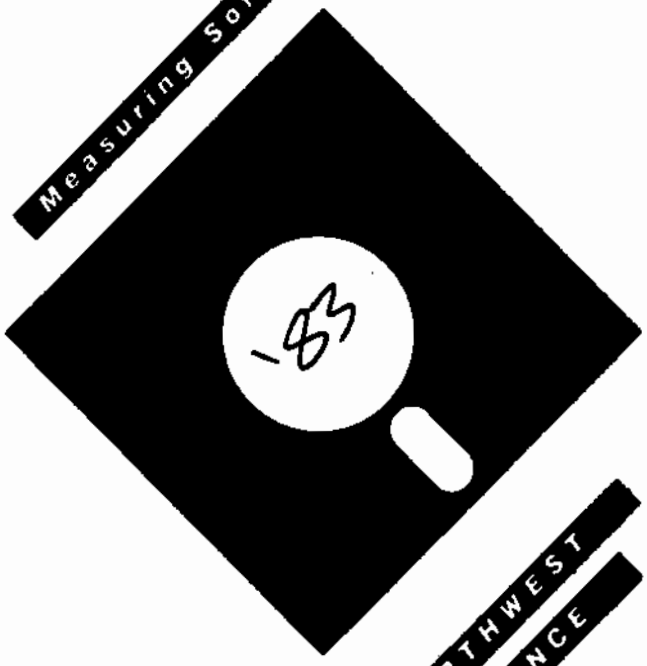
Donna Wengrowski Mentor Graphics Corporation

BIOGRAPHY

Donna Wengrowski is currently a Q.A. Software Engineer at Mentor Graphics Corp in Beaverton Oregon. She received a B.S. degree in Computer Science from Arizona State University in 1981.

She has also been a Computer Systems Manager at Swan Wooster Engineering in Portland, and a Research & Development Engineer for Hewlett-Packard in Corvallis.

Measuring Software Quality



PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

September 26, 1983

Corvallis, Oregon

1983 Northwest Software Quality Conference

Table of Contents

CONFERENCE INTRODUCTION

Kenneth P. Oar
Software Quality Manager
Portable Computer Division
Hewlett Packard

The first Pacific Northwest Software Quality Conference was held at the Hewlett Packard facility in Corvallis, Oregon on September 26, 1983. I would like to express my appreciation to our Conference Speakers and to the 1983 Committee for the many hours of planning and organi zation that contributed to making our first conference a huge success. We had 231 conference attendees representing the following companies:

Hewlett Packard
Boeing Aerospace Company
Mentor Graphics
Metheus Corporation
Tektronix Incorporated
Floatng Point Systems
Servio Logic
John Fluke Mfg. Company
Intel Corporation
Oregon State University
Delphi Computing Consultants
SonaTech Incorporated
Boeing Computer Services

I'd like to mention a little background about the conference. In the fall of 1982 a group of Software Quality Engineers and Managers at the Portable Computer Division of Hewlett Packard met with representatives from Tektronix, Oregon State University, and the Oregon Graduate Center to discuss the possibility of creating a local conference devoted to Software Quality. It was our goal to have an inexpensive conference which could be attended by literally anyone who had a desire to participate. To keep costs down we limited the conference to a single day thus avoiding ovenight travel accomodations for most people and as a non profit organization charge only a nominal fee to cover conference expenses. We also wanted the conference to be highly interactive inviting free flow of information between the speakers and their audience. And finally we wanted the conference to stimulate interest for improving Software Quality by providing local opportunities for engineers and managers to share their successes publicly.

The idea sounded good, a committee was formed and we selected the theme of "Measuring Software Quality" for 1983. Since this was the first conference we asked speakers to participate rather than issuing a call for papers. This ultimately posed quite a problem in developing proceedings for the conference. Since no papers were turned in we videotaped the conference. Unfortunately only one third of the tapes were reproduceable, the problem being very poor audio due to microphones being far away from the speakers. This created significant problems in getting the tapes transcribed. We have made a best effort to understand what was on the videotapes and edit the talks for clarity in written form. In most cases we have left the conversational tone in the transcription.

KEY PARTICIPANTS IN THE 1983 CONFERENCE

Chairman

Kenneth P. Oar, Software Quality Manager
Hewlett Packard, Portable Computer Division
1000 NE Circle Blvd.
Corvallis, Or 97330

Cochairman

Charles F. Martiny, Software Center Manager
Tektronix, Inc.
P.O. Box 500
Beaverton, Or 97077

Committee

Sherry Sisson, Software Quality Manager
Hewlett Packard, Portable Computer Division
1000 NE Circle Blvd.
Corvallis, Oregon 97330

Pat Megowan, Software Quality Engineer
Hewlett Packard, Portable Computer Division
1000 NE Circle Blvd.
Corvallis, Oregon 97330

George Tice, Jr., Senior Performance Assurance Engr.
Tektronix, Inc. 92-606
P.O. Box 500
Beaverton, Oregon 97077

John Vance, Product Evaluation Manager
Tektronix, Inc. 92-526
P.O. Box 500
Beaverton, Oregon 97077

Ted Lewis, Professor
Oregon State University
Computer Science Department
Corvalalis, Oregon 97330

Robert G. Babb II, Assistant Professor
Department of Computer Science and Engineering
Oregon Graduate Center
19600 N.W. Walker Road
Beaverton, Oregon 97006

CONFERENCE SPEAKERS

Stephen Swerling, Vice President of Engr.
Mentor Graphics Corporation
10200 SW Nimbus Ave.
Tigard, Oregon

T. Michael Ward, Computer Group Quality Manager
Hewlett Packard Company
19310 Pruneridge Ave
Cupertino, California

Curtis Cook, Professor
Computer Science Department
Oregon State University
Corvallis, Oregon

P. Edward Presson, Software Research Engineer
Boeing Aerospace Company
P.O. Box 3999 (M.S. 8C-19)
Seattle, Washington 98124

Ramune Arlauskas, R&D Software Manager
Tektronix, Incoporated
Wilsonville Industrial Park 63-523
P.O. Box 1000
Wilsonville, Oregon 97070

Bill Gervasi R&D Software Manager
Intel Corporation
5200 NE Elam Young Parkway
Hillsboro, Oregon 97123

Ned Thanhouser, R&D Software Manager
Intel Corporation
5200 NE Elam Young Parkway
Hillsboro, Oregon 97123

Thomas Bowen, Software Research Engineer
Boeing Aerospace Company
P.O.Box 3999 (M.S. 8C-19)
Seattle, Washington 98124

William J. Moore, Software Quality Assurance Mgr.
Metheus Corporation
P.O. Box 1049
Hillsboro, Oregon 97123

Nelson Mills R&D Software Manager
Hewlett Packard, Portable Computer Division
1000 NE Circle Blvd.
Corvallis, Oregon 97330

Janice Cleary Software Engineering Manager
Intel Corporation HF2-1-800
5200 NE Elam Young Parkway
Hillsboro, Oregon 97123

Bruce Coorpender R&D Manager
Tektronix Incorporated
Wilsonville Industrial Park 63-205
P.O. Box 1000
Wilsonville, Oregon 97070

Paul Nichols Software Quality Manager
Hewlett Packard Information Networks Division
19420 Homestead Road
Cupertino, California 95014

SOFTWARE QUALITY-THE REAL WORLD

Stephen Swerling
Vice President of Engineering
Mentor Graphics Corporation

The text of this paper was transcribed from videotape. It has been edited for clarity in written form, but purposely left in a distinctly conversational tone. Slides and pictures have been included where appropriate.

INTRODUCTION

I am really pleased to be participating in this conference. Software quality is something I spend a lot of time worrying about. The fact that so many of you showed up today to discuss this subject and hear other people's views on "What is quality" and "How does one measure" and "what we can do about" indicates the importance of the topic. We expect that we're going to see a lot more professional activity in this area in the future.

I'd like to make a couple of minor corrections in so far as the publicity material is concerned. First of all the topic of the talk, "the real world" as it was written up in the brochure. Some people have suggested that I'm not in a position to talk about the real world. There are several possible reasons. Some of you are familiar with the fact that Mentor Graphics is two years old, has launched off into a nonexistent market, has grown to a sales rate of about 20 million dollars this year and twice that next year, and my background is hardware, not software. So you have to judge for yourself. You may take what I say with a grain of salt, if you'd like. The other minor correction is that at the time the brochure was written we mentioned that our past releases have had 250,000 lines of code and we're now at approximately 350,000 lines of released code in production. We at least have as a group at Mentor, experience in the area of assuring software quality.

ABOUT MENTOR GRAPHICS

Let me tell you just very briefly what Mentor Graphics is and what we do. As I said we were formed in the spring of 1981. We were a group of Tek people for the most part. We decided to go into what's called CAE, Computer Aided Engineering industry. That's a name we're about to change, I think, because nobody's happy with that name. Basically what we do is provide software tools for hardware engineers that enhance their ability to do their job.

Examples are schematic entry, editing, schematic checking and then various kinds of sophisticated analysis tools such as logic and circuit simulation. I'm sure many of you are familiar with SPICE. Once an engineer has a logic circuit diagram that's entered into our system, the engineer is able to run an interactive version of SPICE where the schematic is on view on the screen. You can plot wave forms in the

circuit, either time domain, frequency domain, or whatever you want. You get all the capability of SPICE, but you never have to deal with the rather poor human interface that SPICE traditionally has, for those of you who have worked in that area. I'm telling you this not to sell the product, but to give you a flavor for the complexity of some of our software. We have features such as the ability to put the cursor on a circuit node on the schematic and probe that node and plot it's waveform without knowing what the node is named and even if it's not named. In normal SPICE, of course, that kind of thing is not possible.

The other kind of major analysis tool that we have is logic simulation. It's analogous to the analog circuit simulation of SPICE. Basically you're able to model any logic system with a very sophisticated simulator and again, have all that human interactivity that we've built into the tools. I won't dwell on products any more

We are based on the Apollo computer which is a 68,000 based computer, about the power of a small VAX . It's primary claim to fame is a very sophisticated multitasking, multiwindow, networking operating system, including a lot of software capabilities that made it possible for us to deliver the amount of code that we did and to deliver it at the quality level that we did. I'll get more into that later. I think the one message that I want to make sure is clear from the discussion about the company is that we have a helluva time testing our software and the reason is, there's a lot of graphics involvement, a lot of interactivity. I think, as many of you know then, it is very difficult to define a test set that's elaborate enough to adequately prove to yourself that the software is working right. We do pretty well on logic simulation and so on because those are more batch like processes but it's still a real problem for us.

QUALITY DEFINITION

Let me speak very briefly about the definition of quality. The specification must match the application and the software functionality must match the specification. Sometimes they are lumped into one item by people. There's no point in debating that. I think we know how to manage those two processes. We know pretty well that if we involve the customers, talk to people and don't develop a specification in a vacuum then we're going to end up with a pretty good match to the application. And we know we can manage the software development process well enough to make sure that the software, at least to a reasonable approximation, resembles the specification and so on. I think those are moderately well in hand.

The primary thrust, I think, of most software quality measurement work is the third part of the definition, the aberration frequency. We're going to talk much more about that as the day goes by. By aberration, I am talking about all unexpected behaviors, wrong behaviors, crashes, and so on. Although we're focusing on that and many of the tools we'll talk about are in that area, of course, you have to be careful. You want to make sure that your quality is high in the first two areas, because you don't want anybody saying "ah, their software works okay, but it doesn't do what I want". I'm not sure which is worse, to have lousy

software that doesn't work, or to have lousy software that works, but doesn't do what I want.

Why do we measure software quality? Why is it important to do work in this area? You have to have a target, and that target depends on your context. If you're developing software that's going to manage an antiballistic missile firing system, you've got one set of problems. If you are making a word processing package for the APPLE you have a totally different set of problems. I consider in our case, for example we're somewhere in the middle. I think most of the people in the room are involved in projects somewhere in the middle. The volumes aren't PC volumes, well, perhaps they are for a couple of people, but they are products intended to be sold in quantities. One obvious ramification of quantities, is, you've got to be careful. It costs a lot to correct a problem that's gotten out to the field that can't wait for some future release. In our case our target market is the hundreds of thousands of electronic engineers in the world. If we aren't careful, we'll have the problem of fixing problems and changing things all over the world, literally, because we're currently selling internationally. And this is very complex software, it is not a single floppy that you can replicate very easily. There is a reason why software quality measurement is critical. You have to make management decisions. Where am I in this release process? How bad is it? How good is it? Am I ready to let it go? Do I have to delay? and so on. We try to be very quantitative about these things, and I'll tell you more about that later.

You have to be able to assess the quality of your test suites as well. Not only do you have to measure the quality of your software, you have to know how good your test suite is. And those two are interlinked of course, but it requires some kind of quality measurement in order to do the test suite qualification as well. Also I believe strongly in building historical data bases. I think once we've got some data on how our particular environment works we're more able to process future releases more effectively.

QUALITY IN THE DESIGN

Well, what we all want to do of course is bring the quality level as close to the desired goal as possible. As I said earlier the first two definitions of quality, specification match the application and software match to the spec, I think we can manage those processes pretty well. The third one is an interesting area. There are an awful lot of tools around that impact that area and I'm going to talk much more about that. I think we need to pay attention to all those tools and look at how they fit into your particular environment. Now some of the things in our strategy are very motherhood and apple pie kind of stuff. It's no longer debateable that you should have good specs and design reviews and modern structured languages. You know there's no point harping on all that stuff, everybody understands that, right? But I think it's still the case that a lot of people don't pay enough attention to what some of these things really mean. The detailed specification process for example has made a major difference at Mentor. I'll show you some data later, comparing some cases where the specs were different quality, different detail. I think they are

one of the key items. The elaborate debugging aids mentioned earlier also have made it possible for us to not only deliver the software, but reach a quality level that was consistent with our goals.

The Apollo happens to come with a lot of tools that make it very easy for the engineers to do their job. It has a powerful command file system. I should mention, that's a tool that has also made a major difference to us. It's possible for us to store away a sequence of events that led to a problem and replay it later, step by step in debug mode. Out ability to find problems in software that way and fix them is very high.

THE QA PHASE

After software has been released into our internal QA process we go into a mode that's kind of interesting. Somewhat different in my experience than in other organizations. So far in all our major releases, we have used cross team testing. For the most part, my development teams have been synchronized in their activities, that is to say virtually everyone has been contributing to the same release and there were few ongoing projects that transcended the release point and could not be disrupted. That's changing of course as the company grows, but up until now we've had a very small Quality Assurance organization that basically coordinated and managed the process and helped to make test plans and so on, but we throw most of engineering into that test period. Where we'll go from there is unclear. I think that it's the case where the quality organization is growing and will continue to grow, but it will never reach the level, I think, where it will be large enough to be able to manage and do that entire process of software qualification. We kind of like the cross team testing approach.

Focused, detailed test plans are a major factor. I can't stress that enough. They look at the areas of software that have changed, new features, weak areas, they sort of look in the back of their mind and if they are afraid that something is screwed up, or might be screwed up in some weird case, they most likely or very often put those down in the test plans and they have a lot of success there. I think that the way we are working now, we're gradually working to the point where the quality assurance organization will spend a lot of time enhancing those test plans and making sure they are complete enough. Test plans are a realistic alternative to exhaustive testing in my view.

The other thing we do during this cross team testing period is to invoke a very formal internal bug reporting system. That's separate from the field reporting system. We call it a TPR system. All of the problems flow through the quality assurance organization. They are then passed to the developers involved and eventually go back to the testors for reverification of fixes and so on. This past release that system handled many many hundreds of problems following the end of testing by the project teams. We think it works very well. It allows us to manage the process of making changes. It gives me management visibility. It gives my reports management visibility of what's going on, provides historical data and again I think it allows us to make much better decisions.

The other thing we do, of course, is have a formal escalating approval system for changes. In the last weeks just before release it's only me who can approve changes.

AFTER RELEASE

After release I don't think we do anything very surprising. We do have an on line problem reporting system from the field. Both the field offices as well as customers can phone in directly to a node and send in transcripts, inspect SPR's, problem reports in process, and so on. Again, the command file system is tremendously valuable here. It's possible for someone in the field to send in one of these command files and we can replicate their problem exactly on a node in the space of hours.

RESULTS

What has all this resulted in? Well, we'll go to the last definition of quality first.

POST-RELEASE QUALITY

Application	KBytes/SPR	Nature
A	76.5	complex, carefully done
B	65.4	larger % utilities than others
C	19.1	middle of road
D	8.7	last minute

This is actual data. Our field reporting system, the SPR system has been up and running since this spring. Since it came up, the statistics for this release in the field look as shown. The top most application actually was done by one individual who was so fastidious and did such a detailed specification and detailed design, even working alone he was able to achieve by far the highest quality product in the system. From any of you in the room who know him you might be speculating on who it is. The next one down is interesting because we have a very large base of utilities. Something on the order of a third of our delivered binary software is utilities. So roughly two thirds of the grand total is applications. Application B doesn't have much in the application area, it's primarily a utilities job and it was done fairly quickly. Going down to the bottom, the last application specification was kind of coarse, it was done in a hurry, there was inadequate testing, and so on. So this kind of 10 to 1 range is our experience on an earlier release. We're changing this. We're not going to let the range cover quite that broad an area. The whole range is moving up.

In so far as specification, I think I'm satisfied with that. Again, we tightly manage that process, we had a lot of coordination between teams and the bottom line is of course customers saying, "Yes, you have the best functionality, it's exactly what we need, except for the following 17 things." Now of course there are a few things I wish I could do over again if I could. You know we started with a non existent market and had very little in the way of precedence for

products. So in many ways we had to make a lot of decisions based on not a lot of data. So surely there are areas I'd like to do again. But I'm not going to tell you what they are.

HARDWARE SOFTWARE ANALOGIES

Now as I mentioned earlier, my background is hardware and I see a lot of parallels between the hardware and software testing worlds. The hardware world product can be as good as you want it to be. That problem is solved. The software world isn't quite in that condition. And I think the analysis of some of these analogies is useful. How many of you are familiar with any of these concepts. Fault simulation, testability analysis of hardware, and so on.

Fault simulation basically answers the question "for a given set of tests, how many 'stuck at' faults in this particular board, let us say, can be found?" Meaning if I just look at the edge connector of the board with hardware test equipment instrumentation, can I tell whether a fault exists at a particular node in the circuit, or not, given that particular set of test vectors. Now, there's some very interesting things about fault simulation. First of all the modeling process that's used is mainly stuck at zero and stuck at one.It doesn't relate to the actual hardware faults that occur over time in a piece of hardware, very well at all, especially with modern IC technology. It was more true in the past. It's much less true now. Although that modeling is inexact it turns out that fault simulation is an excellent tool for qualifying a set of test vectors and determining whether or not you can find, with automatic test equipment in a manufacturing environment, chips that have failed. It's very effective.

There is no direct 100% analogy to that in software, because as you all know, software doesn't fail. It works for all time to an acceptable percentage of error level, it works for all time exactly as it did on the first day that it was made. The fact it may produce aberrant results later because somebody sticks in different data is a different problem entirely. That has nothing to do with this issue.

There are two kinds of activity in software quality measurement that are interesting analogies to fault simulation. One is coverage analysis. How many of you are using coverage analysis in some way or another? Quite a crowd, 2. I think we all need to do more work there. We're not using it either, but I intend to in fact spend some energy on this. Coverage analysis basically says,"has this routine been exercised during this given set of tests? And in many cases, have all the branches of this particular conditional been exercised by a given set of tests?" I know that there are many reasons why people are not using that kind of instrument in software. I have a feeling though, that for complex systems like ours and for many of the rest of you in the room, I think it's worth doing, especially if you have the support tools to make it easy and if you focus on the areas that need it. I guess I am not in favor of instrumenting all my 350,000 lines of code but I have a feeling that it has more payback than people realize.

The other analogy to fault simulation which is a loose analogy, is failure seeding. Again, it's very very infrequently used. Basically, it's analogous in that you deliberately put problems into the software and see if the test team finds them. Well, like I said, that's exactly what fault simulation does. The problem is, the analogy is loose. The problems are totally different kinds of problems that are inserted. However, you can qualify a test suite that way, and perhaps there's some use there. I guess I'm not very optimistic about that though.

The next item on the chart, testability analysis, basically is an attempt by the hardware test people to solve the following problem. It turns out that fault simulation is incredibly computer consumptive. It's very nature is such, that for hardware designs, you can run your largest machines for the rest of it's life to go through a fault simulation. And people have tried to find ways to deal with that problem. One way is statistical fault simulation which is suddenly getting a lot of attention the last few years, where you only run your fault simulation over some small percentage, some random sample, of the test vectors. What people try to do is say, "Let's not really simulate the design, let's not use test vectors at all, instead, let's try to measure the complexity of the hardware" Can a signal coming in from the edge connector reach into an internal node in the board and then can and IO connector get visibility of that node? How difficult is it I should say, to get visibility of that node? That's what testability analysis does. The analogy in software is of course, complexity measurement. This is purely a relative kind of measurement which depends entirely on the situation and on your particular measurement software. It's useful in the hardware world. It's suspect for several technical reasons in software. There are people who say you can compare two boards, and yes this one is more testable than that one according to the software and low and behold when I put them into production, I have the same results. I think the jury is still out on this area. It does have intrinsic value, the question is what value it is.

The last type of hardware world measurement process I'll talk about is self diagnosis. I'm sure all of you are familiar with systems that do that. Computer systems, military systems, one of Teks recent instrumentation products had a very high level of combined hardware/firmware diagnostic capability. I'm not sure what all the analogies are to self diagnosis in pure software. Assertion checking again is something we're very interested in and are about to ramp up instrumentation activity to do that. It's very useful for the intial test in software where you are trying to determine whether or not you have produced bizarre values in software at unexpected times. It can also help you determine whether or not your test suite is covering the whole range of values you are interested in for a particular routine. So I think assertion testing is a very useful tool. What we'll do at some point is stick the instrumentation in and leave it in for some period of time and see what happens.

CONCLUSION

I think that in closing off this subject what I think I would say is, I think we have a long way to go in the software world. I think that the attention that you folks are paying to this subject today is an indication that we're all going to be making progress here. I know we will.

To close I would like to tell you about a survey that was done. The survey was of an audience much like this. After a formal presentation, the surveyors found that at the end of the talk, 10 percent of the audience was sound asleep, 10 percent was paying attention, and 80 percent of the audience was involved in intense sexual fantasies. Thank you very much for giving you the opportunity to enjoy yourself.

BIOGRAPHY:Stephen Swerling received his education at MIT (BSEE 1963) and Yale (M. Eng 1964) He spent a number of years at the Arthur D Little Consulting Company, and 5 years at Tektronix before he joined Mentor Graphics Corporation as V.P. engineering in 1981.

YHP WINS THE DEMING AWARD IN JAPAN!

T. Michael Ward
Product Assurance Manager
Computer Support Division
Hewlett Packard

The text of this paper was transcribed from videotape. It has been edited for clarity in written form, but purposely left in a distinctly conversational tone. Slides and pictures have been included where appropriate.

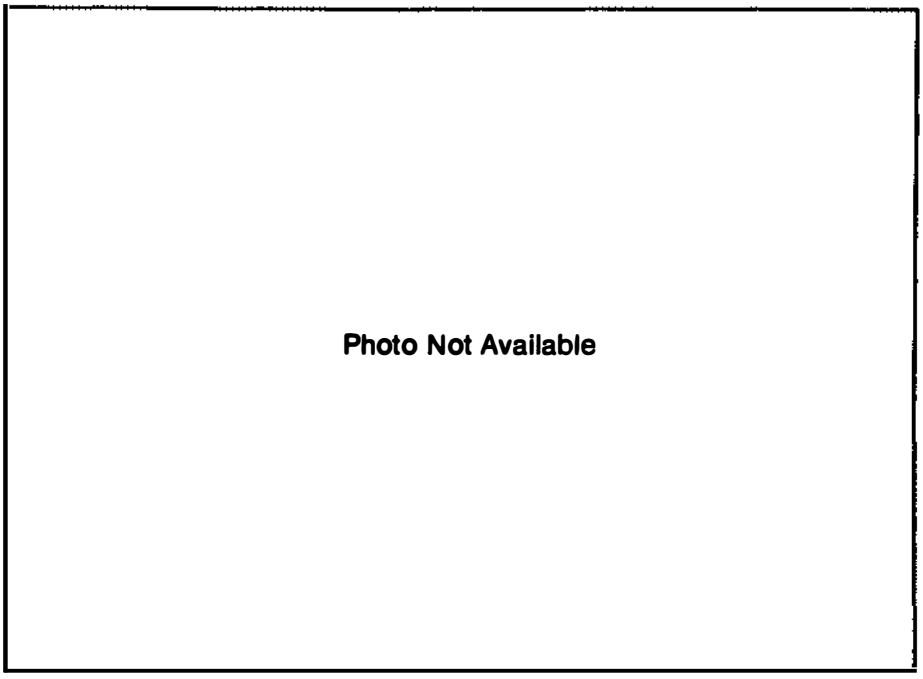
INTRODUCTION

I would like to spend a little time today telling you about the Deming award that was recently won by Yokogawa-Hewlett Packard, our sister corporation in Japan. I want to tell you a little bit about YHP, a little bit about the Deming award, what has happened to the organization as a result of that and where they expect to go in the future.

ABOUT DEMING

How many of you know who Deming is, W. Edwards Deming? Oh, super!! You guys pass. Deming is a pretty well known statistician in the U.S. who became famous in World War II doing some defense planning. He then tried to apply those methods throughout the U.S. in manufacturing operations because he thought he knew the right answer. I've heard him talk and he's disappointed that the United States hasn't listened very well to what he's had to say. So in the '50s, he got hired as a consultant to go to Japan, and they listened. They spent a lot of time educating managers, supervisors, and engineers, on the meaning and understanding of statistical quality control. Actually, he and Juran spent a lot of time in Japan training. In fact if you go to Japan today and if you speak Japanese, you can turn the radio on at six am. in the morning and you can get analysis of variants, statistical quality control or design of experiments courses on the radio every day. And a lot of people do that. An impressive amount of training goes on which is very necessary for success in this effort. Here's a picture of W. Edwards Deming who is about 82 or 83 years old now I think. He'll come up and give a four hour talk in front of a group like this and thank them all for having the energy to listen to him. He's a very dramatic guy who's charging around all the time.(next page)

So, what I wanted to cover today was a little bit about the Deming prize, what the award is and why YHP chose to try for it. What was their motivation? What is TQC, total quality control or company wide quality control as it is sometimes referred to? What does that mean in their example? And how did they implement it throughout their corporation? Then I'd like to show you some tangible results which were a requirement for the award. You had to show real progress in order to achieve the award over there. What lessons can we all learn from that, what lessons did they learn from it?



YHP BACKGROUND

Yokogawa Hewlett Packard was founded in 1963 as a joint venture. The Japanese had a law up until this year, that you couldn't have a wholly owned foreign company in their country, so Yokogawa Hewlett Packard is 51% Yokogawa Electric Works, and 49% Hewlett Packard. It is a joint venture designed so that HP could manufacture products in Japan and sell our own manufactured products through YHP. They could also design their own products.

Now, in Hewlett Packard, a division's success is really measured by how well they do with their own products, not how well they do with products they are selling for somebody else. There's a lot of emphasis on how well you do in your own products. For several years, I guess the first 10 or 12 years of their existence, they were really not a very good division. Boy, you just had to have excuses why their standard times were low, why their profit was low, why they couldn't build products as fast, why they had higher inventory-- they just had all kinds of reasons why they weren't doing well. It had to do with importation ties, customs lags, people weren't trained, the documentation was all in the wrong language and so forth. There were a lot of reasons why they weren't successful.

In about 1976 Kenzo Sasaoka who was the president of YHP, decided he really needed something to galvanize this organization into being more

successful, overcoming all these excuses they'd had and to surpass the other Hewlett Packard divisions. He didn't like coming in second best and having to explain to Dave and Bill all the time why he wasn't doing so well. So he was looking around for a vehicle. About the same time, in '76 the rest of the company was having a strong drive in reliability. With that drive in reliability, YHP discovered TQC at about that time. He got some education in TQC and felt like that would be a really good way to pull his organization together, to train them, to give them some motivation and some goals and to achieve excellence in their performance in all areas. One of the things he described is that we also weren't being real responsive to customer demands. You know, requests for new products and requests for improvements in old products. He referred to our sales techniques as kind of lordly. We've got the product, if you want it, it's for sale. That really wasn't a good way to interact with your customers. Consequently they were not penetrating the market very well. So, that's a little bit about YHP and how they weren't doing very well.

THE DEMING AWARD

Let me tell you just a few words about the Deming prize. It's awarded to somebody who makes an outstanding, measureable improvement in their quality efforts, either products or services, over a given year. It's awarded by JUSE, the Joint Union of Scientists and Engineers. There are four Deming prizes which may be awarded to a large organization, a small organization, an individual person, or an individual division of a company. There is a fifth award that's kind of a master's award. It's the QC prize. You're only eligible for the QC prize 5 years after you've achieved a Deming prize. So you can't just say well, we received the Deming award this year and next year we're going for the QC prize. Actually you want to rest for a while it turns out because a lot of work goes into winning the Deming award. You are self nominating, nobody turns you in, you have to decide if you want to achieve this award. You have to apply for it. And then there is a very very rigorous examination that occurs. The issues that they look for and apply are in the initial document cover policy. They include things like:

- What does your organization do?
- What do they say they're going to do?
- Does management actually achieve and act the same way policy reads?
- How well do you collect information, and understand it and use it to correct your process?
- Do you understand the problem analysis process in order to resolve problems and remove the cause of problems and make progress that way?
- Have you institutionalized your processes?
- Do you standardize your processes and are they really used?
- Is your process under control?
- Do you measure It?
- Describe the quality assurance organization and how effective it is.
- What's your future planning?

One of the things we've learned early, is if you don't measure your

process, it's really hard to know if you're out of control. In other words, if you win the Deming award that's just not the end of the career, you've got to have plans to go on. And each time you resolve a problem you also look for future efforts that need to be done. Where can you go from here, What can be done to improve even more? And that's a really important thing.

At Hewlett Packard we do a lot of what's called management by objectives. And so we kind of set some goals and if you aren't achieving those goals or you're not on the correct rate you take corrective action based on exceptions. They don't do that, they keep measuring progress all the time. Because they don't assume that the goal they set was perfect. Maybe you can do better and you should continually look for ways you can do better. And that's a pretty interesting difference there.

The 1982 prize winners were

YHP
KAJIMA KENSETSU (construction)
YAMAGATA NEC (semiconductor)
RHYTHM TOKEI (clocks)
AISHIN KAKO (auto parts)
SHINWA KOGYO (auto parts)
AISHIN WERNER (auto xmission gears)

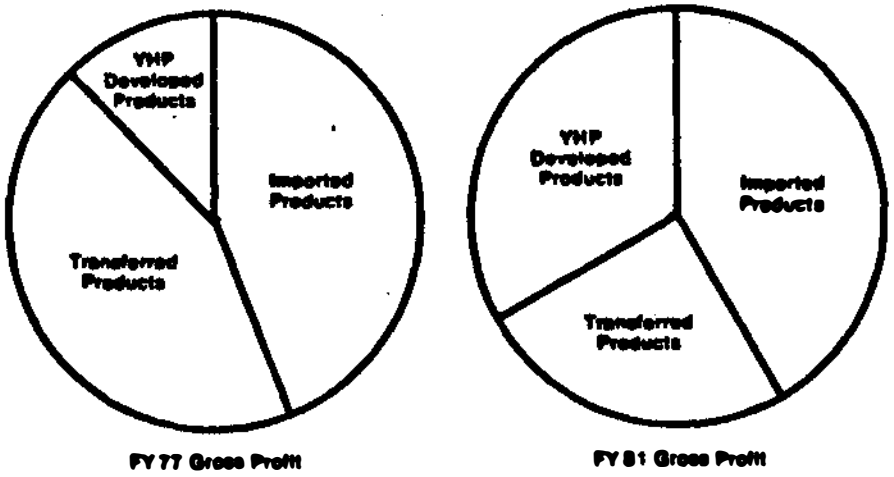
It's very interesting that in a country that is renowned for its electronics and cameras there were only two electronics companies that won the award in 1982. Now, we have no idea how many applied because if you apply and don't make it you just kind of disappear quietly in the west and they don't mention it. Only if you win is there any publicity and this kind of stuff is front page news in Japan. The Deming Award is really a BIG deal. It took a lot of hard work. Some of which the U.S. divisions of Hewlett Packard shared in and I'll tell you a little about that too.

The award was sought for and achieved by both the manufacturing operation and the sales and service organization, so it was a company wide activity including design, manufacturing, sales, service and support.

MOTIVATION FOR DEMING AWARD TRY

Well, why did YHP want to go after the Deming Award? I think I mentioned to you before that they were a little disappointed in their performance, they didn't like having to make excuses, and even further than that they felt that they would not survive as an entity if they didn't do a lot better in the Japanese market place. They were very heavily dependent on HP, we had expatriates over there helping them in manufacturing, R & D and sales. They had a few successes but they seemed to be pretty random. And they didn't have much market penetration, they weren't growing, they were just kind of staying flat while the rest of industry was growing. As I mentioned, their profit

wasn't very good and the employees were frustrated, they just weren't happy being in that kind of an operation. Also, as I said, in about 1976 they encountered TQC. I also talked about the contribution of products. There are two pie charts here.



FY 77 Gross Profit FY 81 Gross Profit

On the left hand side is the 1977 profit contribution based on transferred products, ones that we had given them manufacturing instructions and rights to, imported products where it was just being resold as other Hewlett Packard products, and the smallest portion of products they had developed themselves. By 1981, they had dramatically improved their profit percentage from their own developed products. Their overall growth was also dramatic so actually the right hand circle should be a lot larger. They had made tremendous improvement in their own products and profitability and market penetration. In fact by 1981 they had become world leaders in component measurement devices where they have achieved dominance in the Japanese market and a very strong 30%-40% market share in the rest of the world on some of their component measuring devices.

THE INITIAL AUDIT

So let's take a look at the chronology of what happened. In 1976 they invited JUSE to come in and give them a preliminary audit just to see how they were doing, and they weren't too surprised with the results. They weren't doing very well. But some specific advice was given that they should clarify and understand what total quality control meant in their organization and how to use it. They weren't using problem analysis methods that were very effective, the PDCA (plan, do, check and act) which was perhaps a translation of a more complicated phrase that they reduced for us English speaking folks but made it simple. There weren't any practical and visible examples of the use of statistics. They were just managing by the bottom line like many of us do. So the recommendations were, "Gee don't just listen to HP, you guys have to get up on the step and do something a lot stronger. Be

your own people. Be your own managers and masters. You have to understand TQC and train everybody from TQC" . And there was a lot of work done on that. In the 5 years between 1976 and 1982, they spent about 20,000 person hours in management training mostly done through JUSE. They hired two consultants on TQC and they did 18,000 hours of internal training. This is in a company with 2,000 employees and sales last year of $250 million dollars. So that sounds like a lot of training and it is. It was spread over several years but it was really required to achieve results and the results were dramatic as I'll show you in a little bit. They also suggested that YHP increase communications with HP. You know, let's help those guys across the water and transmit a little Japanese quality to your friends over there and see if we can't make them more successful, which they did do. And that, we're very pleased with.

IT STARTED IN 1977

So, in 1977, Ken Sasaoka said "We are going to embark on a TQC process. Not a project, but it's going to be a way of life. We're going to manage by TQC. Starting with manufacturing." Remember at that time they imported products and they manufactured products and they had a small amount where they designed them. The biggest contribution they felt they could make initially was in their manufacturing process where they had the largest impact and control. So they went after process improvement and I'll show you an example of that a little later. They did make a committment to training as I mentioned. And they set quality objectives in every organization.

In the next couple of years they expanded the TQC processes to R & D and Marketing, which one might expect is a little harder to do. It's like expanding it into software. It takes a little bit of thinking to figure out how to use statistics in software, but it is possible. They went after inventory management using JIT, Just In Time, inventory control to cut their costs. They established a tremendous number of quality circles. They had just been on the edge of that and they just blossomed greatly and also influenced the rest of Hewlett Packard in that endeavor.

And they started a top management audit. Again, the president of the company, Ken Sasaoka, personally audited not your financial performance, he did that too, but he spent a full week every year in a quality audit. What are you objectives, what are your points of measurement, what are your successes, what corrective action have you taken and what future plans do you have? He does this every year still today. It's very very effective when the President comes down and asks you about your quality and you think "Gee, I better know about my quality because that's probably important."

THE FIRST TRY

In 1980, they felt that they were far enough along that they could try

for the Deming Award. Remember now, this started in 1976. They were training, they were working on it, they had some internal audits, they had some external audits, but they hadn't really said they were going for it until late 1980. They also were continuing to expand TQC, had taken on field service, and had started their standardization process. They started documenting things. And they wanted to document them in a way that they were living, working documents, not just a set of books on the side wall that when you get the audit you say "There they are over there." These folks really understood and used them and they were hierarchical, meaning they started from the mission and the objectives of the organization, the objectives of the department, the quality performance measures, the financial performance measures, sales and so forth. And if the process wasn't working, they would go back and revise their method of doing it. So they really did an excellent job on documentation control and I've recently been over there and you'll see some people, you know, plastic envelopes on their desk, permanent plastic, with process flow documentation showing what's supposed to be done, who's responsible, what level of decision they can or cannot make and where you go if you have a problem. The amount of documentation they have done is very very impressive. Not much by computers either. It's almost all hand written too. That's one of the problems you have in Japan, printing the Katakana and the Kanji is difficult, so everybody writes everything by hand or they type it in English. So it's a difficult process and there's still a lot of scribes over there today.

THE SECOND AUDIT

They had their second audit and they were not real thrilled with the results because they thought they had come along pretty well. But they again found out that they hadn't internalized this PDCA, this plan-do-check-act process. They hadn't gotten the people across the water, us, involved in very many of their projects. And there were no visible results again from the statistical quality control. They were a little disappointed in that. And it was recommended that they do these things and correct those problems.

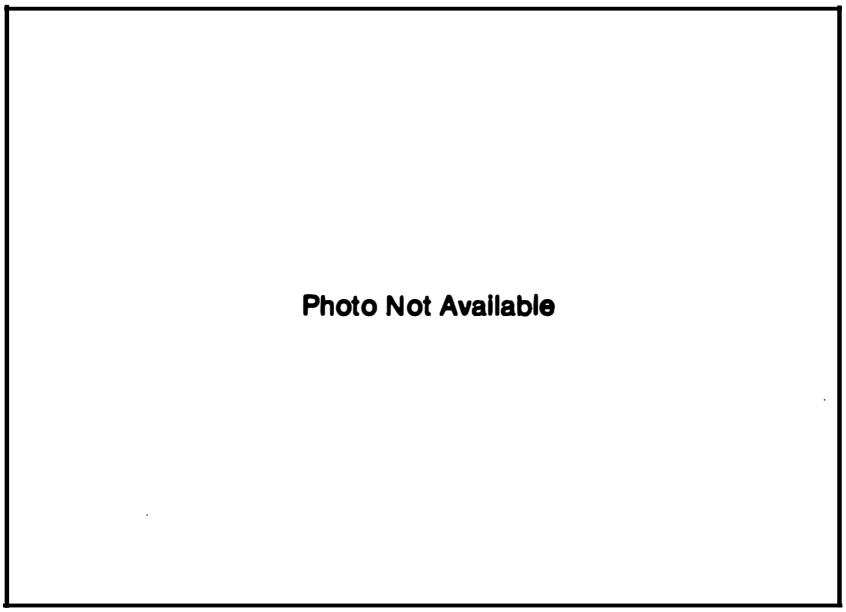
AFTER THE AUDIT

After the audit many joint quality circles were established between us and the Japanese division on specific products. The 9845B desktop computer was manufactured both in Japan and in the U.S. The Japanese one had a failure rate that was about twice as bad as the U.S. Part of the reason was their applications. In the Japanese businesses they don't air condition their offices. By contrast most of the U.S. applications are in comfortable air conditioned places like we like to be in. Our computers also like air conditioned environments and they didn't like the high humidity and the high temperature in Japan. In fact, they broke a lot. So YHP attacked that problem to find the causes for the high failure rate. Well, the cause was bad design. We hadn't designed for their application. You know, we were being a little lordly once again. So we took corrective action on that in the 1982 period.

We needed to work on TQC everywhere in the organization to really make it a working event, help the Americans out a little bit, and make quality information available and visible throughout the organization. If you go over there today, you'll see along the wall, many , many, walls are covered with graphs and charts describing projects that were worked on, what the problem statement was, what the analysis was, what the proposed solution was, the test of that proposed solution, the results of that and then future activity required. These are painted on the wall on paper with really big graphics. I mean they would fill this room three times with this kind of documentation throughout the plant in marketing, manufacturing, and elsewhere, including R & D.

THEY WON

They were successful in their endeavors. Here is a picture of Ken Sasaoka accepting the award from Dr. W. Edwards Deming.



They were very pleased with their effort. But Ken Sasaoka's speech to the entire gathered assembly, after they had won the award, was one of admonition for further work. He said "This is just a beginning. The efforts and things we have done today are excellent. But they are just shining stars if we have only picked specific areas and made successes out of them. Now we need to transmit this success, institutionalize it, and bring it up throughout the organization, the field, sales and R & D." I think maybe his admonition made people a little tired because they had been working very hard.

THE PERSONAL SACRIFICES

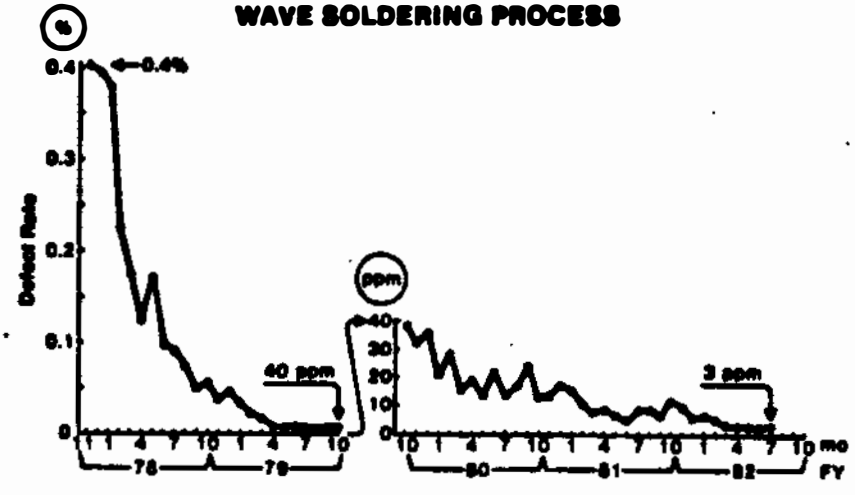
For the last 9 months just prior to receiving the award all the supervisory staff was working 80 to 120 hours a week. How many hours are there in a week? There are not many more than that. Some companies have had people die in the effort from putting so much out for it.

During the last month or so they brought cots in and people were sleeping at the plant. I asked them if there were any deaths or divorces and they didn't think that was too funny. In fact, HP did not have any deaths, I don't know about the divorces.

Kenzo Sasaoka did write a letter to every spouse of every supervisor and manager asking for their forgiveness during this period when the company was working so hard to achieve this award. And they really poured it on. They probably did a lot more than was necessary to win the award. But they certainly learned a tremendous amount about it and they shared a lot with other HP divisions in that endeavor.

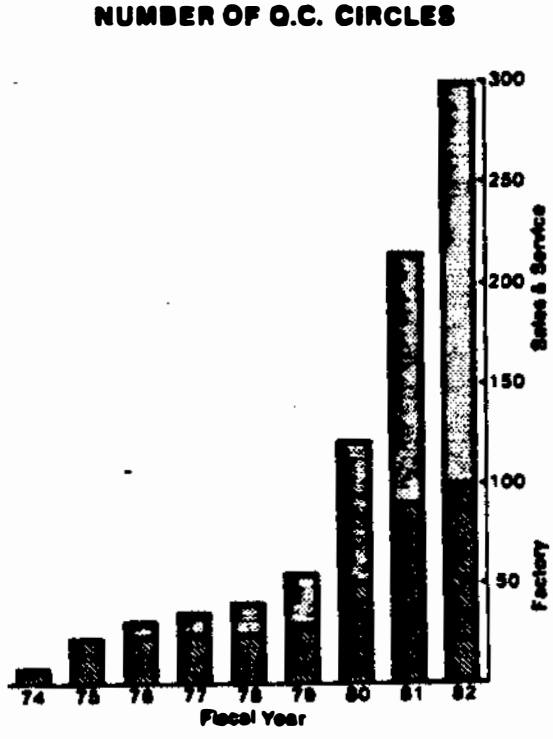
WHAT THEY ACHIEVED

Let me show you some of the examples of what kind of things they did achieve. As I mentioned, they started out in manufacturing and this particular graph shows the wave soldering process which was their shining point as they talk about it.



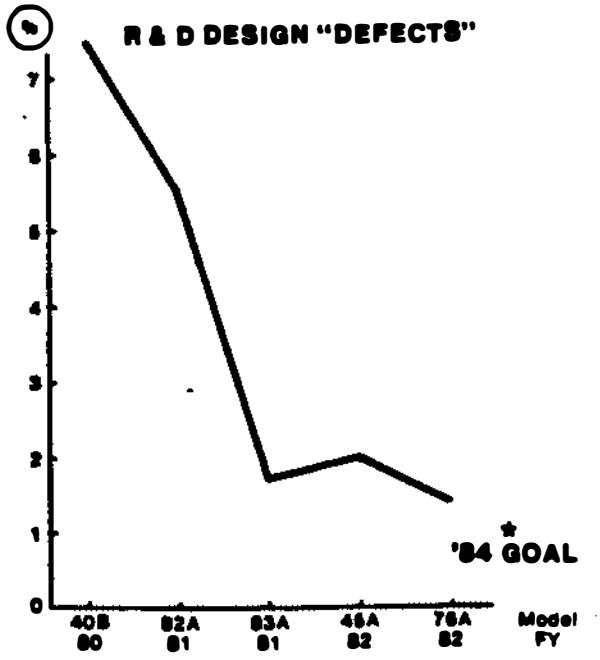
They keep using flowers and shiny things in their conversation. It's pretty neat. Talking about budding opportunities or flowering opportunities in their sales organization. But they started out in 1978 measuring their percent defect per hole in the wave solder process. It was .4%, that is, 4 defects per 1000 holes. Now you're probably all familiar with the PC operation where you take a PC board and you stuff it with components and go across a wave solder operation. Then you turn it over and you check out what happened. You look for solder bridges or bubbles that are missing. And that was kind of average for the corporation at the time. The interesting thing about this wave solder machine was it was declared obsolete and surplus in 1973 down in a Hewlett Packard division in California and we kind of helped them out. We gave them this toy to work with so they could have a low cost wave solder machine. They ran it fine and were doing just as well as the rest of the folks in reliability and quality, but as I said, they had decided that they were going to hit manufacturing and understand the causes of these problems. So with this machine, which we retired in 1973, they had been living with the junk it had been turning out up until 1978. Then they really went after it. And as you can see by the end they got down to 3 parts per million. So this 18 year old machine which is as long as I've been at HP is about 10 to 100 times better than any other machine we have-and we want it back! But it's not the machine, it's the process. Now as they were going down their improvement process every time they went out of control from their target improvement rate they analyzed the cause. I happen to know a couple of them. At this point they undertook a civic acitivity where they hired a lot of older people from the local community to come in and be helper workers to expand their labor force and at the same time help out the retirement community. These folks didn't understand the process very well and management thought they were going to the moon again with increasing yield problems. So they analyzed the problem, trained the people, made them more careful and made them understand it. They turned it around. Down in this area they went through a process they called foolproofing which was really making the instructions extremely clear and the process extremely straight forward. And they still are driving as hard today at 3 parts per million as they were back at 40 parts per million, you know. They are really working very hard and being very successful at it.

As I mentioned, they thought QC Circles were going to be a big help to them in analyzing and understanding problems and also in training people in the problem solving method. This graph shows their QC growth from 1974 through 1982.



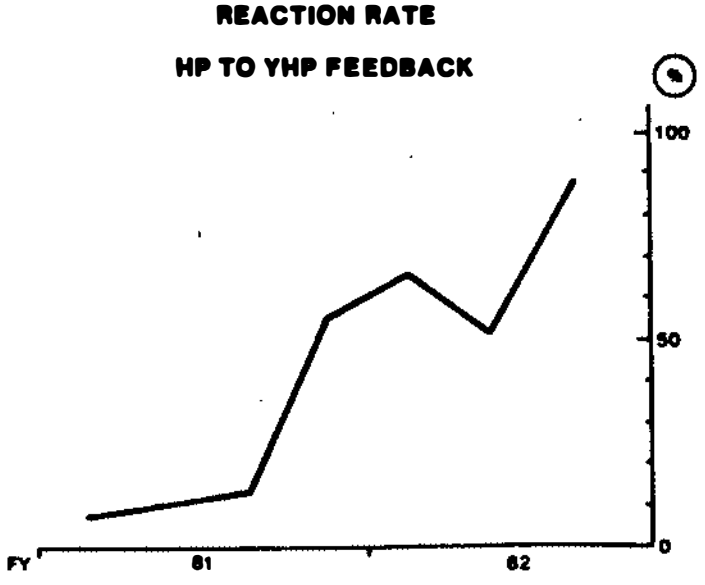
As you can see, now they're up to 300 quality circles in their company. Most of them now are in the field, in the sales organization and the service organization. That's pretty unusual I think. Most often you think of them as being a standard factory operation. They have almost 90% participation of all their employees in QC circles. And they have almost half of all the QC circles in Hewlett Packard. We have about 700 total worldwide and 300 of them are at YHP. So there is a tremendous amount of activity in that area.

Let me show you another area that was measured, understood and improved. And that's R & D design defects. This was not even a term I had heard of until I went over there. What they're measuring in this graph is the amount of engineering effort spent after a product is introduced vs. the amount of engineering that went into it during the development phase.



This is like sustaining engineering or what we call CPE or Continuing Process Engineering. You'd like that not to be a very big number you know, and there are two ways you can make that not be a very big number. One, you can say "forget it, we're done and we're not going to work on it again," which is not what they chose to do. They chose to respond to customer requests for improvement. This is arranged not chronologically but by product number as it was introduced in earlier years so it's not a linear scale. Pretty impressive progress, though.

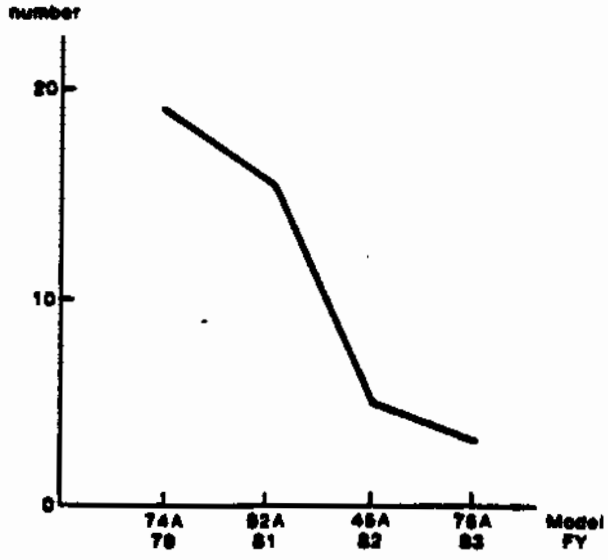
Here's another example of their improvements. As I mentioned they were trying to get some help from the U.S. divisions and they weren't really successful in that. This graph shows the response rate to their pink sheet or corrective action request.



They were developing a formal way to communicate with the U.S. design divisions to make corrective action. And as you can see back in '81 we just thought this was some kind of a strange letter from the east that anamolied a very small group of the market place and didn't deserve much attention. We were pretty busy with our own stuff you know. But we got on the band wagon and really committed to help them out. As you can see we're answering the mail almost 100% of the time, but not all of it. Some of them are tough questions like "Why don't you redeisgn this and double the memory in the same space?" Pretty tough questions.

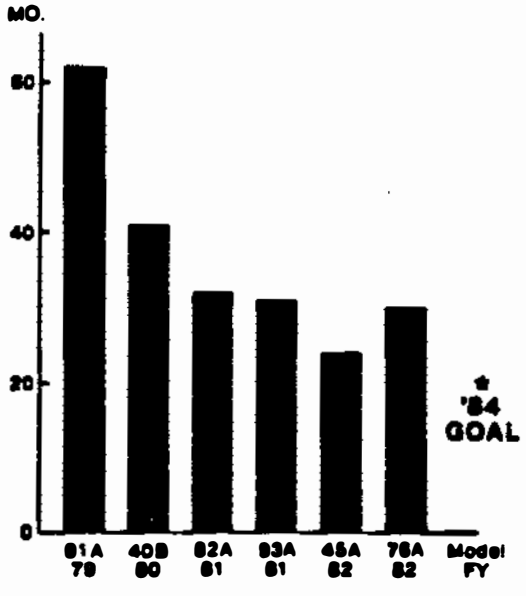
Here's another question they ask themselves on their own design products. When they go out to design a product they interview their customers on what kinds of features are needed. Then after design is complete they measure how many features they actually got in there that their customers asked for. Again they developed a goal to get that down to be a very low number.

OVERLOOKED MARKET NEEDS



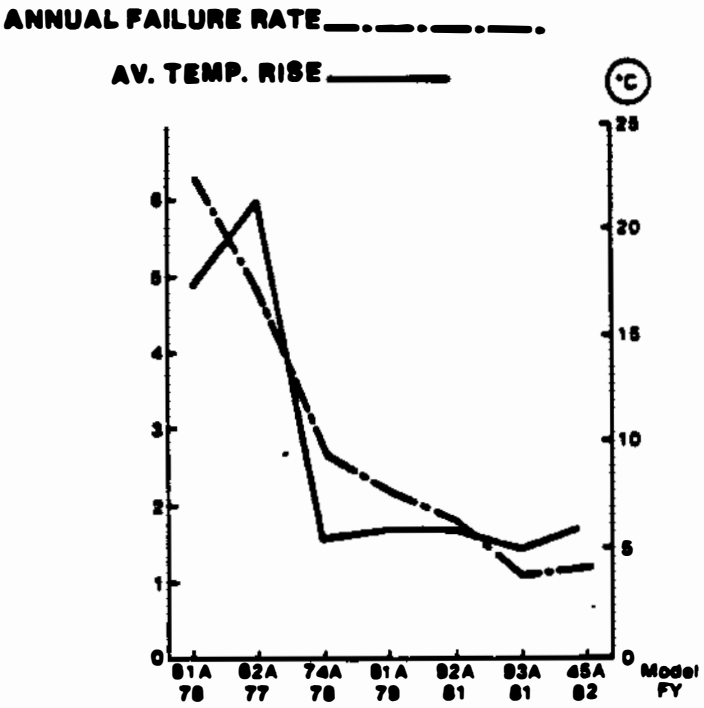
I'm going to race through the last few here.

R & D CYCLE TIME

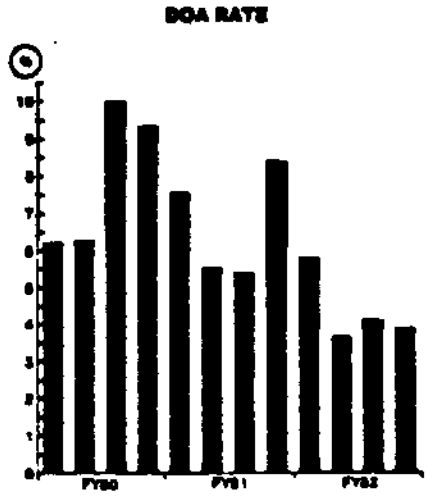


R and D cycle time is a very large productivity measure. How long does it take you to invent a product? When they started measuring this it was 60 months, 5 years. The average for a U.S. division in Hewlett Packard is about 3 years including the software and hardware. Their goal is to get that just under 2 years turnaround.

I want to show you a chart here on failure rate.



This one is normalized against the sales price of our product to level out the product complexity or price. So this is, over time, the annualized failure rate per thousand dollars of product price and overlayed on it is the temperature inside the product. They found they correlated failure rates to temperatures very very strongly. In solving this problem, they improved the reliability of their product and they improved the productivity of R & D. At first, when they were back here making excuses they said "well, we can't really test the temperature rise 'til you give us a good product to test. So, you bend up the sheet metal, you give us the boards, you give us the prototype and we'll check it out and tell you it's bad." Which they did. Then to improve this situation they said "why don't we figure out a way to test these things before we've invented the product?" So they figured out the functionality, they modeled the IC's in the proposed design and the heat rise, and the anticipated use of heat sinks, and they made a mock up out of cardboard. They put transducers inside. And they put in little heat sources, and they measured how hot is was going to be and where it was going to be hot. Then they'd redesign the sheet metal and the cooling and the layout of the circuit. They eliminated the problem before they even bent any sheet metal. And they got down to remarkably good reliability levels. Again, they were facing a marketplace that doesn't air condition, so it was a pretty clever corrective action.



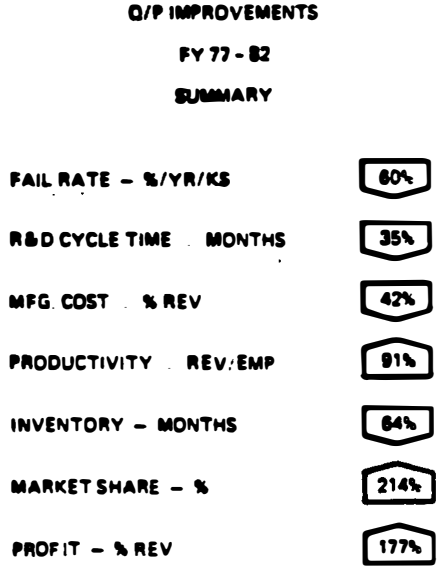
Here is another one that is my least favorite. It is the defect on arrival rate. How many of you have ever bought anything from Hewlett Packard? Nobody?, oh! Lots of you! OK. How many of you have ever, perhaps, not gotten a manual or didn't get a cable or there was a missing IC or the option was wrong. 1, oh, 2,3, and some very polite people out there I'm sure. In Japan they were seeing 10 or 15% and they call that DOA. Not just a hard failure which we would normally count as a DOA, but if there was anything missing that the customer expected, they said sorry, and it's a 0 and 1 game and that's a 0. And they graded themselves against that. It is now down to about 2% and they're shooting for 1%. That still is a terrible number. I mean, you shouldn't get anything wrong. That should be the easiest kind of problem to solve. So we're working on that.

SUMMARY

Let me just summarize in the last minute or so here to show you some traditional measures and what their progress was from 1977 to 1982.

Their failure rates dropped by 60% across the board. Their R & D cycle time was cut by one third, and that's on significant projects. Their manufacturing cost was almost cut in half. Productivity was almost double. That sounds like a reciprocal. Their inventory was reduced. They really went after just in time inventory and that reduces your costs a lot. Their market share had been nearly flat and it more than tripled increase in percent. And their profit went up by 177%.

Now this was a division that I told you was a kind of a sad sister. For the last two years they have been one of the leading divisions in Hewlett Packard in profit. And they didn't do it by attacking the bottom line. They did it by attacking quality. So the final message I want to give you is it takes a long time, it's not a sudden process. It takes a tremendous amount of training. And it takes top management committment. It's very difficult to do from an entrepreneurial middle level. You really have to have top management committment. But it's pøssible! We're trying to take these same things and institutionalize them in our U.S. divisions and take advantage of them and make some of the same progress. Thank you very much.



SOFTWARE COMPLEXITY MEASURES

Curtis R. Cook
Computer Science Department
Oregon State University

The text of this paper was transcribed from videotape. It has been edited for clarity in written form but purposely left in a distinctly conversational tone. Slides and pictures have been included where appropriate.

Here is a broad outline of my talk today so you'll know where I am and where I am headed.

OUTLINE

I. WHAT IS SOFTWARE QUALITY?
II. SURVEY OF SOFTWARE COMPLEXITY MEASURES
III. CLOZE PROCEDURE AND EXPERIMENTATION
IV. VALIDATION OF MEASURES
V. CONCLUSIONS AND PERSONAL SUGGESTION

First I'll define what software complexity is. Secondly I'll survey some of the current metrics and talk about them. What I will do in the survey, is pick a few specific examples of a given class, talk about those and survey those examples instead of trying to survey every possible measure. Then we'll do an experiment with the CLOZE procedure and it will lead into a discussion of experimentation. This really falls under IV., how do you really validate a metric? How do you validate that something really measures what it is supposed to measure. Then, I'll conclude with a summary and a personal suggestion for a start at what might be considered a software complexity measure. I won't stick my neck out that much, and you'll see it is a very general suggestion. This is a very new and evolving area. No one has the answers.

I.WHAT IS SOFTWARE COMPLEXITY?

When you mention the word complexity in computer science, probably the first response will be something in the area of computational complexity which is the complexity of the algorithm or how the algorithm performs. It is usually stated in terms of time, ie."How long does it take to work?" It includes Beta structure and things like this. So what you have in terms of computational complexity, as an example, would be a sorting algorithm. You'd say that I have an n log or sorting algorithm, where n is the size of my list. What is referred to as computational complexity, talks about the performance of the program or the algorithm. That's not what I'm interested in.

What I am interested in is the software psychological complexity. This deals with the human performance aspect of the program. That is, how difficult is it to work with this program if you are maintaining or testing it? Or, if you're using the program, how easy is it to use? How easy is it to learn to use? Things like this fall in the area of

software psychological complexity. For the purposes of this talk I'm going to limit the aspects of sotware complexity to those that consider the difficulty of a programmer working with a program, for example, testing it and maintaining it. We're going to try to look at how you could measure a program and see how easy it is to maintain or test.

It is important to study software. If you look at the typical stages in software development, the first number in parentheses after each of the steps, represent the percent of time during development spent on that aspect. The second number refers to the percent of the total time spent on the project including time after the program has been developed. As you look at this you can see that a major part of the development effort is spent on testing. In terms of total time, development time is dominated by the maintenance cost. So testing and maintenance are the two major cost components of software.

IMPORTANCE OF SOFTWARE COMPLEXITY (PSYCHOLOGICAL COMPLEXITY)

1. SOFTWARE DEVELOPMENT LIFE CYCLE

   STAGES

   1. REQUIREMENTS ANALYSIS: DEFINE REQUIREMENTS FOR AN ACCEPTABLE SOLUTION TO THE PROBLEM. (10%) (3%)
   2. SPECIFICATION: PRECISE DESCRIPTION OF WHAT IS TO BE DONE; FUNCTIONAL SPECIFICATION. (10%) (3%)
   3. DESIGN: DEVELOPMENT OF ALGORITHMS AND OVERALL STRUCTURE OF SYSTEM. DESCRIBES HOW IT IS TO BE DONE. (15%) (5%)
   4. CODING: WRITING OF ACTUAL CODE. EASIEST STAGE LEAST TROUBLESOME. (20%) (7%)
   5. TESTING: DEMONSTRATION THAT PROGRAM MEETS SPECIFICATIONS.
      MODULE TESTING (25%) (8%)
      INTEGRATION TESTING (20%) (7%)
      SYSTEMS TESTING - DONE BY INDEPENDENT GROUP; PART OF USER ACCEPTANCE REQUIREMENT
   6. OPERATION AND MAINTENANCE: ( ) (67%)

REFERENCE: M. ZELKOWITZ, A. SHAW AND J. GANNON, PRINCIPLES OF SOFTWARE ENGINEERING AND DESIGN. PRENTICE-HALL, (1979), PAGES 2-10.

There's a lot of data out on maintenance. It is quite interesting. The Department of Defense studied some of their programs. This was after they had been tested and delivered by the contractor. They spent 60% to 70% of their money after that time. In terms of errors, you would guess that the maintenance effort was spent on correcting errors. Yes, coding errors were 30% of that and the other 70% were actually specification errors that had to be corrected. In terms of maintenance there are three classes: corrective, adaptive, and perfective. The last two, adaptive and perfective actually occupied more time than getting the errors out. If you get a new machine, a new operating

system, some change in the environment, and the perfective is to make it work better. Probably a good quote is the one that I have up here from Perlis, Sayward and Shaw's book. They indicate the need for better understanding of the software development process and the software development product. What is needed is metrics or measures to help us compare different products and different development processes. So that is the problem. You're comparing different products and different processes. How are you going to compare the two? It's like comparing apples and oranges.

MAINTENANCE CLASSIFICATION

1. CORRECTIVE - FIX ERROR IN SPECIFICATIONS OR CODE.
2. ADAPTIVE - MODIFY FOR ENVIRONMENT CHANGE.
3. PERFECTIVE - IMPROVE OR AUGMENT PERFORMING CAPABILITIES.

"NEED BETTER UNDERSTANDING OF THE SOFTWARE DEVELOPMENT PROCESS AND THE SOFTWARE DEVELOPMENT PRODUCT... METRICS OR MEASURES HELP US COMPARE DIFFERENT PRODUCTS AND DIFFERENT DEVELOPMENT PROCESSES."

A. PERLIS, F. SAYWARD AND M SHAW
SOFTWARE METRICS
MIT PRESS (1981)
PAGE 150

If you do have a measure, you might ask "what use would it be?" One obvious use for a programmer would be feedback on whether the program they've created is going to be difficult to test or difficult to maintain. Managers could use results from the measures to estimate resources needed to test or maintain programs. Both programmers and managers can use the metric results.

II. SURVEY OF SOFTWARE COMPLEXITY MEASURES

That's the motivation part, now let's get into the complexity part. What I have done for the survey part is break the complexity measures into four classes:

1) based on syntactic structures
2) based on flow of control
3) based on module interconnections
4) based on combination of program properties

My classes are not perfect, there can be some overlap. Also there have been some classes of measures that have been ignored. One major one would be a class based on entropy. For each of the four classes I will select one or two representative measures and talk about each one in the following detail:

1) describe what it measures
2) give some indication of it's usefulness
3) show ease of computing the measure
4) identify limitations

A lot of what I'll be saying is influenced considerably by two survey

papers by Bill Curtiss from ITT.

Let's look at the first class and just one example, Halstead's E or effort measure. The measures are based on syntactic structures. Halstead has by far thoroughly dominated this class. He developed an area called software science. He was at Purdue University and died about three years ago. What he was attempting to do was begin a scientific investigation of algorithms. He had a very simple basis for what he was doing. Everything that he did was based on four basic measurements (counts).

SOFTWARE SCIENCE

4 BASIS MEASUREMENTS (COUNTS)

- n_1 = THE NUMBER OF DISTINCT OPERATORS APPEARING IN PROGRAM.
- n_2 = THE NUMBER OF DISTINCT OPERANDS APPEARING IN PROGRAM.
- N_1 = THE TOTAL NUMBER OF OCCURRENCES OF OPERATORS IN PROGRAM
- N_2 = THE TOTAL NUMBER OF OCCURRENCES OF OPERANDS IN PROGRAM

An operand is quite easy to define, it's usually a variable or a constant. The operator is the one that usually leads to the greatest difficulty in applying what Halstead has done. He defined an operator as a symbol or combination of symbols that affect the value or ordering of an operand. Halstead is quite interesting in that when he did his work in the early '70s it was not highly thought of by the academic world. But in the last five years, there has been quite a rebirth of interest in his ideas. Let me give you an example showing Halstead's count in a program.

EXAMPLE

```
      SUBROUTINE SORT(X,N)
      DIMENSION X(N)
      IF(N .LT. 2) RETURN
      DO 20 I = 2,N
         DO 10 J = 1,I
            IF(X(I) .GE. X(J)) GO TO 10
            SAVE = X(I)
            X(I) = X(J)
            X(J) = SAVE
10    CONTINUE
20    CONTINUE
      RETURN
      END
```

OPERATORS

			COUNT
	1	END OF STATEMENT	7
	2	ARRAY SUBSCRIPT	6
	3	=	5
	4	IF( )	2
	5	DO	2
	6	,	2
	7	END OF PROGRAM	1
	8	.LT.	1
	9	.GE.	1
n_1 =	10	GO TO 10	1
			28 = N_1

OPERANDS

	1	X	6
	2	I	5
	3	J	4
	4	N	2
	5	2	2
	6	SAVE	2
n_2 =	7	1	1
			22 = N_2

I pulled this example out of the computer survey article. It is a very simple sort program in FORTRAN. Beside the program, I have given you the operators and operands count. Notice there are some strange operators. Number one, the end of a statement is an operator. There are seven statements so I have seven for that particular count. An array subscript, is an operator, and there's six of those. The equal sign is an operator. Most people accept that. IF, is an operator, number five is a DO and that is fairly obvious. Number six is a comma, that's an operator that's a little unusual. End of program is an operator as are the two logical operators LT and GE. GO TO 10 is another unusual operator, and if I had a GO TO 20, that would have been another distinct operator. I think the operands are fairly obvious. I want to point out here that the arithmetic operators are not exactly obvious what they should be. This should give you a little flavor for what Halstead's counts are and how to use them.

MEASUREMENTS OF PROGRAMS

1. SIZE OF VOCABULARY $n = n_1 + n_2$
2. LENGTH OF PROGRAM $N = N_1 + N_2$
3. ESTIMATION OF PROGRAM LENGTH $\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$
4. PROGRAM VOLUME $V = N \log_2 n$

   NOTE: PROGRAM VOLUME INCREASES FOR LOWER-LEVEL LANGUAGES
5. PROGRAM LEVEL (LEVEL OF ABSTRACTION) L

   CONSERVATION LAW: LV = CONSTANT

   $L = V^*/V$ WHERE V^* IS VOLUME OF HIGHEST-LEVEL REPRESENTATION OF PROGRAM

   LEVEL ESTIMATOR: $\hat{L} = (2/n_1)(n_2/N_2)$
6. LANGUAGE LEVEL: $\lambda = LV^* = L^2V$
7. EFFORT MEASURES MENTAL EFFORT REQUIRED TO CREATE PROGRAM.

   $E = V/L$

   EFFORT ESTIMATOR: $\hat{E} = V/\hat{L} = (n_1 N_2 (N_1 + N_2) \log_2 (n_1 + n_2))/(2N_2)$

   NOTE: E IS MOST COMMONLY USED COMPLEXITY MEASURE.
8. TIME REFERS TO PROGRAMMING TIME. $T = E/S$

   WHERE S IS THE SPEED OF THE PROGRAMMER

PUBLISHED STUDIES HAVE SOME CLOSE CORRESPONDENCES AND SOME NOT SO CLOSE.

ORIGINAL HYPOTHESIS (1972) WAS THAT THE COUNT OF OPERATORS AND OPERANDS IN A PROGRAM IS STRONGLY CORRELATED TO THE NUMBER OF BUGS DISCOVERED IN A PROGRAM.

Halstead defined various measurements of a program based on counts. The measures 1-4 and 7 are of interest in our discussion. Calculating the vocabulary and length of program are obvious. They are simply the

sum of the numbers of occurrences of operators and operands. He also came up with an estimator for the length of a program. He said that it was the log formula.

The one I want to focus on first is the program volume. That is the sum of the number of operators and operands and the log of the number of distinct operands and operators. The logic behind his development of that log two formula represents the number of bits that you need to represent all the operands and operators. What's interesting in this formula, is that if you have a higher level of language, you usually have a richer set of operators and hence your program lines will decrease. So it agrees some with our intuition.

Let me go down to number seven. This is one of interest to us. This is the effort or E measure. It is supposed to measure the mental effort required to create the program. As it turns out it is the volume of the program divided by the level indicators. You would think that the lower the level of language the more effort would be required to create the program. Now this E is the Halstead measure that is most commonly used for measuring complexity. If you want to measure the complexity of a program using the E you count the number of operators and operands and then plug each number into the E formula and that's the complexity. Halstead did not use the term complexity, he just used effort, but others have used that as a measure of complexity.

There have been a lot of published studies on Halstead's work.Some have shown a close correspondence between his values and what has actually occurred in practice. And there have also been some that have not corresponded as well, so the reviews are quite mixed. What he was really trying to show was a correlation between the number of errors in a program and the number of operators and operands. He thought that the count of the operators and operands were somehow correlated to the number of errors. That was the original hypothesis under which he worked.

Let's comment a little bit about the usefulness of Halstead's measure. Researchers have applied Halstead's work to experimental data and empirical data from controlled experimentation. They have found high correlation between the E number and how difficult it is to understand a program, time to locate a bug in a program and the number of errors in a program. They have also shown examples where the correlation is not so high. It is not clear at this time whether the Halstead E numbers are good predictors. So there have been a lot of studies and some high and not so high correlations. We'll talk a little more about the experimentation and validation things later.

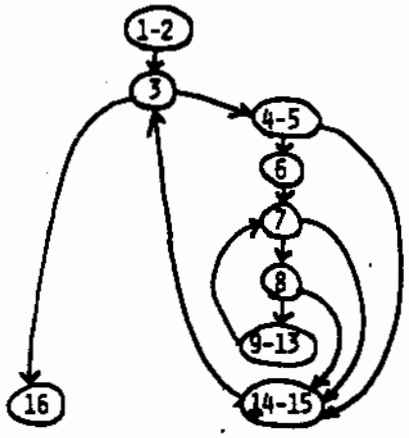
Ease of computation is very simple, once you have those counts. However, the limitation is that the definition of operators is extremely ambiguous. It has been shown that you can get fairly substantial differences depending on our use and definition of the term operator. If you look at E, the effort measure doesn't include the data structure used, variable names used, comments, anything like this. All that information about the program is totally ignored by the E. You'd think that should somehow be involved in the complexity.

The second class are the measures based on flow control. There are two that we will consider here. The first is McCabe's measure of cyclomatic complexity which is quite well known. It agrees a lot with intuition. The second one is called the knot count.

McCabe's measure is based on graph theory but the interesting thing is that when it comes to actually computing it you don't need to even worry about the graph. Let me give you an example of a program and then we'll look at a graph that's derived from the program so we can easily define McCabes measure. Here's simple bubble sort in sort of a combination of languages.

EXAMPLE

```
1    SUBROUTINE BUBBLE (A,N)
2        BEGIN
3        FOR I = 2 STEP 1 UNTIL N DO
4            BEGIN
5              IF A(I) GE A(I-1) THEN GOTO NEXT
6              J = I'
7    LOOP:     IF J LE 1 THEN GOTO NEXT
8              IF A(J)  GE A(J-1) THEN GOTO NEXT
9                TEMP = A(J)
10               A(J) = A(J-1)
11               A(J-1) = TEMP
12               J = J-1
13               GOTO LOOP
14   NEXT:     NULL
15           END
16       END
```



FLOW GRAPH

You can easily see the flow graph from the example program. Each node in the flow diagram corresponds to a basic block in the program. A basic block is a sequence of program statements and the property that if you start executing the ffirst statement in the program, then you will execute all the remaining statements in that block and the only branching is from one statement to the next sequential statement. In the last statement, you can branch to various other places. For this particular program, statements one and two are a block. Statement 3 is a block by itself, because in the FOR statement you can either go through the loop or exit. Then inside the loop the BEGIN and the IF are together and then after the IF statement you hae a choice of going to the end or going to the next statement, etc. So this flow graph can easily be formed from the program. A lot of test measurements use something like this flow graph. It turns out in a nice fashion that you really don't even need to construct this flow graph to take the measurements, but, we'll see that later.

McCabe's cyclomatic complexity measure is the cyclomatic number of the program control graph for a program. The cyclomatic number V(G) of a graph G with N vertices, E edges and P connected components is V(G)=E-N+P. When McCabe originally proposed his measure, what he was aiming at was that just saying that 50 statements was as large as a module should be, or 100 statements or any other number of

statements, is not a good rule. He thought that the complexity of a module depended on how many different ways and different paths there were through the program from start to end. If you look at the graph, how many different ways are there through the graph? That's the complexity . However, for a reasonable size program, the number of distinct paths is a very large number, so it is impractical to compute that. As a compromise, instead of computing all the paths, let me compute all of the basic paths. Once I have my set of basic paths, then every path in the program can be expressed as a combination of these basic paths. The more basic paths you have, the more paths you have in the program. It's much easier to compute the basic paths rather than all possible paths. It turns out the number of basic paths is actually the cyclomatic number of that program. What the cyclomatic complexity is attempting to measure, is the number of basic paths in the program. Meaning every possible path can be expressed as a combination of these paths.

Now, the usefulness of this is that it is actually very commonly used and the reason is probably that it is so easy to compute. It turns out the way you compute the number of basic paths is count the number of predicates in your program and add one and that's the cyclomatic complexity. You just go in with FORTRAN, count the number of DO and IF statements and add one and that's it, unless you have GO TO statements and then you have to add these too. You don't have to construct any graph at all. And it turns out when Farr and Zagorski did a study in 1965 they found that the density of the IF statements was a major factor in the comlexity of the program. This is exactly what McCabe is saying, the density of the IF statements is a measure of the complexity.

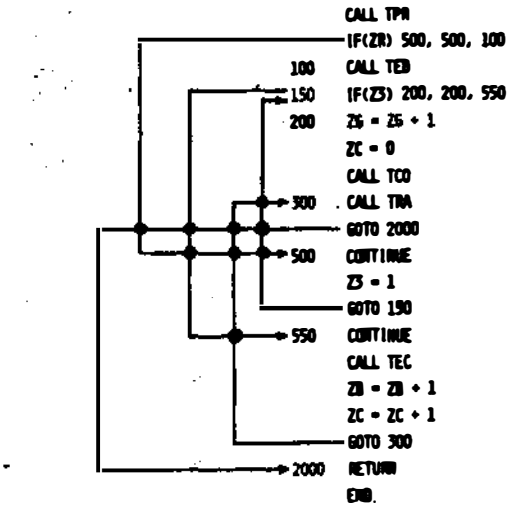
Probably the reason for McCabes measure being so popular is the fact that it is so easy to compute. You can forget the graph completely and just count the number of predicates.

McCabe recommended rather than limit a module to 50 or 100 statements, that no module should have a cyclomatic complexity of more than 10. In general, if you have a module that has a complexity of more than ten, you should consider rewriting that module. He defines the complexity of a program with many subprograms to be the sum of the complexities of the main program and the subprograms.

The shortcomings of McCabes measure is that there is no taking into account data structures, use of Mnemonic variable names, the comments, choice of algorithms, portability, flexibility, or efficiency. There's a lot of factors there. That's McCabes measure. The most common of this class and widely accepted.

A second measure we'll talk about is the knot count. For this one you draw a flow control arrow in the left hand margin of your listing from a statement to whatever statement it branches to if it's not the next statement. Whereever two of these arrows cross you have a knot and the complexity is the number of knots in the program. Let me put up an example. Here's a program and I have drawn the arrows in the left hand margin. In this program there turned out to be nine knots. I only draw arrows when there's a branch from one statement to other than the

next sequential statement. This example is interesting in that by rearranging parts of the program you can actually reduce the knot count of this program. It starts out with nine, but if you rearrange things, the knot count can be reduced to two. What is the usefulness of the knot count? The authors felt that besides measuring complexity it measured the unstructuredness. The important difference from McCabe's measure, is that the number of knots is highly dependent on the ordering of the statements in the program.



The McCabe measure would be the same for that program, because I have the same number predicates, whether I rearrange the statements of not. So the knot count does measure the ordering or structuredness of the programs. If you have spaghetti code, the knot couunt gets you. It's not really difficult to compute the knot count. In fact, maybe as you are reading the program you sort of do the same thing, drawing arrows to where you are going. The limitations are essentially the same as the McCabe metrics.

There are several other measures that are based on flow of control. A lot of these measures deal with levels of nesting. Nesting of control structures is one common thing that these other measures go after. McCabe's measure doesn't address nesting at all.

The third class we're going to talk about is based on module interconnections. The one that I'll talk about here is Henry and Kafura's information flow. These measures go after the communication between modules, figuring the more communication, the more complex the program. In their definition, they define the FAN IN and the FAN OUT of a given module. Rather than reading all this I think you can visualize that the FAN IN would sort of symbolize the number of communications coming into a module and the FAN OUT of course, is calling some other module. The important thing to look at, is the formula. Their complexity measure is the length of the module times the FAN IN times the FAN OUT and that quantity squared. That's the base that deals with complexity. So you see if there is a lot of FAN IN/FAN OUT activity, notice that term squared, is going to increase quite a bit. The length here could be either Halstead's E or MCabe's V(G) for a very simple reason. They use either number there because a lot of studies have shown that the number of statements for Halstead's E and McCabes V(G) measures don't vary much in terms of some

experimental data. Henry and Kafura feel that the FAN IN and the FAN OUT quantity squared is the most important aspect.

The reason they chose squared, was that's what their data indicated. To determine the usefulness of this metric Henry and Kafura analyzed data from the Unix system, because it's a fairly well documented system in terms of the changes that were made. They assumed that whenever there is a change reported, that's probably an error that had to be corrected. So they analyzed the data and found a high correlation between their FAN IN/FAN OUT quantity squared and the percent of procedures with changes. The higher that value is for a given procedure, the more likely there are to be changes in that particular module. An important advantage of this measure is that other measures are applied to the program after they are written. This one is applied during the design stage. In terms of computing, this one is a little difficult to compute and I don't think the jury is in yet on it's usefulness. It seems appealing, but who knows?

Henry & Kafura's Information Flow

Define a complexity measure based on module connectivity and information flow.

DEFINITION OF COMPLEXITY MEASURE

For a procedure A, its FAN-IN is the local flows into A + number of data structures from which A retrieves information. Its FAN-OUT is the local flows out of A + number of data structures updated by A. There is a local flow of information from procedure A to procedure B if either of the following hold:

1. If A calls B.
2. If B calls A and A returns a value to B. which B subsequently utilizes, or
3. If C call both A and B passing an output value from A to B.

Complexity Measure = Length * (fan-in * fan-out)^2

WHAT DOES IT MEASURE?

Attempts to measure connectivity of procedure.

They feel that fan-in and fan-out component contributes most to complexity

HOW EASILY COMPUTED?

Moderately difficult.

Now I finally get to the last one. These are measures based on combinations of properties. One is McClure's complexity profile and the other is McTap complexity. I may not say much about the two because of the time. These measures are characterized by not being based on just one property, but a multitude of program properties. Let's look at the McClures complexity profile.

She defines a complexity profile as having ten parts to it.

1. Size of program (number of statements or instructions)
2. Number of modules in program
3. Number of variable in program
4. Number of global variables in a program
5. Average module size (in number of statements)
6. Average number of compares per module
7. Average number of modules accessing a global variable
8. List of common modules
9. List of modules that access more than the average number of global variables
10. List of modules that exceed the modules size limit of 50 statements or exceed the module compare limit of 10 compares

You can almost go to a shotgun kind of approach, plug in everything, and some of it should be useful. It's hard to think of something that's not in here. In one and three there's some of Halsteads in that, directing counts. Besides the data, the major thing to look at in this measure is the heavy emphasis on computing averages and then looking for modules that exceed the averages. Her theory is that if the module exceeds the averages, then that is a possible place where extra testing or maintenance effort may be required. So you are actually trying to indicate where there are potential problems.

For the McTap complexity, you select the set of program features and assign a relative weight to each. This turns out to be your feature vector which is what you work with. Then you select a reference vector for each of these features and compute a difference between the reference and feature vector. Now that you have a difference between those, you multiply each positive value in your difference vector by the weight and that will give you a score vector. Then you sum up your entries and your score vector and that's your complexity. The intent is to go after things that deviate from the average (reference vector). It's wide open as to what your program features are. He leaves it up to the person to use whatever features that they want. There's been some work done in COBAL for McTap like measures.

That's the survey, it's been quite quick. I've tried to give you a flavor of the different classes and what's been done.

III. CLOZE PROCEDURE AND EXPERIMENTATION

Let me talk to you about the CLOZE procedure and that will relate to the experimental materials I handed out. This is a slight digression into some work I've been doing at Oregon State University called the

CLOZE procedure. What we are attempting to measure is program understanding based on the premise that understanding is critical to testing and maintenance. If you can't understand the program how can you expect to test or maintain it. Some of the studies that have been done have looked at controlled experimentation using:

Modularization
Comments
Structured coding
Mnemonic variable names
Program length
Flowcharts
Control flow
Data flow

All these things impact understanding. Then, when you do an experiment, you like to do some sort of measurement to see if there is a difference. Some of the measurements they use are:

Time to locate a bug
Comprehension quiz
Ability to reproduce a functionally equivalent program
Time to perform a modification
Halstead's E (effort)
Speed of hand execution
Subjective responses

A lot of things like this attempt to measure performance. Of course, my claim is whether these measures of understanding are good measures. All of these have shortcomings or limitations. For example, when the subject is required to reproduce a functionally equivalent program, how does that ensure understanding? Isn't it more ability of memorization? Of all these measures, the comprehension quiz, if carefully done, is most commonly used and accepted. The danger with a comprehension quiz is that you may let your question give answers to other questions. You have to be careful when you design your comprehension quiz that you don't give away certain information.

What we propose is using the CLOZE procedure as a measure of understanding. What CLOZE refers to here is the human tendency to complete a familiar, but not quite finished pattern. Most of it is there, but there are a few pieces missing and you have to fill it in. When we measured using the CLOZE procedure, we would give the subject a program with some of the tokens omitted. The subject was required to fill in the missing tokens. Their score of course, was how many they filled in correctly.

We did an experiment where we compared the CLOZE procedure with the comprehension quiz.

	CS 212	CS 318	CS 415
Program 1			
Comprehension quiz			
Mean	32.0	48.0	57.8
Std dev	17.8	17.6	13.3
N	36	18	16
Cloze score			
Mean	53.0	67.5	71.4
Std dev	17.5	15.3	9.4
N	37	18	15
Program 2			
Comprehension quiz			
Mean	46.7	61.1	73.5
Std dev	19.8	14.9	12.6
N	37	16	17
Cloze score			
Mean	64.8	76.8	79.1
Std dev	17.6	9.8	10.6
N	37	17	15

Table 2. Means and standard deviations for each class, program version and method.

We took students from three computer science classes, CS 212 which was a prerequisite for CS 318 which was a prerequisite for CS 415. We thought that students would correspond to three different levels of experience. We had two shell sort PASCAL programs. We gave the students either a PASCAL listing of one program and comprehension quiz over that program, or we gave a student a description of the CLOZE procedure and then a CLOZE version of one program and they would have to fill in the answers. For Halstead's E, one of the programs had Halstead E's around 8,000. The other one had Halstead E's around 20,000, so there was quite a difference between the programs. We gave each subject one of the two programs and one of the two forms.

Let me just indicate the results. For the first program, it looked like the students in the later classes did better. The same was true for the second program. Standard deviations were quite decent. The CLOZE scores indicated the higher the class the better the performance. Just looking at the data, it looks like there's a pretty nice correlation between the CLOZE scores and comprehension scores.

The whole key here is, if this is shown to be a good measure then it is much easier to do a CLOZE procedure than to actually define a comprehension quiz and grade. Believe me, we had a terrible time constructing the quiz, and then the grading has to be done very very carefully.

The two programs that you have, hopefully you got both, one side look at the green one, the other side, look at the pink one.

The green one is an example of a program that has an error in it. Look at the output. The stars indicate that the answer was beyond the format specification, so we have an error.

```
      PROGRAM STAT(INPUT,OUTPUT,TAPE1=OUTPUT)
      INTEGER S,SCNT
      DIMENSION A(15),S(5),TOTAL(3),AVER(3),VMIN(3),VMAX(3)
      READ(*,100)  (A(I),I=1,15)
      READ(*,101)  (S(I),I=1,5)
  100 FORMAT(15F3.1)
  101 FORMAT(5I2)
      WRITE(1,101)(S(I),I=1,5)
      WRITE(1,102)(A(I),I=1,15)
  102 FORMAT(15F4.1)
      NV=3
      NO=5
      DO 1 K=1,NV
      TOTAL(K)=0.0
      AVER(K)=0.0
      VMIN(K)=-1.0E+10
    1 VMAX(K)= 1.0E10
      SCNT=0
      DO 7 J=1,NO
      IJ=J-NO
      IF(S(J) .EQ. 0) GO TO 7
      SCNT=SCNT+1
      DO 6 I=1,NV
      IJ=IJ+NO
      TOTAL(I)=TOTAL(I)+A(IJ)
      IF(A(IJ) .LT. VMIN(I))VMIN(I)=A(IJ)
      IF(A(IJ) .GT. VMAX(I))VMAX(I)=A(IJ)
    6 CONTINUE
    7 CONTINUE
      IF(SCNT .EQ. 0) GO TO 15
      DO 10 I=1,NV
   10 AVER(I)=TOTAL(I)/SCNT
   15 DO 20 I=1,3
   20 WRITE(1,103) TOTAL(I),AVER(I),VMIN(I),VMAX(I)
  103 FORMAT(4F6.1)
      STOP
      END
```

```
1 2 0 4 5
4.0 6.0 7.5 4.5 6.5 8.0 5.0 7.0 8.5 5.5 7.5 9.5 6.0 8.0 9.5
 21.0   5.3    .0******
 27.0   6.8   -.0******
 34.5   8.6   -.0******
```

The pink one is a CLOZE procedure. There are a few blanks to fill in.

```
      PROGRAM STAT(INPUT,OUTPUT,TAPE1=OUTPUT)
      INTEGER S,SCNT
      DIMENSION A(15),S(5),TOTAL(3),AVER(3),VMIN(3),VMAX(3)
      READ(*,100)  (A(I),I=1,15)
      READ(*,101)  (S(I),I=1,5)
  100 FORMAT(15F3.1)
  101 FORMAT(5I2)
      WRITE(1,101)(S(I),I=1,5)
      WRITE(1,102)(A(I),I=1,15)
  102 FORMAT(15F4.1)
      NV=3
      NO=5
      DO 1 K=1,NV
      TOTAL(K)=0.0
      AVER(K)=0.0
      VMIN(K)=1.0E10
    1 VMAX(K)=
      SCNT=0
      DO 7 J=1,
      IJ=J-NO
      IF(S(J) .EQ. 0) GO TO
      SCNT=SCNT+1
      DO 6 I=
      IJ=IJ+NO
      TOTAL(I)=TOTAL(I)+A(IJ)
      IF(A(IJ)          VMIN(I))VMIN(I)=A(IJ)
      IF(A(IJ) .GT. VMAX(I))VMAX(I)=A(IJ)
    6 CONTINUE
    7 CONTINUE
      IF(SCNT          0) GO TO 15
      DO 10 I=1,NV
   10 AVER(I)=TOTAL(I)/SCNT
   15 DO 20 I=1,3
   20 WRITE(1,103) TOTAL(I),AVER(I),VMIN(I),VMAX(I)
  103 FORMAT(4F6.1)
      STOP
      END
```

What the students would have received when they did this experiment was the following documentation.

DOCUMENTATION

Purpose of Program:

To calculate the total, average, minimum and maximum values of a set of observations.

Data:

NO is number of observations

NV is number of variables per observation

A(1),...,A(NO) first variable in all observations
A(NO+1),...,A(2*NO) second variable in all observations
A(2*NO+1),...,A(3*NO) third variable in all observations
etc.

S(1),...,S(NO) input vector indicating which observations are to be considered in the calculations. Only observations with a nonzero S(J) value will be considered.

TOTAL(1),...,TOTAL(NV) hold totals for each variable

AVER(1),...,AVER(NV) hold averages for each variable

VMAX(1),...,VMAX(NV) hold maximums for each variable

VMIN(1),...,VMIN(NV) hold minimums for each variable

INPUT DATA (NO=5,NV=3)

```
4.06.07.54.56.58.05.07.08.55.57.59.56.08.09.5
0102000405
```

CORRECT OUTPUT

VARIABLE	TOTAL	AVERAGE	VMIN	VMAX
1	21.0	5.3	4.0	6.5
2	27.0	6.8	5.0	8.5
3	34.5	8.6	7.5	9.5

Let's see if the people on this side can find the bug in the program and the people on the other side can fill in the blanks correctly, given this documentation. This is what the students had, the documentation, and either the green one or the pink one. We have indicated the correct output at the bottom of the documentation page. Your experience in this exercise will help you understand controlled experimentation.

Anyone want to venture a guess as to the error in the green one? The error in the green one is right here at the initialization for Vmax and Vmin. When we initialized Vmin to a very low number and Vmax to a very high number, we actually reversed the two. If we had reversed these, it would have worked correctly. The students were given the documentation and told there was one error in the program and they should correct it. The thing that surprised us most about this, was the fact that it turned out to be a very difficult problem. We did it in a junior level class. If you are presented with a program you have never seen before and not the greatest documentation and you're told there is a single error in the program, to find it is quite difficult.

For those of you who had the pink version, the CLOZE version, the first blank you had to fill in was the value for Vmax. The value you should have had for that is a large negative number. The second missing token was in the DO statements. The token NO was omitted, as the upper bound on the DO loop. The third blank was the label on the GO TO statements which should be 7. The statement DO 6 should be I=1, NV. The missing compare in the IF statement should be LT. and the last blank should be IF (SCNT EQ.0) GO TO to 15.

We were very selective in the tokens for the CLOZE version. There were actually three different versions of this program each with a different error, and we were trying to see whether one type of error was harder to find than another kind of error We made sure that our CLOZE version had all those error sources as tokens to be filled in. They had about twenty minutes, and that surprised us. The ones that were doing the CLOZE versions had no trouble doing it in the twenty minutes, but the other group finding the error had trouble getting it done in the twenty minutes. In fact, we had an experiement where we asked them to perform a modification on a program, and that one was a disaster, where at most one or two students were able to do that. What that told us was that performing a modification was a very difficult task. Certainly not one to be done in 20 minutes, so you learn as you do this process.

Let's talk a little bit more about controlled experimentation. What I want you to do is have a little bit of background on controlled experimentation. It would be a good time to ask, "what problems do you see in performing controlled experiments like this where you're trying to validate a measure?" For example, what problem, or problems, did you notice? Would it be possible to give one group the program with good documentation and another

group the program without any documentation? I think the answer is yes. They have done experiments on documentation with flow charting versus pseudo code, and have found that documentation is helpful.

Documentation is difficult to quantify as quality. The size has no meaning at all in terms of documentation. Essentially good quality documentation would make a task quite easy to do.

Any comments or questions you see regarding the validity of a controlled experiment? How do you know that they are trying? You have to be careful in an academic environment, you couldn't threaten to flunk them, what carrot could you use? Everything had to be anonymous. All you could do is ask them to do their best. Motivation is a problem. In some situations, what you do is pay them. That seems to provide the best motivation.

How does a simple program like this generalize to a real world environment? That is a big problem. What they've done is a realistic task, and the researchers proposing a specific measure say it doesn't matter if the program is larger or not, you still get the same results. There's a serious question as to whether that is a valid assumption. The problem with controlled experimentation is that it isn't always a similar situation to the real world.

I won't belabor this anymore, but let me mention three problems with controlled experiments highlighted in a paper by Brooks in the April 1980 issue of Communications.

The three general problems are choice of participants, choice of experimental material and choice of task to be performed on materials. In terms of choice of participants, what you want in terms of a class of participants, is a sample that is reflective of the general population. How would you select such a sample for your experiment? If you experiment at HP could you come up with 30 programmers who are representative of the general programming population? That's a problem because there is no uniform programmer. In the middle '60s, Sakoman did a study showing a 29 to 1 difference in programmer's productivity. Programmers are known for their non uniformity but yet you want them to be uniform so your experiment has validity. So those are two opposing views. Students, on the other hand, tend to be uniform, but are they representative?

If you look at most of these experiments, they were done on students, do they generalize for the real world? A solution partially, is to do "within subject" experimental design. What you do is make sure each participant does everything. So all the participants would do the same thing. In our experiment we would have each participant do both a CLOZE procedure and find a bug in a program. In other words, I wouldn't just have one group do one thing and the other group do something different, hoping that my ability level for each group is similar.

A second problem is choice of experimental material. How do you choose experimental material that is representative of the task? Is it realistic to have a program with one error in it? Does that correspond to what's out in the real world? Probably not. If you are going to try an experiment relating to maintenance, how do you select that material? You want it to vary according to what you are trying to run your experiment on. For example, if you're investigating structuredness, how does that impact maintenance? You would have your materials vary in how structured the program is. You need this variation in order to draw effective statistical conclusions. The problem is that you want the material to be short enough so they can do the experiment in a reasonable amount of time.

A third problem is the choice of the task to be performed. What is really a representative task? Is it realistic? I thought someone would say that it wasn't really realistic, if you are trying to see whether something impacts debugging if you have only one bug in the program. How many programs have only one error? How do you know ahead of time there is only one error? Sometimes with the task you tell them to perform, say for example you want them to do a modification task, you may have to specify that task in such detail that you give them a tremendous understanding of

the actual program. By specifying what they are supposed to do you actually give away the information you wanted them to discern from the program. Brooks really has a vendetta against timed experiments. For example, if you are doing an experiment on how much time to find a bug, or perform modifications, how do you know the subject spent all that time on that task? How do you discount daydreaming, reading instructions, etc.?

All I've tried to do in terms of controlled experiments, is make you aware of the tremendous problems in validating measures, and some of the problems with controlled experimentation. I'm not discounting what's been done, but I want you to see it's quite difficult.

IV. VALIDATION

How do you demonstrate that a software measure measures what it's supposed to? Well, you can collect empirical data. But, if you collect empirical data, there is a problem with how available and reliable the data. The computer field is notorious for poor reliability of data. A second approach is to go in and study actual performance. Have one team do a system a certain way, and then have another team do the same thing, using another technique. Then, try to judge the merits of the various techniques. But what company has the resources to do something like this? How can you really hold all things equal other than the one variable you're testing? Actual performance studying is quite difficult. One thing you can start doing is compare the results of a proposed metric with existing metrics. That is typically how a lot of the measures I've discussed today have been justified. They have compared the measure they're studying with others that are pseudo accepted and if theirs is pretty close, then it must be pretty decent. Another thing that I would like to see is controlled experimentation. That is another possible way to try to validate the measures. But all this discussion underscores is that validation is an extremely difficult problem.

V. CONCLUSIONS AND PERSONAL SUGGESTIONS

In conclusion I'll make the following points:

1. Measuring complexity is difficult.
2. No ONE single program property or single number characterizes software complexity. I think that has been underscored by all of the methods
3. Research in this area has produced some useful information and promising beginnings. We have a little bit of understanding of complexity, maybe what we see scares us, but we do have a better understanding.
4. There's pressure for a useful measure now. There are some available, but if you use them, be aware that they provide only an approximate measure.

Let me give you some of my personal suggestions. Since no one

single number characterizes complexity, I would go for a combination. My combination would go after both the micro and macro aspect. I think you have to separate them. For a micro measure you want to look at how complex an individual module is. Here is where you can use a McCabe measure or something like this. This would look at an individual module. You could use Halstead's, or something else. All these give you very local properties about a given module. The other thing you need to introduce into your measure is some overall measure of program comlexity. I call this the Macro. What is the module interconnection structure? The average number of interconnections between modules appears to be a significant number. Henry and Kafura seems useful here. You need something that will actually measure the overall interconnection structure. So, that's what I mean by looking at the macro and micro aspects. I think it is really unclear exactly what measures you should put under each.

BIOGRAPHY

Curtis R. Cook received his BA degree in mathematics from Augustana College in 1965 and his MS and PhD degrees in computer science from the University of Iowa in 1967 and 1970 respectively. He joined Oregon State University in 1970. He is a professor in the Computer Science Department. His research interests are graph theory applications in computer science, software complexity measures, formal languages and minimal perfect hashing functions.

DISCUSSIONS ON IEEE WORKING GROUP ON SOFTWARE RELIABILITY MEASUREMENT AT PACIFIC NW SOFTWARE QUALITY CONFERENCE

Ted Workman was originally scheduled to conduct a session on the working group and the standard being developed. When he was no longer available, Edward Presson was asked to fill in. Mr. Presson first gave a description of the background and history of the working group and its goals. This was followed by summary of the proposed contents of the Guidebook for Software Reliability Measurement and its current status. The audience was then asked to contribute their questions, comments, and suggestions for the guidebook, which are summarized below.

QUESTION: Will the guidebook tell me which measures are most valuable to me?

ANSWER: No, it will describe the inputs and outputs of the measurement technique, when it is applicable in the life cycle, the relative maturity of the technique, its theoretical background, and provide references for further reading. The guidebook will also have a general chapter on the reasons for measuring and predicting software reliability. It is up to the reader of the guidebook to determine which measures are most valuable in his/her environment.

QUESTION: Is the guidebook part of a overall plan to measure and rate software reliability that would lead to an "Underwriters Laboratory Seal of Approval" for a software product?

ANSWER: No, the science (or art) of software reliability measurement is not yet mature enough to permit such standards to be set. The working group views the guidebook as a valuable step in setting standards for measurement and discussion of software reliability. With a common vocabulary and measurement approach, we may accelerate the understanding necessary to permit the eventual development of such an industry standard. But this is not a formal goal of the working group.

SUGGESTION: Include a section on generic data gathering; that is, suggestions for software managers and engineers about what types of data to collect that may be of use in validation of existing or future measures.

RESPONSE: Sounds like a good idea. While we can't always predict what future measures may require as primitive data, we might be able to suggest types of data to collect during the software life cycle. I'll relay the suggestion to the working group.

SUGGESTION: If you really want to accelerate progress in software reliability measurement and prediction, why doesn't the working group set up a cooperative effort between interested researchers to exchange experience and data. Maybe a new working group would have to be formed.

RESPONSE: Interesting ideas. It is beyond the charter of the current working group. Perhaps the guidebook, when completed could serve as the basis for such a follow-on group. I'll pass the idea along.

SUGGESTION: I'd like to see a "tool box" section in the guidebook which would list all the techniques in terms of their inputs, outputs, and objectives. A sort of cross-reference of objectives to various reliability measurement and prediction techniques. Then I would not have to read through many descriptions of inappropriate techniques to find those I could use.

RESPONSE: Chapter 6 of the guidebook may serve this purpose. I can't really answer because the chapter organization has not been given to the working group for review. Certainly, the suggestion is a good one; I'll pass it on.

SUGGESTION: I notice the outline of the guidebook includes a section on human engineering for each measurement technique. I think that's good, since I'm especially interested in that area. Are you going to use the work being done by Sid Smith (in another IEEE group) in this subsection?

RESPONSE: I'll relay your concern.

QUESTION: Does the working group make its work available to other organizations. And how does the group know what other researchers and IEEE groups are doing?

ANSWER: One section of the working group is set up as an interface group to keep information flowing to and from other organizations and working groups.

QUESTIONS: You have said the working group draws a large group. Can you characterize the membership of the working group?

ANSWER: Ted Workman could have probably given you an accurate assessment. I can only take a guess: 20% academia, 50% industry, and 30% government (principally the National Bureau of Standards and the DoD). I'd also guess that at least 60% of the industry group are involved in defense contracts. Keep in mind that these are rough guesses, and may be off by 10 percentage points in either direction.

QUESTION: I notice that the working group is heavily attended by government and defense industry people. Will the guidebook be useful for those of use in the commercial industry?

RESPONSE: Yes, the working group is trying to avoid the terminology of any one group in the text of the guidebook. Both the language and methods should be applicable, in general, for all software developers and testers.

DRAFT GUIDE TO

SOFTWARE RELIABILITY MEASUREMENTS

SPONSOR

TECHNICAL COMMITTEE ON SOFTWARE ENGINEERING

OF THE

IEEE COMPUTER SOCIETY

PREPARED BY

SOFTWARE RELIABILITY MEASUREMENTS WORKING GROUP

OF THE

SOFTWARE ENGINEERING STANDARDS SUBCOMMITTEE

June 15, 1983

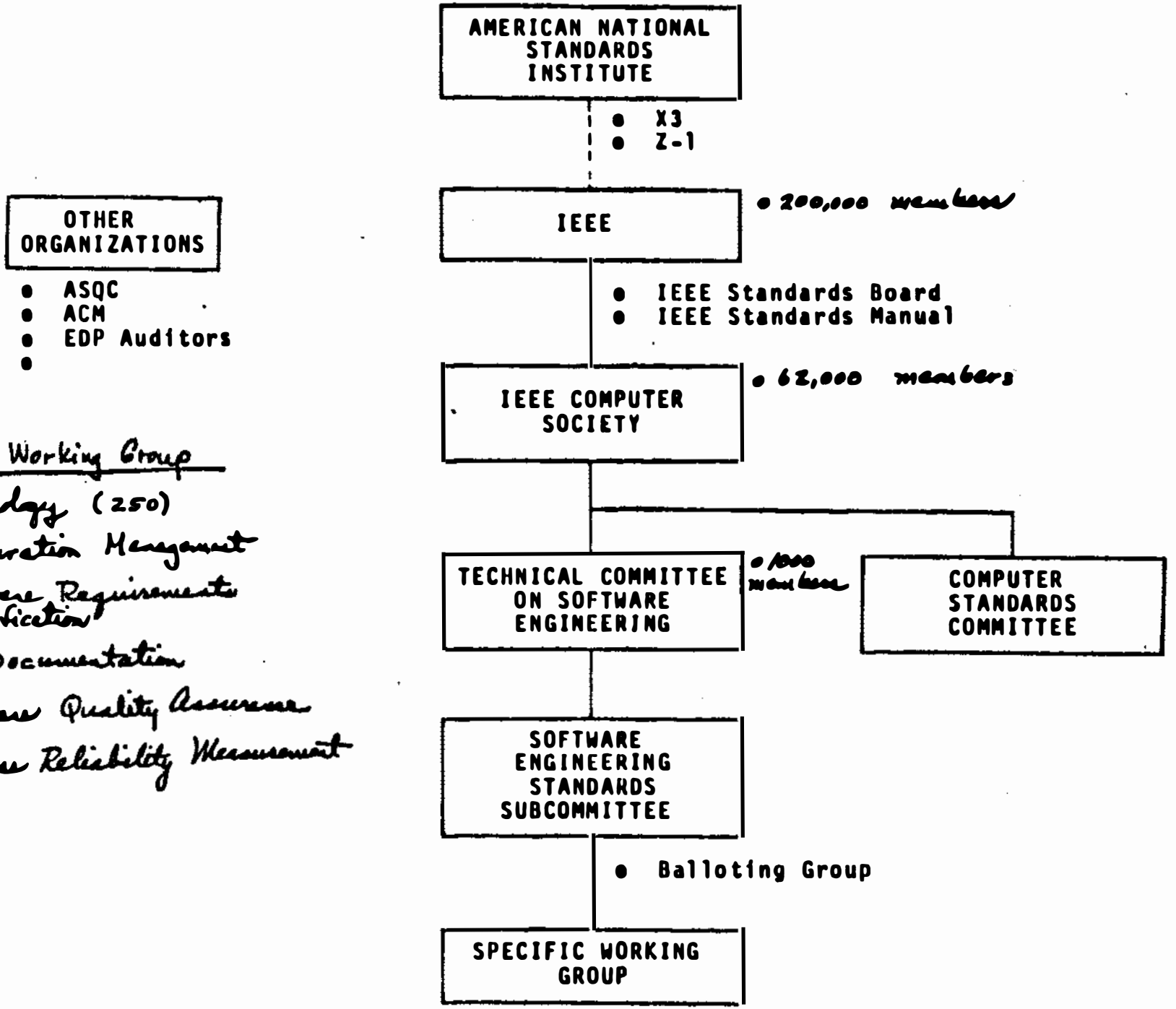
Guidebook Status

-Not yet assembled as a whole

1.	Scope	reasonable draft
2.	Definitions & Acronyms	reasonable draft
3.	S/W Reliability Measurement	1st draft
4.	Standards	1st draft not complete
5.	Application of Measures	28 complete 40 - 50 in preparation
6.	Measures	concept in preparation

ONE SEQUENCE OF EVENTS

1. WRITE SCOPE SECTION
2. DRAFT AN OUTLINE
3. FILL IN OUTLINE
4. REDO, REDO, REDO UNTIL REASONABLY STABILIZED
5. DETERMINE TRIAL USE/FULL-USE
6. ASSEMBLE BALLOTING GROUP
7. FINALIZE DOCUMENT
8. BALLOT
9. RESOLVE COMMENTS
10. SEND TO IEEE


AMERICAN NATIONAL STANDARDS INSTITUTE
• X3
• Z-1
IEEE
• 200,000 members
• IEEE Standards Board
• IEEE Standards Manual
IEEE COMPUTER SOCIETY
• 62,000 members
TECHNICAL COMMITTEE ON SOFTWARE ENGINEERING
• 1000 members
COMPUTER STANDARDS COMMITTEE
SOFTWARE ENGINEERING STANDARDS SUBCOMMITTEE
• Balloting Group
SPECIFIC WORKING GROUP
OTHER ORGANIZATIONS
• ASQC
• ACM
• EDP Auditors
Specific Working Group
• Terminology (250)
• Configuration Management
• Software Requirements Specification
• Test Documentation
• Software Quality Assurance
• Software Reliability Measurement
• ADA

P. Powell

June 15, 1983

Guide to Software Reliability Measurements Document Outline

Foreword
? Abstract
History
? Purpose
Audience
Committee Lists
? Relationship to other Committees

Table of Contents

1.0 Scope

2.0 Definitions and Acronyms

2.1 Definitions

2.2 Acronyms

3.0 Software Reliability Measurement

3.1 Introduction

ERROR, FAULT, FAILURE; PURPOSE OF SOFTWARE RELIABILITY MEASUREMENT PROGRAM; LIFE CYCLE

3.2 Features of Software Reliability Measurement

SOFTWARE RELIABILITY AS AN INTEGRAL PART OF LIFE CYCLE ACTIVITIES; CANDIDATES FOR MEASUREMENT; IMPORTANCE OF FRONT END INITIATION OF MEASUREMENT ON TO COMPLETION; QUANTIFIABLE AND SUBJECTIVE MEASUREMENT; LIFE CYCLE RELATIONSHIPS TO SW RELIABILITY

3.3 Fundamentals of Software Reliability Measurement Approach

MINIMIZATION OF FAILURE EVENTS; PURSUE "FAULT-AVOIDANCE EFFICIENCY"; CONTROL AND IMPROVE "FAULT-AVOIDANCE EFFICIENCY" LEVELS; ANALYSIS OF SOFTWARE RELIABILITY EVENTS (FAILURE, FAULT & ERROR ANALYSIS); MEASUREMENT OF OTHER PROGRAM ASPECTS

3.4 Requirements to Standardize Measurements

This section contains a discussion of a need to standardize, to the extent practical, the features, e.g. primitives, of software reliability measurement. This section will point to Section 4.

3.5 How to Use This Guide

WHAT IS IN REMAINING SECTIONS; IMPORTANCE OF RELATING MEASURES TO EACH OTHER AND TO THE DELIVERED PRODUCT; 5 STEPS PRIOR TO IMPLEMENTING A RELIABILITY PROGRAM

3.6 Summary

3.7 References

4.0 Standards for Software Reliability Measurement

taxonomy and terminology which is not in 729 and others; at this point, this section may be limited in content, but as the guide matures and before a standard is developed, this would serve as a convenient repository for standardization ideas

4.1 Introduction

4.2 Candidates for Standardization

E.G. failure, fault, error

4.3 Software Reliability Measurement Classification

4.4 Measurement Primitives

E.G. failure triplet, complexity primitives

Antonio's Schema of Information

5.0 Matrix and Application of Measures (practical section)

5.1 Introduction

A life cycle oriented measurement plan

life cycle: concept (no measures if not applicable); requirements; design; implementation; test; installation and check out; operations and maintenance; retirement

(This section will contain a discussion of relationships between measures during different life cycle stages and present the matrix of life cycle vs measure)

? Discussion of when metric category could be used, e.g. complexity could be used when maintainability is a critical factor - this is a possible area for standardization.

The reader is referred to Chapter 6 for basic abstract definitions.

5.2 Concept Phase

possibly no measures in which case this section will be deleted

5.3 Requirements Phase

(includes architecture)

5.3.x MEASURE, E.G. CYCLOMATIC

5.3.x.1 PRIMITIVES USED; REFERRED TO CHAPTER 6 FOR DEFINITIONS

5.3.x.2 APPLICATION - HOW A USER APPLIES METRIC, E.G. GENERATING TEST CASES

5.3.x.3 IMPLEMENTATION - HOW THE METRIC IS GENERATED, E.G. HOW CYCLOMATIC COMPLEXITY IS COMPUTED

5.3.x.4 INTERPRETATION OF RESULT, E.G. A LOW NUMBER NEEDS TO BE BALANCED VS INCREASING SYSTEM COMPLEXITY

5.3.x.5 CONSIDERATIONS

- RESTRICTIONS, E.G. LANGUAGE DEPENDENT, STRUCTURED CODE
- LEVELS OF KNOWLEDGE FOR UNDERSTANDING
- EASE OR DIFFICULTY OF APPLICATION
- EASE OR DIFFICULTY TO IMPLEMENT

5.3.x.6 TRAINING REQUIRED

5.3.x.7 EXAMPLE - POINT TO REFERENCES IF LONG

5.3.x.8 BENEFITS - GOOD FEATURES, E.G. TEST ADEQUACY

5.3.x.9 EXPERIENCE - IS THERE DATA CONCERNING ACTUAL USE OF THIS MEASURE, E.G. IS THE EVIDENCE OF USEFULNESS EMPIRICAL OR ACTUAL

5.3.x.10 REFERENCES

? RATIONALE FOR INCLUDING MEASURE

? BACKGROUND FOR INCLUDING SPECIFIC MEASURE

5.4 Design Phase

Same subsections as the requirements phase.

5.5 Implementation Phase

Same subsections as the requirements phase.

5.6 Test

Same subsections as the requirements phase.

5.7 Installation and Check-out Phase

Same subsections as the requirements phase.

5.8 Operations and Maintenance Phase

Same subsections as the requirements phase.
(includes conversion)

5.9 Retirement

Same subsections as the requirements phase.

6.0 Measures

This section is abstract in nature; classification of measures according to distinct concepts, approaches, and applications; measures having same characteristics are grouped together

6.1 Introduction

6.2 Complexity

6.2.1 Definition

6.2.2 Explanation

6.2.3 Derivations (Software)

6.2.4 Primitives

6.3 Reliability - Growth

(same basic subsections as 6.2)

6.4 Performance Measurement and Evaluation (PME)

(same basic subsections as 6.2)

6.5 Human Engineering

(same basic subsections as 6.2)

6.6 Omnibus Approach

(same basic subsections as 6.2)

6.7 Software Dynamics

(same basic subsections as 6.2)

6.8 Software Experiments

(same basic subsections as 6.2)

7.0 References

This section may not be necessary if references are included in each section.

PLANNING AND SOFTWARE QUALITY

Ramune Arlauskas
Firmware Manager
Tektronix

The text of this paper was transcribed from videotape. It has been edited for clarity in written form but purposely left in a distinctly conversational tone. Slides and pictures have been included where appropriate.

INTRODUCTION

I am going to be talking about planning and software quality. There are a lot of different ways to measure software quality. I think pretty much in all software production companys the bottom line is user satisfaction and that definitely is the case at Tektronix. I feel that software quality can greatly be improved and enhanced through adequate planning, not only up front, but through the entire development process of a software product. We used planning extensively on the development of the 4105 firmware. The 4105 is one of seven products that Tektronix introduced in April. Since I'm at HP and with 2 Intel people on the panel I decided I should first share with you a little bit about what these products are and then talk in a little bit more detail about the planning process we used in the firmware development.

PRODUCT BACKGROUND

These terminals that I will be talking about are a series of three known as the 410x series of terminals or as they are known in house, the unicorn line. As I said, three of them were introduced in April. We are currently manufacturing and delivering one and the other two are completing the engineering phase. They consist of three terminals the 4105, 4107,& 4109. We also introduced 2 companion products to these terminals. A low cost ink jet printer and then also the 4170 which is a local processing unit.

The 4105 is the first in this line of low cost, color, raster graphics terminals and is the beginning of coupling together color graphics and color alpha/numerics. The 4105 has a 13" display. It has a very high quality 60 Hz non interlace tube with 460 x 380 resolution. It has eight colors in the alphanumeric phase and 8 in the graphic. That means that there is a total visibility of 16 different colors out of a pallet of 64. We have a very wide assortment of graphic features in the product that are Plot 10 compatible. The graphics include 8 different types of line styles, polygon definition in fill, over 140 predefined patterns, and we also have scaleable and rotateable text. The alpha/numerics reside in a dialogue area and these are color alpha/numerics. The dialogue area can be adjustable. You can have it anywhere from 2 lines to 30 lines and it's scrollable in a virtual buffer. Also, there are 80 columns and you could scroll through 132 columns. We're ANSI 3.64 compatible and also DEC VT100 compatible so with these terminals you can use the wide gamut of full screen editors currently available for ANSI 3.64. One of the very nice features of the product is you can view both the dialogue area and the graphics area at

the same time. The background of the dialogue area is made transparent so you can see the graphics area underneath.

The 4107 is the big brother to the 4105. The resolution of the graphics is higher and the product contains extensions in the graphics area. You can do local segments and local pan and zoom.

The 4109 has essentially the same features as the 4107 except with a 19" display. There is also a pallet of 4,000 colors on the 4109.

Reliability and service is the name of the game on these products. It was our goal throughout both the hardware and firmware design and implementation. This enabled us to issue a 1 year warranty on the system. Above that, you can also purchase an optional 2 year warranty on the system. This includes the hardware and the software. So you can see why reliability and quality had to be a key goal. It was one of the driving forces on this project.

THE FIRMWARE PLANNING PROCESS

OK, I've talked about the product. I'll next talk about the firmware planning process that we used in the development of the firmware. It's referred to as iterative planning, in that, typically planning is done up front in a project. If you're real careful about your planning you'll generate volumes of material. Then very often it's forgotten about and the plans are shoved in a drawer somewhere and not looked at too much throughout the implementation of a product. On the unicorn development we used planning pretty much consistently throughout the design and implementation of the product. We feel that greatly contributed to the high quality we're experiencing now. Also there were some very nice effects on the engineering climate. I'll also talk about the results that are being realized as far as the software quality goes.

This was a project that had a lot of very aggressive goals in it. We had to do it right the first time due to the aggressive cost, schedule and performance goals. We really didn't have the option of doing it over again if it didn't work. The way we chose to minimize the risk of things going wrong in the firmware was by having extensive planning meetings to clearly define what we wanted to do before we decided how we would do it and defining how we would do it before we actually implemented it.

We developed a structured environment in the firmware development. By this I don't mean that we were doing top down design or using so and so's technique on implementation. What I mean is that we knew what we were doing and we had processes in place when they were needed. We weren't dissipating any of our energies by waiting for things to happen to let us proceed with our job. Some of the things that contributed to the structured environment were:

- Good solid specs
- Adequate control
- High visibility
- Good communication
- Well partitioned, manageable

Good solid specs were a goal from the beginning of the project. We spent a lot of time in the external specification phase and came up with a solid set of specs before we went into the internal design. Ideally we would have liked to freeze the specs but I think, as everybody who has worked on a software project knows, as the project matures you understand your user better, you really understand what the project is all about a little bit better, so you're always making changes. What you want to do is minimize the size and scope of these changes. You don't want to make a change in the implementation phase that requires significant redesign work. On the unicorn development we chose to make the firmware changes difficult. We had a change control procedure that gave a lot of visibility to any of the changes and made it a difficult process to go through.

We also used adequate control. I use the word adequate because we used both tight and loose control in the firmware development. There were areas that we used very tight control on and other areas that we were looser on the control. One of the places where we used loose control is that we were hoping to be able to buy a change control configuration system for our source control. It turned out that it wasn't available when we needed it. We decided that since the decomposition of our system had a number of subsystems that individual engineers had responsibility for, that individual engineer would worry about the source control for his subsystem. The subsystems that had more engineers working on them would come up with their own way of handling source control.

We also maintained high visibility of what we were doing. We maintained high visibility not only of what the product was, but what the product wasn't. We had regular reviews with upper management and marketing letting them know exactly what the product wasn't. We had a number of goals put on us such as being 4100 compatible and VT100 compatible. We kept that up front the whole time making sure that everybody understood where we were with the compatibility issues.

We also maintained good communications, not only within the group among the engineers, but also strove to maintain good communications with the various support groups we were dealing with.

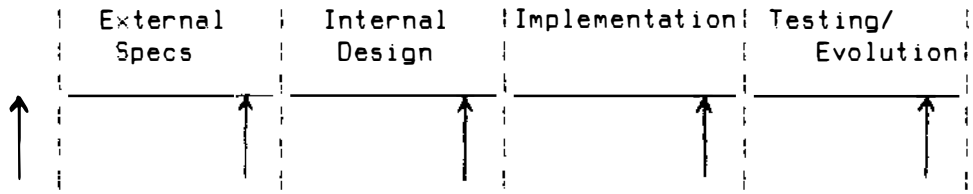
We worked towards having a very well managed and partitionable system. We had a person that acted as a chief architect and another person who was a project leader. These people were concerned mainly about the interfaces between the various subsystems that the other individual engineers had responsibility for.

The way we came up with this structured environment was through planning. As I mentioned, we knew what it was that we were doing before we had the detail of how we were going to do it, and we had the how defined before we actually went into the implementation. This culminated in very high productivity. I think that high productivity definitely adds to software quality. If you've got a group of people who want to do a good job and are doing a good job you can't help but end up with a very solid product.

We used planning at different levels in our design and implementation. The planning that we did was outside of our regular status meetings. Our status meetings were used for tracking how you were proceeding against your schedule. These meetings were for a special purpose, geared just for the planning effort.

UNICORN FIRMWARE ITERATIVE PLANNING

All day pre phase meetings
- Establish plan
- Verify and expand for next phase
- Establish or detail processes for upcoming phase
- Set up controls
- Give assumptions visibility
- Delineate responsibilities



The first type of meeting we had were all day prephase meetings. These went all day or whatever part of a day was necessary. We had these before entering the various phases of development. As you can see, we use a very text book type of approach for our development. It's very straight forward. An external phase, an internal design phase, the implementation and the testing phase. The arrows indicate when we would have these prephase planning meetings.

We were also new as a group. The firmware team had not worked with each other before so we didn't know what engineering techniques everybody was used to or what their beliefs were or anything else. To minimize that adjustment period we had an initial all day meeting to establish a plan for how we were going to define and develop this firmware. I have a sample of the index to our plan.

I. Scope of project plan
II. Project description
III. Project management plan
IV. Resource management plan
V. Development plan
VI. Development/documentation methodology
VII. Configuration control plan
VIII. Integration and test plan
IX. Organization plan
X. Schedule
XI. Assumptions, risks, contingencies
XII. List of firmware documents

It's a very typical type of project plan. Some of the more interesting areas were in configuration control. For example, we didn't always know how we were going to do something, but we knew we wanted to place certain items under configuration control and had them listed in here with dates for when the processes had to be in place so that we could move on cleanly to the next phases and have the processes there available for us. The organizational plan was another important area, in that we had listed not only what our firmware organization was and who had responsibility for what, but we also had a number of very complicated dependencies on parallel developments in the company on other products. We had those dependencies listed out in this section, a responsible person associated with it, what we thought our agreement was, what we thought we were getting from that group or what we thought we were supplying that group and when we thought it was due. Another area that was important was the assumptions and risks. These were assumptions and risks from the firmware end, that is, what we saw the risks being and what contingencies we had. We talked about and defined all these areas in our first planning meeting, so really as a group we were setting out how we were going to implement and define this product. We wrote up the plan and then distributed it not only to the firmware engineers in the group but also to upper management and to the various support groups and various people listed in the organization plan. They then review that and either agree to "Yes, that is how they understood they were working with us" or "No, we made some wrong assumptions somewhere."

With subsequent planning meetings we iterated on this plan and expanded on it to make sure that everything was in place when we needed it. We also set all the controls as a group in the planning meeting. That gave us immediate buy/in by everybody on how we were going to handle control. We were minimizing the amount of time it took to convince every engineer that "Yes, this is how we are going to handle our control system" We gave our assumptions plenty of visibility in these planning meetings. If anybody questioned what the goals were or why we were doing something or why we weren't doing something else, this gave a formal channel to discuss it as an engineering group and get us back on focus. We frequently reviewed our responsibilities. When you make your initial schedule it's pretty difficult to have a good assessment of what the implementation takes. So we reviewed the responsibilities and redistributed responsibilities wherever necessary to make sure that any one person wasn't being overloaded.

An outcome of this very first meeting was that we were beginning to function as a group. For us, that was very important since we had never worked together as a group. We were a group of engineers with very diverse skills and were beginning to pull our knowledge base together even before we got into the external specification phase. We were already beginning to function as a productive unit.

The next type of planning meetings we had were intermediate check points which were established during the prephase planning meetings. They were sort of like minimilestones to see if our scheduling and our development were progressing as we hoped it would. We identified within every single phase many minimilestones. We tracked those very carefully and had specific planning meetings to make sure we were

meeting those various milestones. One minimilestone that we had in the external specification phase was to release a preliminary external specification. We felt that we needed to give that a lot of visibility. Our typical customer was a Tektronix engineer. We wanted to get the information out and get their input back as soon as possible so that we could hit this external specification milestone cleanly. In the implementation phase another minimilestone we had was a release as early as possible. That was very important to us because we had no prior experience with the processor language or tools we were using for development. We made a lot of assumptions up to this point so we had to make sure as early as possible that we were still on the right track. By this time we had spent nearly 2/3 of our firmware engineering time. We set up these check points and then tracked them very closely with individual meetings, making those minimilestones almost as important as the end of the phase milestones.

Another type of planning meeting that we had was when we started getting close to the actual milestone. There are a lot of little things that fall through the cracks and a lot of things you don't realize until you're close to the end of a phase. So we would begin these meetings early enough so we could make any of the necessary changes to enable us to hit the milestone. One of our goals was to enter the next phase as cleanly as possible.

The iterative planning I'm referring to is planning done outside the regular status meetings. It's done throughout the duration of the project all the way through the evaluation phase. You might think of it as driving a car. You get in your car and decide how to get somewhere. Then when you get to a road block you make a li ttle alteration but you still end up at your destination. That's the effect of the planning. We made all the little changes and corrections necessary to assure we met all our project objectives. The outcome of the iterative planning was that the engineering climate was established and reinforced with that initial meeting. We had talked about how the project was to be run, what methodologies would be used and what each one of the engineers thought was important on the project. We were establishing the base and beginning to function as a group at that first planning meeting. The iterative planning also helped us to define before the design and this was defining before the design throughout the entire development. We defined what processes we needed before we actually defined the process. We had the external specs before we had the internal design. We defined what implementation methodology we would use before we actually had to go in and use it.

There was a continual reinforcement of goals. If we were beginning to shift our focus or question our goals this provided a means to keep us on track. All the controls were set by the group. There was the immediate buy-in and immediate acceptance of all controls. Another thing that we ended up doing with the incremental milestones is that we were really planning for incremental succcess. We realize that now in retrospect. As a new group it was important for us to gain credibility that we attain what it is that we are planning on doing. The credibility is not only with upper management, but also with the various support groups and with the company as a whole. By setting up and making these minimilestones people began to think that this is a

TEST COVERAGE ANALYZER

Bill Gervasi
R&D Software Engineer
Intel Corporation

The text of this paper has been transcribed from videotape. It has been edited for clarity in written form, but purposely left in a distinctly conversational tone. Slides and pictures have been included where appropriate.

At Intel we put together a colloquium, a collection of software people from the major software groups in the company. It was specifically composed of software developers, evaluation engineers, diagnostic engineers and test engineers from the development systems group. All these people had been running off pretty much at random developing tools. We kind of loosely threw them together. At one point about two years ago we decided to try pooling resources. We got together, and laid out our specifications for a family of life cycle support tools. So the tool that I'm going to discuss today, the test coverage analyzer, fell out of the software colloquium. What we realized is that we needed something tangible. We could theorize all we wanted about software tools, but we really needed to have something that would help us do our job, and leverage the quality of our software. Well, leveraging the quality of our software included leveraging the quality of our software test cases. So we defined the Test Coverage Analyzer.

The basic assumption of the test coverage analyzer is that if we come up with very good software test cases, we will implicitly produce higher quality software. What we wanted to do then was not replace our current test procedures, but to augment them. The test coverage analyzer augments our white or black box testing by insuring that every line of code is executed at least once, or we'd better have a reason why it wasn't executed. Traditionally, engineers went through a very manual procedure for determining that their software really worked. You used a manual debugger to stop a piece of code at a point in execution, examine the state of the machine manually and verify that it worked as you intended. That was error prone simply because it was manual and therefore sometimes not reproducible. It often fell between the cracks because you don't write down everything you do as much as we'd like to think so.

The test coverage analyzer then wants to approach the problem of how you keep your test cases consistent and how you measure how well your test cases work, measuring test case effectiveness. One important facet was that we use a lot of programming languages at Intel, and we wanted this thing to be language independent. It needed to execute in something near real time, otherwise it would be too intrusive on the development engineer or evaluation engineer's time. It had to be automated to some degree. It also had to allow interactive modes and options to direct the coverage,and it was a part of a family of lifecycle support tools. Because it was developed by multiple groups everyone had their inputs into what it needed to be. The language was

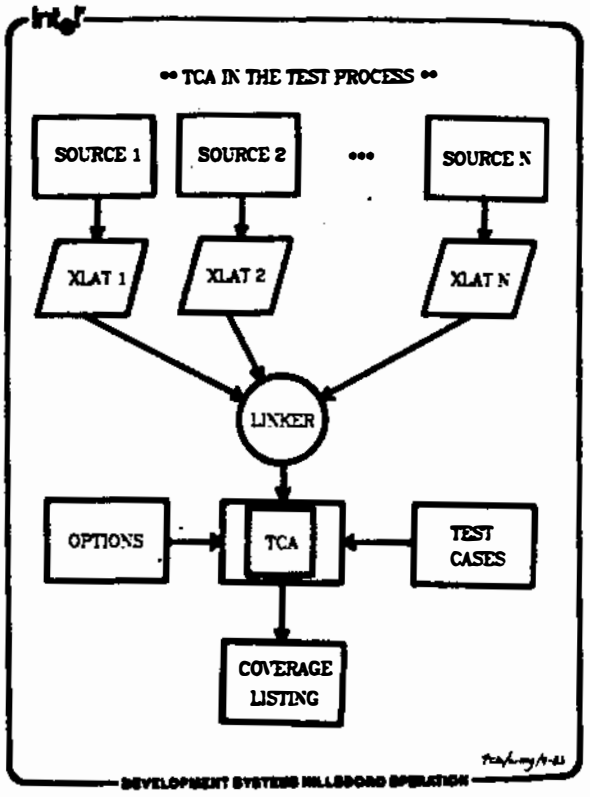
one issue where we bantered around whether PLM or PASCAL or assembly language was the way to go. Well, we decided to support them all. What helped us there was Intel's object module format which produces debugging information compatible with what TCA requires. What we needed out of this thing was line numbers. The TCA wants to know what line numbers were executed, therefore it needs to know the absolute addresses of where line numbers are when it is placed in memory. It will then break on each line number, log that that line number was executed and then continue processing. An interesting thing about the implementation of TCA is that it removes the break point at that line number, so the next time it's executed it runs at real time. It's only the first time that a line is executed that you take the performance hit of having test coverage analyzer in the background.

Automation was very important. That was driven mostly by the evaluation group. They wanted to use this not just in testing software but in long term regression tests. We could come up with a set of regression tests to run on a piece of software and then log that away so that if a change is made in the software we can run this set of regression tests again and we should see similar or improved results from the previous run through.

We wanted to be able to break the process at any time and examine how the test coverage was doing. And indeed TCA allows that. You can break processing, interrogate how well your coverage has worked so far, and then continue processing where you left off, re-entering the program. In that process a design engineer can beat on a piece of his code, try a test case, and see if that got the desired result by breaking out, looking at his coverage, and reentering. He can keep trying more test cases until he gets one that does what he thinks it ought to do.

Obviously if you were going to put break points at every line number through the whole program you would have unacceptable overhead. Particularly if you want to test one module that's not tested until the last quarter of your regression tests. You would not want to run a 12 hour regression test to see if you fixed a problem. One of the modes then of the TCA is to be able to limit the break points to a certain module or to procedures only. One output of TCA is what procedures have and have not been executed.

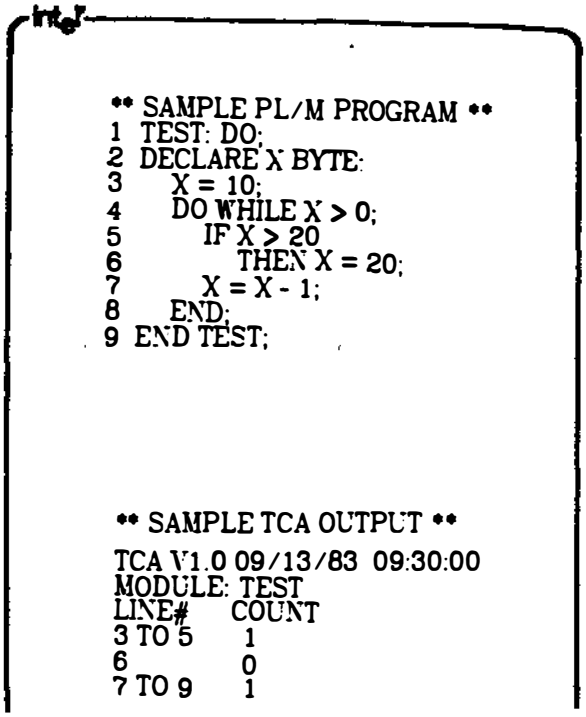
The environment for TCA is oriented at the business that we do the most of, ICE products. ICE products that run on an 8086 in a series three development system. The limitation then that TCA puts on us, is that we have to have the 8086 in this environment and we have to have lots of memory. Basically that's it. All of the TCA and all of the tables must load at the same time and reside in memory at the same time but the program that you are testing may have overlays. It does support multiple tables for all possible overlays in your code.



So where does TCA fit into your design process? You create your source program using the editor of your choice. Any language that Intel supports with a translator is OK. The output of the translator is the 8086 object module format that is loaded by the linker into one object module with all references between programs resolved. The TCA is invoked with run time options. For example, the specification of a certain module or of procedures only. It loads in the object module produced by the linker. You run your software test cases on the TCA and when TCA is done it produces a coverage listing as it exits to the operating system. It is done in three steps then. It loads your test program, sets up the tables, begins execution of your code, and then possibly the interactive modes, and then when you exit to the operating system it intercepts that and utilizes that time in which to produce your coverage listing.

I've put together a sample program that should give us a better flavor for what TCA does. This a PLM program that takes a variable and runs it down from 10. We have one nonsense statement at line 5 that says if x is greater than 20 then we should do something. We should never see that line execute. This is a code loop which would allow me to show how the execution speeds up and then we have a sample TCA output here. One thing that TCA must recognize is that lines 1 and 2 have no executable code. That's recognized in the object module format of the 86 OMF. It starts it's analysis at line 3-5. The way this program will work then, is TCA loads that program, puts in the break points at all the executable lines. It hits line three. Well, line three should

be executable. It breaks, logs that fact, and then goes back and executes that line. When it hits line 8 it will loop back to line 4 but because line 4 is already executed, line 4 executes the second time at real time. So although timing dependent loops may take that performance hit and not execute correctly the first time you run that part of the test, the next time you come back through you can hit it at real time. Specifically, if this, for example, were a condition that were testing some obscure time dependent relationship, that would then execute perhaps the next time you came through this loop. The output, then says that tests 3-5 were run , test 6 was no and 7-9 were run. I think it's rather obvious that TCA is already set up for the next revision of TCA which is to be able to do counts on how often each line was executed, and then do performance analysis on where you are spending too much time in your code. The next version of this will do total counts and then percentage analysis.



```
** SAMPLE PL/M PROGRAM **
1 TEST: DO;
2 DECLARE X BYTE;
3    X = 10;
4    DO WHILE X > 0;
5       IF X > 20
6          THEN X = 20;
7       X = X - 1;
8    END;
9 END TEST;
```

```
** SAMPLE TCA OUTPUT **
TCA V1.0 09/13/83  09:30:00
MODULE: TEST
LINE#     COUNT
3 TO 5      1
6           0
7 TO 9      1
```

In summary then, the test coverage analyzer is set up to help us measure our test effectiveness, assuming that improving our software test cases will also reflect in an improvement in our software. It is language independent as long as the language translator produces the correct debug information. It executes with a performance hit initially, but as your software starts ascending or descending your levels of your code, it approaches real time execution so in reality you can do effectively very little additional overhead to a regression test. It is automated and allows your software to be tested in an automated regression test, but it also has interactive modes and options that allow you to trim exactly what you're looking for. And it is part of a family of tools that was produced by the software colloquium. The most important thing we learned at Intel out of this was that the software colloquium could have quite a bit of clout. We were a bunch of software engineers that got together and pinpointed some needs and put together a plan and Intel management has given us the resources to put some of these tools in place.

SOFTWARE QUALITY

FITNESS FOR RESUSE

Ned Thanhouser
R&D Software Manager
Intel Corporation

The text of this paper was transcribed from videotape. It has been edited for clarity in written form but purposely left in a distinctly conversational tone. Slides and pictures have been included where appropriate.

I appreciate my friends at Tektronix helping me get information together for this presentation. I spent nearly one third of my time there on this project. Juran's statement "fitness for use" triggered thoughts in my mind because what we ended up doing was not so much designing for fitness for use, but fitness for re-use. That is, the recycling of software. I'm going to talk a little bit about some generic definitions of what makes software reuseable and relate that to a project that I was on where some firmware was actually reused. I'll show some measurements that I used to see that it was reused and show some of the benefits to Tek.

When we talk about "fitness for use" and the quality of software, everybody can come up with their own list. So I want to recognize the expert Tony Hoare. In 1972 he produced a list of what he calls "The Quality of Software" Here are 17 very good one line definitions.

THE QUALITY OF SOFTWARE*
FITNESS OF USE

1. **CLEAR DEFINITION OF PURPOSE**
2. **SIMPLICITY OF USE**
3. **RUGGEDNESS**
4. **EARLY AVAILABILITY**
5. **RELIABILITY**
6. **EXTENSIBILITY AND IMPROVABILITY IN LIGHT OF EXPERIENCE**
7. **ADAPTABILITY AND EASY EXTENSION TO DIFFERENT CONFIGURATIONS**
8. **SUITABILITY TO EACH INDIVIDUAL CONFIGURATION OF THE RANGE**
9. **BREVITY**
10. **EFFICIENCY (SPEED)**
11. **OPERATING EASE**
12. **ADAPTABILITY TO WIDE RANGE OF APPLICATIONS**
13. **COHERENCE AND CONSISTENCY WITH OTHER PROGRAMS**
14. **MINIMUM COST TO DEVELOP**
15. **CONFORMITY TO NATIONAL AND INTERNATIONAL STNDARDS**
16. **EARLY AND VALID SALES DOCUMENTATION**
17. **CLEAR ACCURATE AND PRECISE USER'S DOCUMENTATION**

C.A.R. HOARE, SOFTWARE - PRACTICE AND EXPERIENCE, VOL. 2, 103-106 (1972)

These are all extremely important attributes of software. But I don't think you'll find any modern software today that meets all of these objectives fully. The important point is that when you talk about quality and fitness for use, and you have a list of 17 things that you want to do, you're going to run into some conflicts trying to meet your objectives. It's important to recognize that there will be conflicts. It's also important to understand which of your customer requirements are the most important attributes. Then, when you make those compromises you're not doing it based on a wish list. You're doing it from a position that's more realistic than just wishful thinking. So you have to make these decisions about where you are going to put your energy with reasoned background.

Now, we will go away from fitness for use and look at the flip side, fitness for reuse. The subjects I want to cover are the key factors affecting the significant re-use of software. I'd like to relate those to a real live project experience, one that I was on, the 4100 series. It is now five years after we started that project. We'll be able to give you some bottom line results.

But now a little side track..... Who knows what is made at an Aluminum plant?..... Money!!!!

That's the bottom line of why it's in business even though it makes many other things. We have to keep this in perspective when we see the engineering task before us. Whenever you have a project, you have three basic things you can manage which are cost, schedule, and performance or features. All of those involve decisions and trade offs which affect the quality of the software you deliver. From that, why is reuse important? Well, we're all here to make money, and every line of code like every beer can costs dollars and cents to develop. Like the aluminum plant, we're here to make money. The primary control factor we have in software is to limit the number of new lines of code we choose not to develop. Recycling code like recycling beer cans reduces the energy required to obtain end results. So the premise I'm presenting to you is that recycling as much code as possible will give you an opportunity to improve on the quality that was there originally as well as reduce the energy required to produce it. It's a very simple conservation of energy.

KEY FACTORS WHICH AFFECTS REUSE

REUSE MUST BE ONE OF THE FIRST AND KEY PROJECT GOALS

DEVELOPMENT METHODOLOGY MUST BE CONSISTENT WITH REUSE GOALS

SYSTEM MODEL MUST BE DEVELOPED WHICH REFLECTS REUSE OF HIGH-LEVEL COMPONENTS

MACHINE DEPENDENT FEATURES MUST BE ISOLATED TO MODULES WITHIN HIGH-LEVEL COMPONENTS

REUSE IMPLIES A PORTABLE IMPLEMENTATION

HIGH-LEVEL DEVELOPMENT TOOLS MUST ALSO BE REUSABLE

DOCUMENTATION MUST CLEARLY REFLECT WHAT IS AND IS NOT REUSABLE

STAFFING WITH PRESONNEL WHO BELIEVE IN THE VALUE OF REUSE

You should keep these eight key factors in mind when engineering software. There's nothing revolutionary about the factors, but it's important to keep them in mind when you consider the aspects of recycling your code. The first one is that reuse must be a basic principle and objective of your project. From the very beginning you must say "we are going to reuse code" if you plan on this approach. And then you must have a methodology which supports reuse of code, that is, the methodology you choose must not hinder or exclude the development of reuseability.

Your system model is the model that all of your engineers agree on for communication between components of the system. It should be developed to reflect reuse at a high level of components. That is, you should have high level components that can be reused, picked up, transported, and deposited and reused. When you have machine dependencies, these dependencies must be isolated to modules within these components such that if you want to transport machine dependent components within that component, you have modularized the machine dependent features.

That implies a portable implementation. Now everyone says, "oh, there's another vote for high level language". Well, I vote for high level language except when three things are violated. The language won't let you accomplish the task you have to get done, because it gets in the way, or the language produces code which is intolerably slow for the application you're in or the output for the compiler is so large that it won't fit into the code space that you've allocated.

Now we started out the 4100 series project saying "hey,PLM is going to be 80% of the whole thing." It turned out that of the reuseable code none of it was PLM. We ended up going through a crash project to make it fit by converting the PLM to assembler. I think Roger Crystal got down to converting a thousand bytes a week from PLM to assembler. It became a real grind but you have these little ROMS that only take so much and if you overflow into your neighbor's space he gets overflowed. So we violated those rules and went to all assembly language. You'll

see later on, that the process or environment provided the portability as opposed to the language.

High level development tools. Not only must the implementation be picked up and moved, but the tools that you use for implementation or reimplementation and the techniques and training you've invested in your folks also must be portable. Documentation must clearly reflect what is and is not going to be reuseable. This will help the folks on the other end of the project who are going to pick it up and run with it be successful. Although last on the list, but probably more important than the first on the list, is you need to have the people who not only have the skills to implement code in a reuseable fashion but you must have the ethic that reuse is important. It is just like sorting out beer cans in the grocery store. It may be a little bit more plain, but the long term benefit can be great. You need to have people who believe in it as an ethic.

What I would like to do now, after covering the 8 particular things that make code reuseable is relate this to the project I worked on when I was at Tektronix. This project was formally begun with a product proposal in 1978. There had been a lot of work before this, but official approval was given in 1978. Two products were introduced in June, 1981, the 4112 & 4114. Then in October of that year, the 4113 was introduced. We had the simultaneous development of two products for June introduction and then the following product for October. So from the beginning you will see that the 4100 series firmware had reuse as one of its central goals. Probably the fact that we had to develop two allowed it to occur. Then I'm going to go through a brief description of the firmware in terms of what it is and give you some statistics on it to give you a feel for the size of the project, the cost and the investment that was made by the company. Then I'm going to match some of the results and key factors that we used and compare these and then give you the bottom line. A significant amount of the code originally developed for the 4100 series was able to be recycled. Here's some basic statistics.

THE 4100 FIRMWARE

SUPPORTED TWO SEPARATE PRODUCTS 4112 AND 4114

RUNS ON INTEL 8086 MICROPROCESSOR

TOTAL OF 167.5 K BYTES OF NEW CODE

TOTAL OF 163 NEW COMMANDS

TOTAL OF 1,152 MODULES

TOTAL OF 14 SUB-SYSTEMS

TOTAL OF 52 EROMS (32 KBITS EACH)

EXTERNAL USER DOCUMENTATION OVER 1500 PAGES

INTERNAL SYSTEM DOCUMENTATION OVER 980 PAGES (40% COMPLETED)

OVER 41 PERSON-YEARS OF ENGINEERING

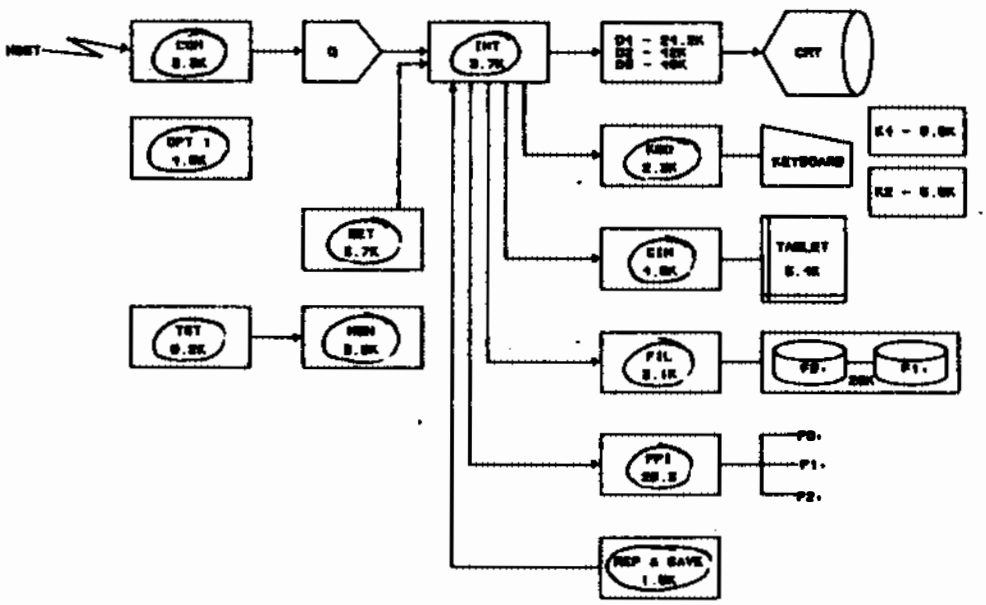
OVER 8.5 PERSON YEARS OF EVALUATION AND TESTING

SIMPLY PUT, A LARGE INVESTMENT BY TEKTRONIX

If you own a calculator you can do all sorts of wondrous things with these figures, but they are potentially useless, so watch out. To deliver one byte of code was $13.39. The coding rate was about 2 bytes an hour. That translates into about 80 bytes per week. I think the Ken Willit method was 100 bytes a week or 50 bytes a week if you were using a high level language. Those were within the range of reality. These statistics, while they seem real low or real expensive depending on what side you're looking at them, were taken from the approval to work on the project until it is shipped out the door and everything works. This is a period of over 2 1/2 years of engineering. So you can do some more funny things with your calculator. The average size of a module was 145 bytes. They are nice, reasonably sized modules, but the cost of a module was $2,000 . Every time we decided to add another module, you could say, well, you threw another $2,000 at the project. With 163 commands, that was $14,000 per command investment by Tektronix, so if you say, well, I think I'll throw another command in, that was another $14,000 worth of energy you were throwing away. Every 32Kbit ROM had $44,000 worth of engineering time stuck into it. This did not include capital equipment. This was just expense items. The bottom line that I'm trying to get across here is that this was a huge investment by Tektronix. Huge. And if you reuse or recycle this code, you can lever off of that investment. That's what this is all about.

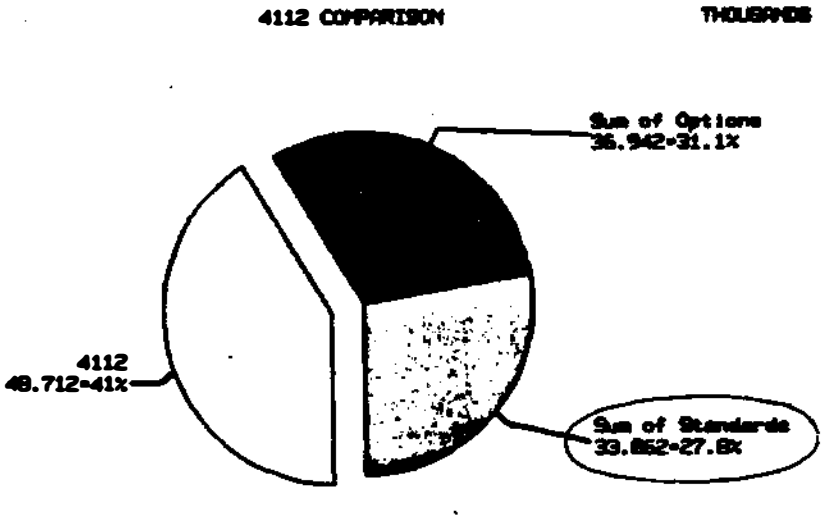
So let me share with you some of the methodologies and system architectures that we used that I believe led to being able to reuse the code. This was our highest level system architecture design.

4100 SERIES - FIRMWARE ARCHITECTURE



We basically had a data flow from left to right across the diagram. The number in each box is the size of the code. The host communications came in to a data communications subsystem to handle interrupts, went into a que to buffer up processing on the side and then there was a command interpreter which was really the heart of the system, that would dispatch off command tables and say "oh, this is yours," and "this is yours" ..... Here is the display handler. You'll notice that there is a display for the 4112, 4113, 4114, and these are the sizes of the modules and those handle all the display specific components. And then we see keyboard handlers, graphic input, optional tablet, file system with a disc handler. This one happened to end up in the high level language of PLM because we 32 Kbytes of code space on the particular board that would hold those ROMS so there was plenty of room. The guy went crazy and wrote 26K. 3ppi was pretty much, this was a peripheral port interface with 3RS232 ports. This is a report and save mechanism. The monitor was basically a multitasking set up and then self test code which would go around and touch and feel all the modules and the hardware to make sure it was working well and then report some status back to you.

Now by design, the stuff circled was targeted for reuse.



4112 UNIQUE AND COMMON CODE

If you were to take a look at the bottom line of what was reuseable in a pie graph, here is the standard shipped with every box (code that was common). As a matter of fact it was the same part numbered ROMS on the 4112 and the 4114. The same part could go in either socket on either machine. This code of 33K or 27% of the 4112 was common. The black segment of the pie represents options that were shipped if you ordered a tablet or a disc or something. The white segment of the pie represents the display unique code. That gives you a flavor for the code size.

4100 FIRMWARE DEVELOPMENT METHODOLOGY

TOP DOWN INCREMENTAL DEVELOPMENT

PHASED RELEASES WITH INCREASING FUNCTIONALITY

HIERARCHICAL DECOMPOSITION BY SUB-SYSTEM

PROJECT LEADER/CHIEF ARCHITECT

PROGRAM LIBRARIAN/CLERK

EXTERNAL "BLACK BOX" EVALUATION

CONFIGURATION CONTROL SYTEMS (FORMALIZED LATE IN PROJECT)

Now, some methodology that we used. We used a top down incremental development technique. We had a document called Helium, which was a high level document that was basically an external specification that, over time, was frozen, thawed and frozen again. But the design resulting from it was one of incremental design. That is, we've got this feature to put in, called segments, we implement the highest level of that and then work down. Then we find out that oops, that doesn't work, so there was some incremental iterative work.

Then we used phased releases where we had laid out five hunks of major command sets that were to be released to our evaluation group. The first hunk was all the software that would make it act like an ASR33 teletype. That is, you could get a character in the back door and see it print on the phoshorous of the screen. That was a major accomplishment, because all the frame work was set up. All the interconnections between the major systems were there and the command interpreter was there. We learned a lot from it . It was the most performance critical aspect of the project. The fact that it was released first and then would be released a second, third, fourth and fifth time means it would have the most wear and tear from the evaluation group. This also meant it would be the stablest part of the code. And it turned out to be. Every release had more functions to it, and we had 5 releases. We used hierarchical decomposition by subsystem. You saw the first layer of the decomposition on the system architecture diagram. Each one of the boxes on the diagram was owned by an engineer as the project leader. The project leader owned the box and just like in Ramune's project they owned a source code control for that and attempt to keep up with it. We had a chief architect who was your monster back type in football that would go around taking care of problems and own the architecture and would coordinate.

We used a program librarian and a clerk to do distribution, generation, documentation generation, handle our request for change forms from the evaluation group and it was very useful by unloading some of the burden off the engineers.

And again we used the external black box evaluation. When we shipped this new release of firmware off to the evaluation group they would shoot holes in it from a users prospective and submit error reports and we would respond to those.

The configuration control system was formalized late in the project. The individual engineers were all keeping track of their own source code modules. And finally, we had to institute "everybody turns in source, no object" because we didn't know what we'd get sometimes. And again one of the tools that Ramune was talking about was that when you have 17 people working on the same project it's hard to have everybody using the same sort of styles. So we ended up having to just turn in source code. But that was done late in the project because the tools were not available. That slowed things down but gave a little more quality in terms of our configuration control.

Let's get back to results. The 4112 firmware was used by plan in the 4105 as some of the base elements for that implementation. 24KBytes were recycled from the 4112. You know there was that one segment of the pie that was 33K Bytes. A portion of that were recycled. This represented 30% of the new 4105 product firmware which is 80KBytes Ramune tells me. Included in that recycling were the operating system, communicatons systems, the setup system (which was interactive, I want to change the Baud rate type of thing) and the command interpreter system. If you take your calculator and use all the constants from the previous slide, you can ask yourself, now, how much did this all cost in the original development? This 24KBytes that was carried over represents an original investment by Tektronix of some $329,000 if you use the numbers of that day. That, in terms of the 2Bytes per hour from start to finish, that's some 12,000 hours of engineering effort which represents six people years of effort in that 24K Bytes. Now, in all fairness, 24KBytes didn't just pick up, walk over and jump on the system and start running. There was a lot of work that needed to be done to get that running. But the key message is that not only the architecture which suported this environment, but also the bulk of the code was already there and ready to be used primarily because the people had the desire and the ethic to do it. So, if we look back at the project in terms of the factors contributing to this reuse, it is that reuse was a goal of both the original 4100 series. That is, we knew we had two products coming out the door simultaneously, so we had to structure for that. We also knew we had projects that were going to come out in the future that were going to have to use this. It was a key goal of both. The architecture that we had, as you saw, isolated to a very high level the main components that we could forecast that could change over time. The display systems, keyboard, graphic input devices, and so on.

The portable implementation ended up being provided by Intel. The 4105 uses the 186 which is a super set instruction set of 8086 (there are 10 more instructions on the 186). The fact that it wasn't written in a high level language didn't hinder this particular environment. The code was portable because of the [cpu] environment. The Intel development tools were reused. Intel bought a bunch of MDS systems, nine of them when I was there. Those particular tools were able to be used as well as the training on some of the tools, people had used before. Not everybody was familiar with the tools so there was some learning. I

have to say that the thing that made it happen were the people in Ramune's group who took this code and said "Hey, we believe this is a good goal and we're going to go with it" and they did. And that's the key results to look at.

It can happen, it did happen. The planning we did in 1978 allowed for the significant reuse of large amount of firmware in 1982. It wasn't easy. The area of making trades between we gotta get it out the door and we didn't finish the documentation, those kind of trade offs have to be made. I think the bottom line results speak for themselves. At the end of Tony Hoare's article on the quality of software he makes some very interesting statements that are very applicable to the reuse of software for getting quality results. "Don't decide exactly what you're going to do until you know how you're going to do it."is one of his statements. You have to know the method in which youre going to do it, and that certainly is true here. You can't decide how you're going to reuse all this firmware until you know how you're going to implement it.

The second point he makes is do not decide how to do it until you've evaluated your plan against all of the desired criteria for quality, which say, that if reuse is one of your high criteria, then your methodology, your ethics, your methods and the people that you hire have to value that. It can be done and it can be successful.

METRICS FOR EVALUATING THE QUALITY OF SOFTWARE

Thomas Bowen
Software Research Engineer
Boeing Aerospace Company

The text of this paper was transcribed from videotape. It has been edited for clarity in written form but purposely left in a distinctly conversational tone. Slides and pictures have been included where appropriate.

The topic I'm talking on is METRICS FOR EVALUATING THE QUALITY OF SOFTWARE . This is largely theoretical and will be kind of an introduction to the next two speakers, Edward Presson and Bill Moore. Here's an outline of what I'll be talking about.

- **BACKGROUND**
- **QUALITY MEASUREMENT CONCEPT**
- **QUALITY METRICS FRAMEWORK**
  - **Software Quality Model**
  - **Quality Factors**
  - **Quality Criteria**
  - **Quality Metrics**
  - **Factor Ratings**
- **LIFE-CYCLE RELATIONSHIPS**
- **FEATURES AND LIMITATIONS**

There was some early work done through RADC contracts by a fellow named McCall at GE. How many of you are familiar with that work? Not very many. OK, Rome Air Development Center in Rome, New York, is located at Griffiths Air Force base. Since 1976 they have let 5 different contracts in the area of quality metrics.

- **Contract F30602-82-C-0137 (RADC)**
  Specification of Software Quality Attributes
  Interim Reports D182-11310-1, D182-11378-1

- **Contract F30602-80-C-0330 (RADC)**
  Quality Metrics for Distributed Systems
  Final Report D182-11377-1, -2, -3

- **Contract F30602-80-C-0265 (RADC)**
  Software Interoperability and Reusability
  Final Report D182-11340-1, -2

- **Contract F30602-78-C-0216 (RADC)**
  Software Quality Metrics Enhancements
  Final Report RADC-TR-80-109

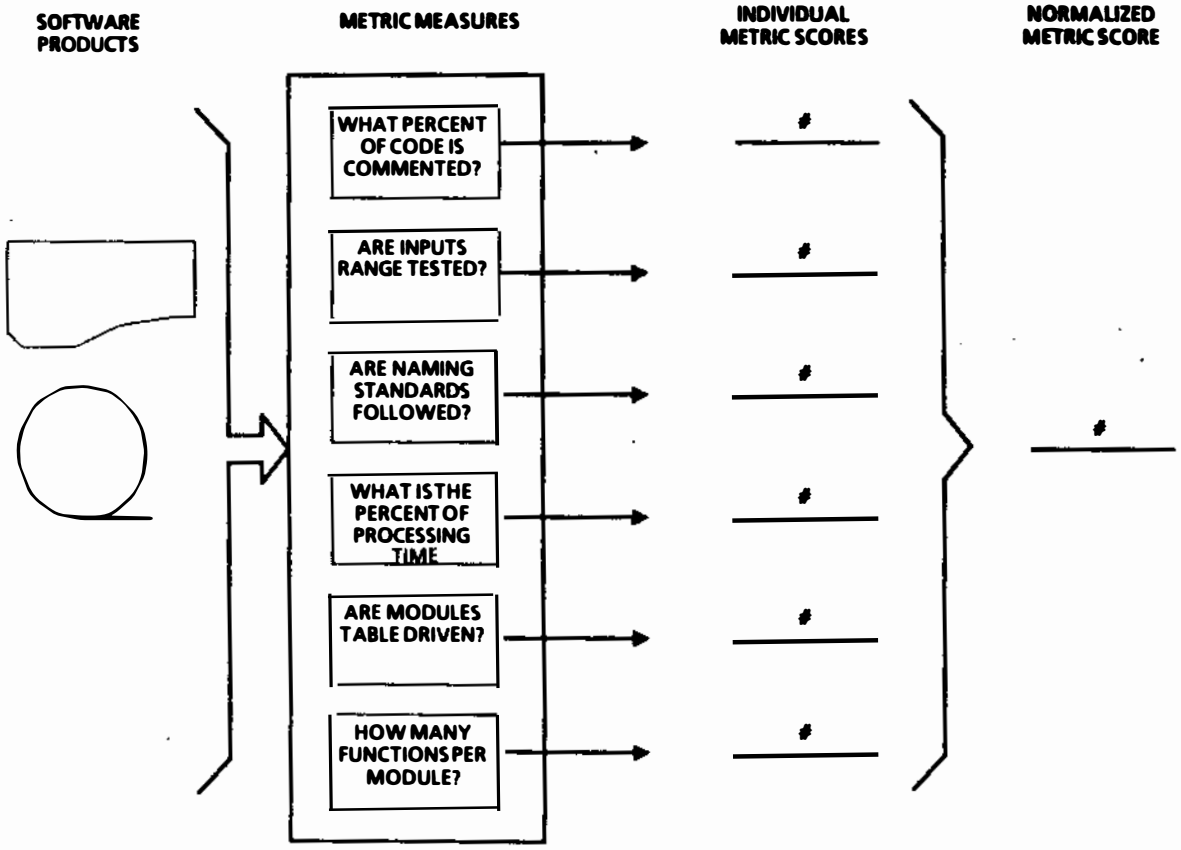
- **Contract F30602-76-C-0417 (RADC)**
  Factors in Software Quality
  Final Report RADC-TR-77-369

The most well known work is the first two (Specification of Software Quality Attributes and Quality Metrics for Distributed Systems) They were done by McCall and others at GE when he was at GE. The other 3 contracts (Software Interoperability and Reuseability, Software Quality Metrics Components, and Factors in Software Quality) have been done by Boeing. These next 2 have been completed by Boeing and the reports are out although they haven't been published by RADC yet. The final contract is the one we're working on right now.The information in this presentation is as though we were about halfway through the first Boeing contract.

The early contracts concentrated on developing metrics, trying to validate metrics, trying to come up with a methodology of use for the metrics and then enhancing all this (coming up with new factors, more metrics and trying to validate). This last contract is trying to tie all these things together. It's trying to polish it up and get it ready for use in the field by Air Force software contractors who will be required to use the metrics.

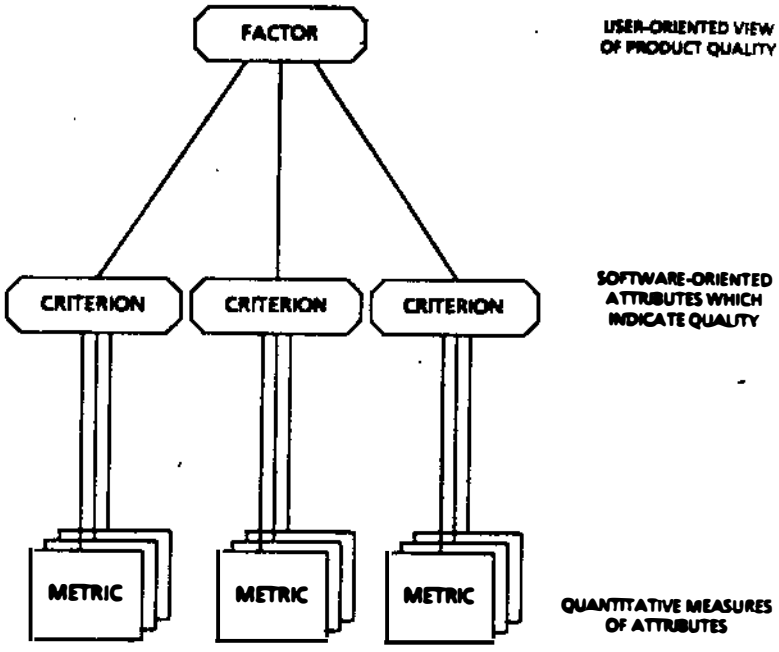
The main thrust of the metrics themselves and the factors is toward controlling defense contract type measures. I think they are equally applicable to commercial applications. This is what I mean when I talk about using metrics to measure software quality. When a person starts out with a software product or representation of the product and a document description specification or a source listing the person may want to know, for example, "If I use this particular product for my application, how reliable is this product going to be?" Or, "How maintainable is the product going to be?" or "How reuseable is it going to be?" To answer those questions, what we do is apply a set of metric measurements to those different products.

SOFTWARE QUALITY MEASUREMENT CONCEPT



Metric measures would be like:"What percent of the code is commented?" The answer to that question would be an input to a formula resulting in a quantitative metric score for that particular entry. Another example might be "Are input ranges tested?" That would give a yes/no answer. In that case the yes answer would be assigned a value of one and the no answer would be assigned a value of zero. After the scores are obtained for the individual metrics, these scores, are summed so a score can be obtained for a quality factor, like reliability or maintainability, or reuseability. So we can actually say we have a reliability of .95 or .98 or maintainability of .8 or a reuseability of .6.

This is the basic model that has been used for quite a while in measuring software quality.

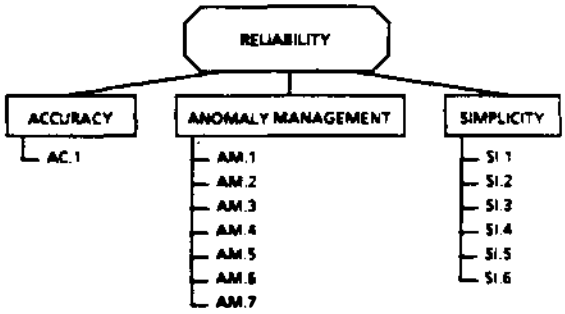


At the top level are quality factors. These are user oriented views of the quality of the software product. Examples would be maintainability, reliability, and reuseability. At the next level down in hierarchy are criteria. These are software oriented attributes which indicate the presence of quality. At the lower level is metrics. These are quantitative measures of the attributes. Each measure in turn is defined by a set of metric elements. Currently within the frame work that I am presenting today, which reflects where we are in this contract, there are 13 quality factors, 28 criteria, 74 metrics and 318 metric elements. A lot of these are new metrics, new metric elements, and several new quality factors. This compares to the McCall work, presented in 1980.

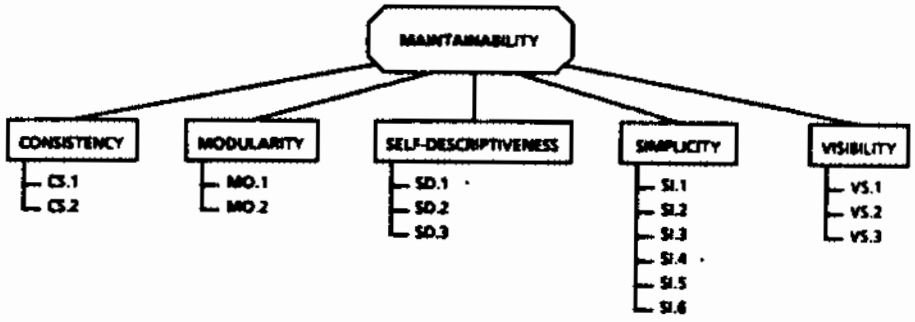
These are the thirteen quality factors.

Software Concern	User Concern	Quality Factor
PERFORMANCE* - HOW WELL DOES IT FUNCTION?	HOW WELL DOES IT UTILIZE A RESOURCE?	EFFICIENCY
	HOW SECURE IT?	INTEGRITY
	WHAT CONFIDENCE CAN BE PLACED IN WHAT IT DOES?	RELIABILITY
	HOW WELL WILL IT PERFORM UNDER ADVERSE CONDITIONS?	SURVIVABILITY
	HOW EASY IS IT TO USE?	USABILITY
INTERNAL DESIGN* - HOW VALID IS THE DESIGN?	HOW WELL DOES IT CONFORM TO THE REQUIREMENTS?*	CORRECTNESS
	HOW EASY IS IT TO REPAIR?	MAINTAINABILTY
	HOW EASY IS IT TO VERIFY ITS PERFORMANCE?	VERIFIABILITY
NEW USE* - HOW ADAPTABLE IS IT?	HOW EASY IS IT TO EXPAND OR UPGRADE ITS CAPABILITY OR PERFORMANCE?	EXPANDABILITY
	HOW EASY IS IT TO CHANGE?	FLEXIBIILTY
	HOW EASY IT IS IT TO INTERFACE WITH ANOTHER SYSTEM?	INTEROPERABILITY
	HOW EASY IS IT TO TRANSPORT?	PORTABILITY
	HOW EASY IS IT TO CONVERT FOR USE IN ANOTHER APPLICATION?	REUSABILITY

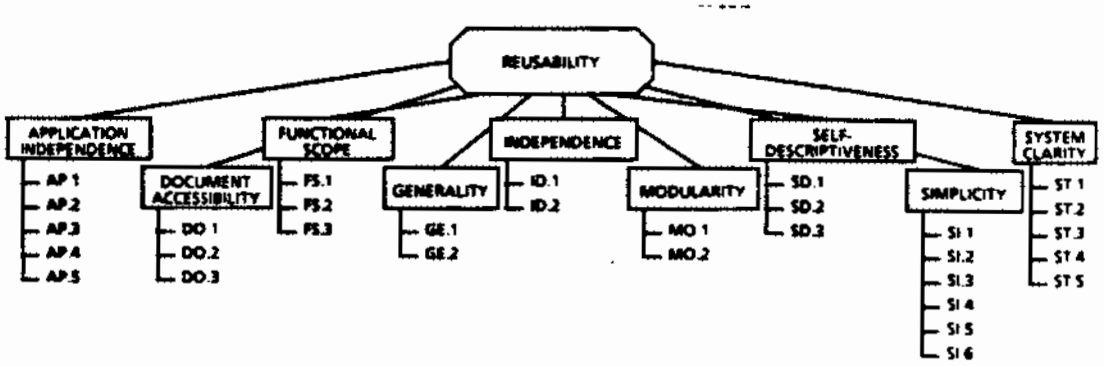
It really slices up what we mean by quality. The chart also shows what the user concern would be and their interest in this particular quality factor. The organization of quality factors is a little bit different than when McCall did his and there are several advantages to that. I simpified the frame work considerably. I've highlighted three of the quality factors here. Some of these next charts are kind of detailed, and I'd like to talk just about each of these quality factors as we go through the charts. This is the way an hierarchical model looks for the quality factor for reliability.



There are three criteria including accuracy, anamoly management and simplicity. Under accuracy, there's one metric called AC.1. Under anamoly management there's seven metrics and under simplicity there's six metrics. Each of these metrics has a name. We'll see what they are later on, it's just too much to put on the chart. And of course, it looks different for every factor, each different factor has different criteria and different metrics. This is what maintainability looks like. Five different criteria and associated metrics.



Reuseability has nine different criteria, and the associated metrics.



Each of the factors, criteria and metrics are very well defined. In the next few charts I'll show you definitions for factors, criteria, and metrics.

Table 3.1-2 Software Quality Factor Definitions

S/W Concern	Quality Factor	Definition
PERFORMANCE	EFFICIENCY	EXTENT TO WHICH RESOURCES ARE UTILIZED (STORAGE, PROCESSING TIME, COMMUNICATION TIME).
	INTEGRITY	EXTENT TO WHICH THE SOFTWARE WILL PERFORM WITHOUT FAILURES DUE TO UNAUTHORIZED ACCESS TO THE CODE OR DATA WITHIN A SPECIFIED TIME PERIOD*.
	RELIABILITY	EXTENT TO WHICH THE SOFTWARE WILL PERFORM WITHOUT ANY FAILURES WITHIN A SPECIFIED TIME PERIOD*.
	SURVIVABILITY	EXTENT TO WHICH THE SOFTWARE WILL PERFORM AND SUPPORT CRITICAL FUNCTIONS WITHOUT FAILURE WITHIN A SPECIFIED TIME PERIOD WHEN A PORTION OF THE SYSTEM IS INOPERABLE*.
	USABILITY	RELATIVE EFFORT FOR TRAINING AND SOFTWARE OPERATION-FAMIARIZATION, INPUT PREPARATION, EXECUTION, OUTPUT INTERPRETATION.
INTERNAL DESIGN	CORRECTNESS	EXTENT TO WHICH THE SOFTWARE CONFORMS TO ITS SPECIFICATIONS AND STANDARDS *
	MAINTAINABILITY	EASE OF EFFORT FOR LOCATING AND FIXING A SOFTWARE FAILURE WITHIN A SPECIFIED TIME PERIOD*.
	VERIFIABILITY	RELATIVE EFFORT TO VERIFY THE SPECIFIED SOFTWARE OPERATION AND PERFORMANCE.
NEW USE	EXPANDABILITY	RELATIVE EFFORT TO INCREASE THE SOFTWARE CAPABILITY OR PERFORMANCE BY ENHANCING CURRENT FUNCTIONS OR BY ADDING NEW FUNCTIONS OR DATA.
	FLEXIBILITY	EASE OF EFFORT FOR CHANGING THE SOFTWARE MISSIONS, FUNCTIONS, OR DATA TO SATISFY OTHER REQUIREMENTS*.
	INTEROPERABILITY	RELATIVE EFFORT TO COUPLE THE SOFTWARE OF ONE SYSTEM TO THE SOFTWARE OF ANOTHER SYSTEM.
	PORTABILITY	RELATIVE EFFORT TO TRANSPORT THE SOFTWARE FOR USE IN ANOTHER ENVIRONMENT (HARDWARE CONFIGURATION, SOFTWARE SYSTEM ENVIRONMENT).
	REUSABILITY	RELATIVE EFFORT TO CONVERT A SOFTWARE COMPONENT FOR USE IN ANOTHER APPLICATION.

* NEW OR MODIFIED

These are the definitions for each of the different quality factors. Some of these charts are pretty busy. The main thing to show you is that they are well defined. You kind of get a feel for what is at the lower levels. These are three of the criteria that are associated with the three factors we just looked at.

The three criteria are anamoly management, modularity and simplicity. And these are the definitions for each. I mentioned before, one of the advantages of new reorganization of quality factors, was that it simplfied things. One of the simplified things is that criteria falls out into three main categories, the same as the quality factors do. Performance, internal design, and new use. It just simplified the use of this frame work.

Table 3.2-1 Quality Criteria Definitions

SOFTWARE CONCERN	CRITERION	ACRONYM	DEFINITION
PERFORMANCE	ACCURACY	AC	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE THE REQUIRED PRECISION IN CALCULATIONS AND OUTPUTS
	ANOMALY MANAGEMENT	AM	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE FOR CONTINUITY OF OPERATIONS UNDER AND RECOVERY FROM NON-NOMINAL CONDITIONS
	AUTONOMY	AU	•THOSE CHARACTERISTICS OF SOFTWARE WHICH DETERMINE ITS NON-DEPENDENCY ON INTERFACES AND FUNCTIONS
	COMMUNICATIVENESS	CM	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE USEFUL INPUTS AND OUTPUTS WHICH CAN BE ASSIMILATED
	DISTRIBUTEDNESS	DI	•THOSE CHARACTERISTICS OF SOFTWARE WHICH DETERMINE THE DEGREE TO WHICH SOFTWARE FUNCTIONS ARE GEOGRAPHICALLY OR LOGICALLY SEPARATED WITHIN THE SYSTEM
	EFFECTIVENESS	EF	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE FOR MINIMUM UTILIZATION OF RESOURCES IN PERFORMING FUNCTIONS
	OPERABILITY	OP	•THOSE CHARACTERISTICS OF SOFTWARE WHICH DETERMINE OPERATIONS AND PROCEDURES CONCERNED WITH OPERATION OF SOFTWARE
	RECONFIGURABILITY	RE	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE FOR CONTINUITY OF SYSTEM OPERATION WHEN ONE OR MORE PROCESSORS, STORAGE UNITS, OR COMMUNICATION LINKS FAILS
	SYSTEM ACCESSIBILITY	SS	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE FOR CONTROL AND AUDIT OF ACCESS TO THE SOFTWARE AND DATA
	TRAINING	TN	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE TRANSITION FROM CURRENT OPERATION AND PROVIDE INITIAL FAMILIARIZATION
INTERNAL DESIGN	COMPLETENESS	CP	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE FULL IMPLEMENTATION OF THE FUNCTIONS REQUIRED
	CONSISTENCY	CS	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE FOR UNIFORM DESIGN AND IMPLEMENTATION TECHNIQUES AND NOTATION
	TRACEABILITY	TC	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE A THREAD OF ORIGIN FROM THE IMPLEMENTATION TO THE REQUIREMENTS WITH RESPECT TO THE SPECIFIED DEVELOPMENT ENVELOPE AND OPERATIONAL ENVIRONMENT
	VISIBILITY	VS	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE STATUS MONITORING OF THE DEVELOPMENT AND OPERATION
NEW USE	APPLICATION INDEPENDENCE	AP	•THOSE CHARACTERISTICS OF SOFTWARE WHICH DETERMINE ITS NON-DEPENDENCY ON DATABASE SYSTEM, MICROCODE, COMPUTER ARCHITECTURE AND ALGORITHMS
	AUGMENTABILITY	AT	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE FOR EXPANSION OF CAPABILITY FOR FUNCTIONS AND DATA
	COMMONALITY	CL	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE FOR THE USE OF INTERFACE STANDARDS FOR PROTOCOLS, ROUTINES, AND DATA REPRESENTATIONS
	DOCUMENT ACCESSIBILITY	DO	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE FOR EASY ACCESS TO SOFTWARE AND SELECTIVE USE OF ITS COMPONENTS
	FUNCTIONAL OVERLAP	FO	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE COMMON FUNCTIONS TO BOTH SYSTEMS
	FUNCTIONAL SCOPE	FS	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE REQUIRED PERFORMED FUNCTIONS: FUNCTION SPECIFICITY, FUNCTION COMMONALITY, AND FUNCTION COMPLETENESS
	GENERALITY	GE	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE BREADTH TO THE FUNCTIONS PERFORMED WITH RESPECT TO THE APPLICATION
	INDEPENDENCE	ID	•THOSE CHARACTERISTICS OF SOFTWARE WHICH DETERMINE ITS NON-DEPENDENCY ON SOFTWARE ENVIRONMENT (COMPUTING SYSTEM, OPERATING SYSTEM, UTILITIES, INPUT/OUTPUT ROUTINES, LIBRARIES)
	SYSTEM CLARITY	ST	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE CLEAR DESCRIPTION OF PROGRAM STRUCTURE IN NON-COMPLEX, EASY UNDERSTANDABLE MANNER
	SYSTEM COMPATIBILITY	SY	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE THE HARDWARE, SOFTWARE AND COMMUNICATION COMPATIBILITY OF TWO SYSTEMS
	VIRTUALITY	VR	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PRESENT SYSTEM THAT DOES NOT REQUIRE USER KNOWLEDGE OF THE PHYSICAL, LOGICAL, OR TOPOLOGICAL CHARACTERISTICS
GENERAL	MODULARITY	MO	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE A STRUCTURE OF HIGHLY COHESIVE MODULES WITH OPTIMUM COUPLING
	SELF-DESCRIPTIVENESS	SD	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE EXPLANATION OF THE IMPLEMENTATION OF FUNCTIONS
	SIMPLICITY	SI	•THOSE CHARACTERISTICS OF SOFTWARE WHICH PROVIDE FOR DEFINITION AND IMPLEMENTATION OF FUNCTIONS IN THE MOST NON-COMPLEX AND UNDERSTANDABLE MANNER

These are the same criteria again, modularity, anamoly management, and simplicity with the names of the metrics that are associated with each.

TABLE 3.3-1 QUALITY METRICS SUMMARY

CRITERION		METRIC	
NAME	ACRONYM	NAME	ACRONYM
ACCURACY	AC	ACCURACY CHECKLIST	AC.1
ANOMALY MANAGEMENT	AM	ERROR TOLERANCE/CONTROL CHECKLIST	AM.1
		IMPROPER INPUT DATA CHECKLIST	AM.2
		COMPUTATIONAL FAILURES CHECKLIST	AM.3
		HARDWARE FAULTS CHECKLIST	AM.4
		DEVICE ERRORS CHECKLIST	AM.5
		COMMUNICATION ERRORS CHECKLIST	AM.6
		NODE/COMMUNICATIONS FAILURES CHECKLIST	AM.7
APPLICATION INDEPENDENCE	AP	DATA BASE SYSTEM INDEPENDENCE	AP.1
		DATA STRUCTURE	AP.2
		ARCHITECTURE STANDARDIZATION	AP.3
		MICROCODE INDEPENDENCE	AP.4
		ALGORITHM	AP.5
AUGMENTABILITY	AT	DATA STORAGE MEASURE	AT.1
		COMPUTATION EXTENSIBILITY MEASURE	AT.2
		CHANNEL EXTENSIBILITY MEASURE	AT.3
		DESIGN EXTENSIBILITY CHECKLIST	AT.4
AUTONOMY	AU	INTERFACE COMPLEXITY MEASURE	AU.1
		SELF-SUFFICIENCY CHECKLIST	AU.2
COMMONALITY	CL	COMMUNICATIONS COMMONALITY CHECKLIST	CL.1
		DATA COMMONALITY CHECKLIST	CL.2
		COMMON VOCABULARY CHECKLIST	CL.3
COMMUNICATIVENESS	CM	USER INPUT INTERFACE MEASURE	CM.1
		USER OUTPUT INTERFACE MEASURE	CM.2
COMPLETENESS	CP	COMPLETENESS CHECKLIST	CP.1
CONSISTENCY	CS	PROCEDURE CONSISTENCY MEASURE	CS.1
		DATA CONSISTENCY MEASURE	CS.2
DISTRIBUTEDNESS	DI	DESIGN STRUCTURE CHECKLIST	DI.1
DOCUMENT ACCESSIBILITY	DO	ACCESS NO-CONTROL	DO.1
		WELL STRUCTURED DOCUMENTATION	DO.2
		SELECTIVE USABILITY	DO.3
EFFECTIVENESS	EF	PERFORMANCE REQUIREMENTS	EF.1
		ITERATIVE PROCESSING EFFICIENCY MEASURE	EF.2
		DATA USAGE EFFICIENCY MEASURE	EF.3
		STORAGE EFFICIENCY MEASURE	EF.4
FUCTIONAL OVERLAP	FO	FUNCTIONAL OVERLAP	FO.1
FUNCTIONAL SCOPE	FS	FUNCTION SPECIFICITY	FS.1
		FUNCTION COMMONALITY	FS.2
		FUNCTION COMPLETENESS	FS.3
GENERALITY	GE	MODULE REFERENCE BY OTHER MODULES	GE.1
		IMPLEMENTATION FOR GENERALITY CHECKLIST	GE.2
INDEPENDENCE	ID	SOFTWARE SYSTEM INDEPENDENCE MEASURE	ID.1
		MACHINE INDEPENDENCE MEASURE	ID.2
MODULARITY	MO	MODULAR IMPLEMENTATION MEASURE	MO.1
		MODULAR DESIGN MEASURE	MO.2
OPERABILITY	OP	OPERABILITY CHECKLIST	OP.1
RECONFIGURABILITY	RE	RESTRUCTURE CHECKLIST	RE.1
SELF-DESCRIPTIVENESS	SD	QUANTITY OF COMMENTS	SD.1
		EFFECTIVENESS OF COMMENTS MEASURE	SD.2
		DESCRIPTIVENESS OF LANGUAGE MEASURE	SD.3
SIMPLICITY	SI	DESIGN STRUCTURE MEASURE	SI.1
		STRUCTURED LANGUAGE OR PREPREPROCESSOR	SI.2
		DATA AND CONTROL FLOW COMPLEXITY MEASURE	SI.3
		CODING SIMPLICITY MEASURE	SI.4
		SPECIFICITY MEASURE	SI.5
		HALSTEAD'S LEVEL OF DIFFICULTY MEASURE	SI.6
SYSTEM ACCESSIBILITY	SS	ACCESS CONTROL CHECKLIST	SS.1
		ACCESS AUDIT CHECKLIST	SS.2
SYSTEM CLARITY	ST	INTERFACE COMPLEXITY	ST.1
		PROGRAM FLOW COMPLEXITY	ST.2
		APPLICTION FUNCTIONAL COMPLEXITY	ST.3
		COMMUNICATION COMPLEXITY	ST.4
		STRUCTURE CLARITY	ST.5
SYSTEM COMPATIBILITY	SY	COMMUNICATION COMPATIBILITY CHECKLIST	SY.1
		DATA COMPATIBILITY CHECKLIST	SY.2
		HARDWARE COMPATIBILITY CHECKLIST	SY.3
		SOFTWARE COMPATIBILITY CHECKLIST	SY.4
		DOCUMENTATION FOR OTHER SYSTEM	SY.5
TRACEABILITY	TC	CROSS REFERENCING	TC.1
TRAINING	TN	TRAINING CHECKLIST	TN.1
VIRTUALITY	VR	SYSTEM/DATA INDEPENDENCE CHECKLIST	VR.1
VISIBILITY	VS	MODULE TESTING MEASURE	VS.1
		INTEGRATION TESTING MEASURE	VS.2
		SYSTEM TESTING MEASURE	VS.3

Remember, we saw the acronyms AM.1 through AM.7 associated with anamoly management on an earlier slide. Now you can see the full name. You can also see the names for the 2 metrics of the criteria modularity. M0.1 is called Modular Implementation Measure, and M0.2 is Modular Design Measure. This chart defines each of the eight metric elements that define the metric M0.1 called Modular Implementation Measure.

CRITERIA: MODULARITY	FACTOR(S): SURVIVABILITY, MAINTAINABILITY, FLEXIBILITY, PORTABILITY, REUSABILITY, INTEROPERABILITY, EXPANDABILITY, VERIFIABILITY									
METRIC	Requirements		Prel Design		Detail Design		Implementation		Test & Integration	
M0.1 MODULAR IMPLEMENTATION MEASURE (p):	Yes/No 1 or 0	Value	Yes/No 1 or 0	Value	Yes/No 1 or 0	Value	Yes/No 1 or 0	Value	Yes/No 1 or 0	Value
(1) Top down hierarchical structure 1 - # modules with violations of hierarchy / total # modules (s)			2.0			3.0		(3.0)	2.0	
(2) Module Size Profile If less than 100 LOC excluding comments = 1, Otherwise = 100/Lines of Code (m)								0.0		
(3) Controlling parameters defined by calling module # control variables / # calling parameters (m)						3.0		0.0		
(4) Input data controlled by calling module					3.0		0.0			
(5) Output data provided to calling module					3.0		0.0			
(6) Control returned to calling module					3.0		0.0			
(7) Modules do not share temporary storage					3.0		(3.0)			
(8) Each subfunction-grouping/module represents one function # Subfunction-groupings/modules follow rule / total # subfunction-groupings/modules (s)			2.0			3.0			2.0	
METRIC VALUE = Total score from applicable elements / # applicable elements										

Each of these metric elements is scored in some way. The first three and the last one have formulas associated with them, and the rest are yes/no type answers. Yes is assigned to a one and no is assigned to a zero. I don't want to explain in detail the right part of the chart, but, in general it shows the different software lifecycle phases and where these different metric elements are applied. Metric element number one is applied during preliminary design, detail design, during implementation and during test integration. The person that scores each of these metric elements at the appropriate time sums up the scores and gives a score for the particular metric. Then, as scores are obtained for each of the metrics, associated with a particular criteria, they are summed up so you get a criteria score. Then scores for all the criteria are summed up in a particular way to get a score for the factor. So eventually we can get to the point where we can say, "Aha, we have a reliability of .95 "or "A maintainability of .8," or whatever it is.

What does it mean? Suppose we have a reliability of .95, so what? What does it mean? These are, again, the 13 quality factors whose definitions we've already looked at, and these are the rating formulas associated with each of the quality factors.

TABLE 3.1-2 SOFTWARE QUALITY FACTOR DEFINITIONS AND RATING FORMULAS

S/W CONCERN	QUALITY FACTOR	DEFINITION	RATING FORMULA	
PERFORMANCE	EFFICIENCY	EXTENT TO WHICH A RESOURCE IS UTILIZED (e.g., STORAGE, PROCESSING TIME, COMMUNICATION TIME)	1- [illegible]	**
	INTEGRITY	EXTENT TO WHICH THE SOFTWARE WILL PERFORM WITHOUT FAILURES DUE TO UNAUTHORIZED ACCESS TO THE CODE OR DATA WITHIN A SPECIFIED TIME PERIOD*.	1- ERRORS / LINES OF CODE	**
	RELIABILITY	EXTENT TO WHICH THE SOFTWARE WILL PERFORM WITHOUT ANY FAILURES WITHIN A SPECIFIED TIME PERIOD*	1- ERRORS / 100 LINES OF CODE	
	SURVIVABILITY	EXTENT TO [illegible] H THE SOFTWARE WILL PERFORM AND SUPPORT CRITICAL FUNCTIONS WITHOUT FAILURES WITHIN A SPECIFIED TIME PERIOD WHEN A PORTION OF THE SYSTEM IS INOPERABLE*.	1- ERRORS / LINES OF CODE	
	USABILITY	RELATIVE EFFORT FOR TRAINING OR SOFTWARE OPERATION (e.g., FAMILIARIZATION, INPUT PREPARATION, EXECUTION, OUTPUT INTERPRETATION)	1- [illegible]	**
INTERNAL DESIGN	CORRECTNESS	EXTENT TO WHICH THE SOFTWARE CONFORMS TO ITS SPECIFICATIONS AND STANDARDS*.	1- ERRORS / LINES OF CODE	**
	MAINTAINABILITY	EASE OF EFFORT FOR LOCATING AND FIXING A SOFTWARE FAILURE WITHIN A SPECIFIED TIME PERIOD*.	1- 0.1 (AVERAGE MANDAYS TO FIX)	
	VERIFIABILITY	RELATIVE EFFORT TO VERIFY THE SPECIFIED SOFTWARE OPERATION AND PERFORMANCE	1- EFFORT TO VERIFY / EFFORT TO DEVELOP	**
NEW USE	EXPANDABILITY	RELATIVE EFFORT TO INCREASE THE SOFTWARE CAPABILITY OR PERFORMANCE BY ENHANCING CURRENT FUNCTIONS OR BY ADDING NEW FUNCTIONS OR DATA	1- EFFORT TO EXPAND / EFFORT TO DEVELOP	
	FLEXIBILITY	EASE OF EFFORT FOR CHANGING THE SOFTWARE MISSIONS, FUNCTIONS, OR DATA TO SATISFY OTHER REQUIREMENTS*.	1- 0.05 (AVERAGE MANDAYS TO CHANGE)	
	INTEROPERABILITY	RELATIVE EFFORT TO COUPLE THE SOFTWARE OF ONE SYSTEM TO THE SOFTWARE OF ANOTHER SYSTEM.	1- EFFORT TO COUPLE / EFFORT TO DEVELOP	*
	PORTABILITY	RELATIVE EFFORT TO TRANSPORT THE SOFTWARE FOR USE IN ANOTHER ENVIRONMENT (HARDWARE CONFIGURATION AND/OR SOFTWARE SYSTEM ENVIRONMENT).	1- EFFORT TO TRANSPORT / EFFORT TO DEVELOP	
	REUSABILITY	RELATIVE EFFORT TO CONVERT A SOFTWARE COMPONENT FOR USE IN ANOTHER APPLICATION	1- EFFORT TO CONVERT / EFFORT TO DEVELOP	

* = NEW OR MODIFIED
** = NOT YET VALIDATED

NOTE: THE RATING VALUE RANGE IS FROM 0 TO 1. IF THE VALUE IS LESS THAN 0, THE RATING VALUE IS ASSIGNED TO 0

So, in this case a reliability of .95 according to this formula, which is 1- errors/100 lines of code would mean, with a metric score of .95 we'd say that we'd expect this product to have five errors per hundred lines of code. And it would be 1- 5/100=.95. These formulas enable a person to get a view of what quality level means in terms of cost, or in terms of performance, or non performance of the software. For maintainability, maintainability of .8 would be an average of two man days to fix a problem. That would be:
1-0.1 (average man days to fix) 1-0.1(2)=.8
For reuseability it's relative cost. Reuseability of .6 would mean that it took 40% of the original effort to convert this software.
1-effort to convert/effort to develop.
1-40%/100%=.6

So, it's a way of getting a handle on what is meant by these quality factors in terms of cost or performance.

Up to this point we have focused on measuring the quality of a product once it has already been developed.

One of the main thrusts of the contract that I'm working on is to be able to specify quality goals before a project even starts up and then be able to monitor, measure, or evaluate what quality level has been obtained periodically throughout the project.

The chart across the top is the system lifecycle phase as it is viewed by the Air Force. Perhaps some of you are familiar with some of the concepts for validating Full Scale Engineering Development or FSED. In the lower diagram we show loosely how the software lifecycle fits into the system lifecycle phase. This isn't exactly, it's just approximate. The typical review points are shown, like the requirements review, the Preliminary Design Review (PDR), the Critical Design Review (CDR), Test Readiness Review (TRR).

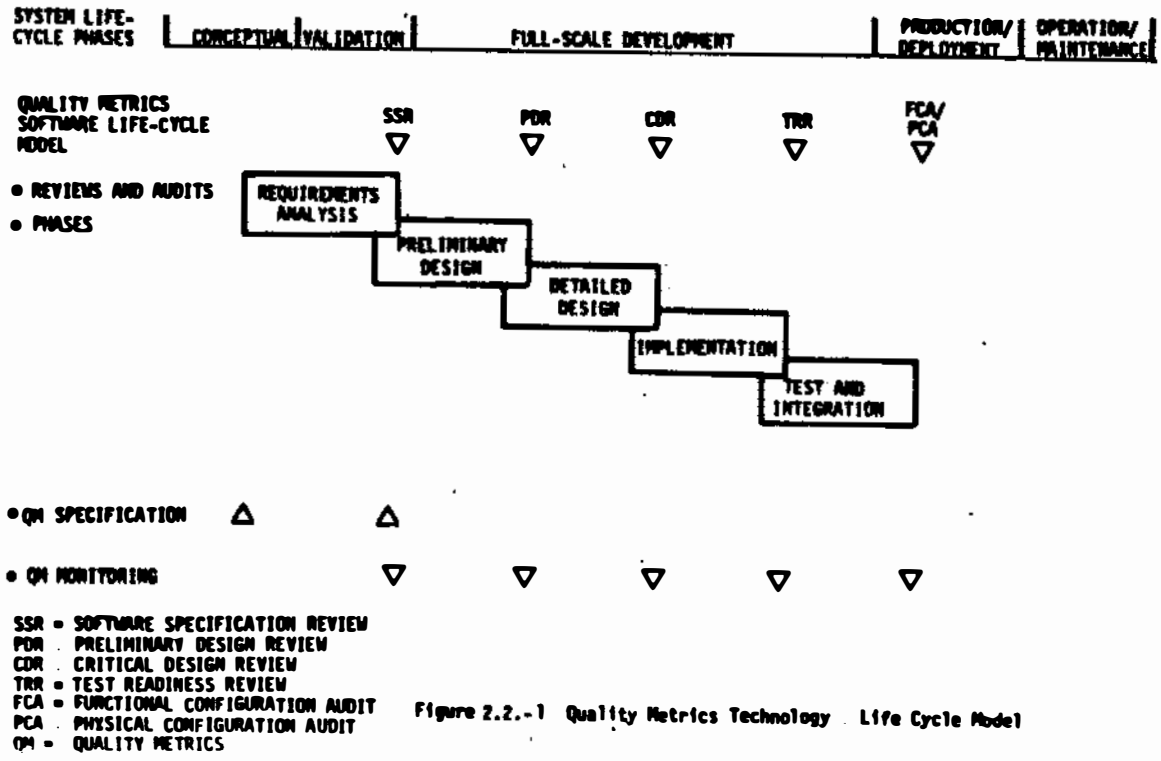


Figure 2.2.-1 Quality Metrics Technology Life Cycle Model

The little carrot at the bottom indicate the minimum number of times we comment that the quality be measured. So what we're saying we'd like to be able to do is define the technical requirements, the performance requirements and also define the set of quality requirements when the project starts up. For Air Force software, our division manager sits down and he says,"Aha, I'm going to make an intelligent decision. I'm going to ask for a reliability for the navigation software of .99 reliability and the weapons delivery software of .95 or something like that. And these are going to be defined at the same time the technical requirements are defined. It's going to be done again when the requirements are reviewed and then each time a product is put out during the development cycle." Products are typically available at review points as a result of the work done in that particular phase. SSR is a review of the requirement spec., PDR is a review of the functional design and CDR for the design detail phase and so forth. Each time a product is released, we recommend you apply these metrics and find out where you are. If you're supposed to have a reliability of .99, find out what it looks like at various points, how it evolves over time. Give the people the feedback. This is good not only for the aquisition manager, he can see he's getting what he asked for, but also for the project manager and engineers, too. They can get the feedback as to how they're doing.

Here are some of the features and limitations of this quality metric technology.

Features and Limitations

- **Features**
  - **Quantitative indication of quality level**
  - **Ability to specify quality goals**
  - **Ability to evaluate quality levels during development**
  - **Early detection of deficiencies and anomalies**
  - **Cost avoidance for high reliability, correctness, maintainability, and verifiability**
  - **Long-term savings for high expandability, flexibility, interoperability, portability, and reusability**
- **Limitations**
  - **Subjective metric scoring**
  - **Automation - Language proliferation**
  - **Validation - Data availability from projects**

One of the big advantages is that there is a quantitative indication of quality level, some number. Now that's good, yet if you ask me how good I can't give you an exact answer. For example, let's say I'm through applying these metrics and I've come out with a quality level of .95. How accurate is that? Well, during this contract, we're getting a little looser and saying "Well, let's go to a rating of high medium and low because we're not real sure if it's .95 or.96." Plus we do get an indication, and it is a quantitative one, of quality level. That's better than what we had.

Ability to specify quality goals is a big feature. We're now able to ask for what we want ahead of time instead of complaining about it later on. Evaluating quality levels during the development and giving periodic feedback to everybody could be very beneficial. Some of these metrics will result in the detection of deficiencies and anamolies early in the development cycle. This happens simply by asking these kinds of questions of the products that come out. Cost avoidance for high reliability, correctness, maintainability, verifiability can now be achieved. We don't have time to go into the details of that now, but, in part of the work that I am doing now I'm finding out that persons who specify a high amount of reliability, correctness, maintainability and verifiability, these four quality factors, are in all likelihood going to save money through cost avoidance later in the project. For example in high reliability, there would be fewer errors to detect, correct, so forth. The cost of the time spent in specifying high reliability and then going out and measuring whether it is there would be more than offset by cost savings, or cost avoidance. By having high reliability there's long term savings for reuse quality factors. That's kind of intuitive.If we're building something to be reuseable, it's most likely because we can save money reusing it. Some of the limitations are subjective metric scoring. If I have a software product here, and I ask two people to apply a set of metrics, for example, for reliability, the score is going to come out differently, because people score things differently. We're trying to clarify the definitions so you know the metrics will be well defined and there's a minimum possibility of interpreting the measures in different ways.

Automation would be helpful in the sense of scanning source code and finding out what these metric scores are. The problem there is the number of different languages that exist today for existing products. For new products ADA might help in taking care of that problem for the command control type applications.

Validation? well the concept of validation is that if we have metrics and we're able to apply them to a product and come out with a metric score, and if we have a rating formula which says, OK what that really means is that we've got 5 errors for every 100 lines of code. How well do those track? How do you validate how well those track? And if each metric varies, up or down, will it affect this score or not." How valid is the metric with respect to what we're trying to measure?

One of the problems from the past has been availability of this kind of data from projects. They don't seem to see that we need information at the module level like configuration management type of information,

cost information, error information, in order to do a good statistical analysis. Projects just don't keep information at that level of detail today. That has been a problem area, and I think Ed will be addressing some of those problems.

BIOGRAPHY: Thomas P. Bowen, a software research engineer, has been with Boeing Aerospace Company in Seattle,, Washington since 1967. Prior to joining Boeing he was with IBM. His BS is from the Unversity of Washington.

SOFTWARE INTEROPERABILITY

P. Edward Presson
Software Research Engineer
Boeing Aerospace Company

The text of this paper was transcribed from videotape. It has been edited for clarity in written form but purposely left in a distinctly conversational tone. Slides and pictures have been used where appropriate.

The contract that I'm going to talk about principally in the next 20 minutes is the one on interoperability and reuseability.

Contract F30602-80-C-0265 (RADC)
Software Interoperability and Reuseability
Final Report D182-11340-1,-2

The original work that McCall had done for Rome Air Development Center was to be extended to a couple of quality factors that the original investigators didn't have data on. Mainly, interoperability and reuseability. This, I think, is maybe an extreme case of what happens when you take a technique and try to apply it in a very questionable area. So I won't be telling you a great success story today. I will be pointing out some of the problems we incurred when you use software metrics, especially when you try to prove that the predictions they make are correct.

We'll talk about data collection first. There are actually two kinds of data we need to collect here if you are going to do this type of validation. Really the contract called for us to do two kinds of work. First of all develop what they call a metric framework. It is to pick out what combination of things you can measure that will ultimately impact the interoperability or the reuseability of software. Once you theoretically have an idea what things might contribute to that, to try to validate it. That means data. It means you go out, gather the metrics, and gather measurements that you conjecture might be right, compute the metric values of the predictions and then try to prove or disprove that the predictions are related to what happened in the real world. In other words, if we predicted very high interoperability, did the real project have high interoperability and the same thing for reuseability?

Now in order to get the measurements, the framework was designed to be applied during the whole lifecycle. Some of the metrics were applied during the requirements stage of the product development. Other measurements were applied at the conceptual, basically the design review stage, when you've got a design but you haven't actually written the code yet. The final set of metrics, is at the time where you've got a program that's ready to go into test.

We had to get data, and it was a laborious process on the whole, because there weren't a lot of tools, to help us. We went through documentation to extract the information we were looking for about what was or was not in requirements that might lead to good results on it's reuseability and interoperability. We looked at source code listings, and here we had some help because the various projects we looked at occasionally had tools to extract information for their own purposes. We could use these tools to get information we needed for metric computation. And finally through review. We finally interviewed the people on the projects to see whether or not the deductions or conclusions we were extracting from these various sources were really correct. We supplemented the actual counting with some interviews to see if we were going the right way and correctly representing what had happened on the project.

I was primarily involved with interoperability. Part of the problem is that interoperability doesn't have a specific definition. It has a number of definitions. The Air Force and all the military are all very concerned about interoperability. What they mean by it, is "How will one system that has imbedded software in it work with another system that also has imbedded software in it, when they weren't originally designed to work together?"

Interoperability - Definitions

1. The effort to couple one system with another.
   McCall, et. al. RADC-TR-80-109

2. The ability of systems, units, or forces to provide services to and accept services from other systems, units, or forces and to use the services so exchanged to enable them to operate effectively together.

   DODD 2010.6 Standardization and Interoperability of Weapon Systems and Equipment Within the North Atlantic Treaty Organization (NATO), 11 March 1977.

3. (The ability of one service's system to receive and process intelligible information of mutual interest transmitted by another service's system.)

   JINTACCS Interoperability
   Reference PM99 21 Dec 1974 HQDA

Probably the best example I use is the AWACS air plane. I don't know if you're familiar with it, it's been in the news more recently. it has sort of a pancake disc on top. It's a radar surveillance airplane. It has lots of computer power on board to track all sorts of targets in the air and on the ground. When NATO was interested in purchasing the AWACS aircraft, they wanted it to work with their own ground radar installations. This is where NATO would have their own system of radar picking up information and tracking airplanes and so forth. So the systems had to interoperate, that is, what the AWACS

airplane picked up had to be able to transmit to the ground. It not only had to be right in terms of the format, number of bits per word and basic things like that, but it also had to be interoperable in the sense that the information transmitted had the same meaning to the people on the ground as it did to the people in the airplane. There were lots of problems as you might guess. The AWACS is a moving platform, so everything is relative to it as far as it's concerned. The NATO sites were ground based radar. They wanted everything in reference to their own fixed geometric and geographic configuration. That's interoperability.

The original conjecture that McCall used in his work was "How can we predict the effort that couples one system with another?" This is kind of nice from a management point of view. If you know how much effort is required, you can make estimates of cost and schedules. But in terms of actual definition, it doesn't mean much. So one of the things I did was try to find out what the Air Force and the other paramilitary people meant by it. Here's a couple of their definitions What they said, in a very general way was we want everything to work together in all ways you can imagine. That's interoperability.

Now we tried to come up with a frame work and conjecture what kind of things would contribute to that factor. In order to do that we read as many articles as were available on the subject and talked to a lot of people. Talking to people turned out to be one of the best things because I didn't come on the project originally. The people who were there first had spent a lot of time doing desk work conjecturing about what factors influence interoperability. The more they thought about it, the more they thought that everything influenced interoperablity. The metric framework was getting bigger and bigger, to the point you could logically say, "Well, since this takes place at a system level, you are talking about one system interoperating with another one, not just one program working with another one. Everything does impact everything." Well, that's unworkable for a lot of reasons and that's probably not right either. So we started doing lots of interviews with managers and engineers who had experience working on projects where they had faced some sort of problem that related to interoperability. From that we built a frame work, considerably reduced from the one that was originally projected. We reviewed case histories, and tried to have all the people we interviewed look at our conjectures and tell us which things from their experience did or did not influence interoperability.

The results of the interviews were kind of interesting. They felt that the most critical decisions you were going to make on a project were very early decisions. They were very basic decisions about the framework of the system you were about to design. The interface specifications for the system should be made as specific as possible. Not general in the sense that it accepts almost everything and rejects almost nothing, but the interface requirements should be very specific. You must specify exactly what data should look like or what format it should be in or what it should mean to the user.

The third one may not be quite as obvious. What do we mean by operational procedures? In a simple sense it means how the data gets operated on at each end by each system. The procedure that the user sees and that each system uses are almost as important as the actual interface between the two systems. And I harp back to the previous example, you can transmit data back and forth real nice, but if the information coming back from an AWACS is telling you how a plane is moving with relation to itself, a moving platform, that doesn't help the guy on the ground who wants to know how fast a plane is moving toward Berlin. So, the procedure, that is, how the data is used, is as important to know as the interface that the data goes through.

So, based on our interviews here was our framework.

Interoperability Framework - Criteria

Augmentability	AG
Commonality	CL
Communicativeness	CM
Independence	ID
Modularity	MO
System Compatibility	SY
Functional Overlap	FO

Rejected Criteria

- Anomaly Management
- Simplicity
- Consistency
- Coupling Factor
- Self-Descriptiveness

We still had two main criteria, (remember, criteria is one level below the software quality factor of interoperability).

COLLECTING DATA

We needed as much data as we could get and unfortunately, we only found three actual projects which had interoperability problems that is they had to modify their code or meet certain requirements to interoperate with other systems, and also had data we could use. So we ended up looking at three different systems, which I called A B C. The number of modules you see there is not the modules in the systems, these are the systems. They were very large, on the order in some cases of three or four hundred modules. So we looked at it as part of the project, isolated particular modules that had to be changed, modified or worked on to cause interoperability.

INTEROPERABILITY CRITERIA SCORES

CRITERIA		MEAN	STD. DEVIATION
Commonality	CL	.58	.40
Communicativeness	CM	.54	.24
System Compatibility	SY	.20	.29
Augmentability	AG	.24	.18
Independence	ID	.99	.01
Functional Overlap	FO	.32	.39
Modularity	MO	.76	.58

We computed the criteria scores and you can see that our standard deviation shows we varied all over the place.

Interoperability Data Collection

SYSTEM	A	B	C
Number of Modules	57	11	18
Lines of Code	5253	1937	6122

We're coming up on one of the major problems with this kind of validation, getting data. When you're done collecting data, and it turns out to be more nebulous than it first appeared, you have a problem. The software managers and engineers tell you "Of course we have records of our whole project, we know exactly what happened." We know through experience that it begins looking like a cloud. From a distance it looks solid, but the closer you get, the foggier it becomes.

We had another problem that I haven't addressed yet that is particularlly pertinent for interoperability. Interoperability was defined as a system level measure. We couldn't measure or ask questions about the interoperability of a subprogram because that, according to definition, would make no sense. We had to talk about the interoperability of a system. So the interoperability ratings are only available at a system level. We looked into various approaches to try to estimate interoperability. We looked at effort figures and we looked at productivity figures, that is, "Were people more productive when they were trying to produce interoperable software than they were

otherwise?" We also did sort of a quasi Delphi Survey to estimate relative efforts devoted to software that had to be interoperable versus the other parts of the software that didn't have to be. Does anybody know what a Delphi Survey is? A friend of mine characterized it as going to all the oracles you can find, and getting their opinion. But it's more formalized than that because you do some statistics and then you go back and ask them if they want to change their mind based on seeing what all the other oracles said. The result is a Delphi Survey. We've had a lot of problems, because it turns out interoperability is very difficult to get at because it is difficult to nail down what it is and it is difficult to measure how interoperable that system is.

So we finally took sort of an effort reading. That is a combination of efforts required to get the programs to be interoperable in comparison to other programs that didn't have to be interoperable. Then we set up a system of equations. Those of you who are mathematicians will know that since we only had three systems, and we only had three independent variables, that is, we had three systems to look at and three scores to look at that were interoperable, so, finding a solution to that system of equations was problematic, to say the least. That was compounded by an exploratory data analysis which indicated that there was a large unexplained variablity in the data. That's a statistical technique that allows you to look at your data and see if your data looks like it's going to be good data even before you're doing anything with it. It sort of worried us-we were in bad shape even before we started. The first solutions we got we took all our criteria, set up a system of equations, dumped in the values that we got, solved them and came out with something being negatively correlated, with interoperability. Some of the things which we conjectured of course seemed to contribute directly to it. So that seemed counter intuitive . Then we decided to do a sensitivity analysis and using techniques that,(and not being an expert mathematician in that area), I can't explain other than we did some things like change the system interoperability rating very slightly to see if that changed the relative contribution of the various metrics afforded. Well, it turned out, that a very small change in your estimate of the system interoperability would make a wild difference in the solution of the equation. It would say, oh yes, this one now contributes very positively to the interoperability where previously it was considered negative.

So we had a system of equations that as you can see had some negative factors that made no sense at all. You can see the data points and values on the chart.

System	Worksheet Scores			IOP Rating
	1	2	3	
A	.375	.364	.606	.752
B	.417	.515	.653	.606
C	.618	.400	.751	.909

Interoperability Rating:

$$R = 1 - \frac{effort_{iop}}{effort_{iop} + effort_{orig}}$$

First Solution: Rating $= a_1WS_1 + a_2WS_2 + a_3WS_3$

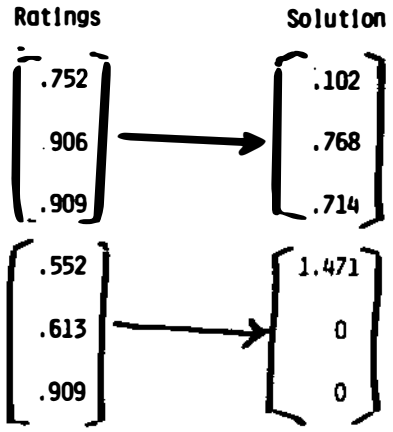
$$\begin{bmatrix} -6.81 & -1.59 & 2.62 \end{bmatrix}$$

not intuitively reasonable

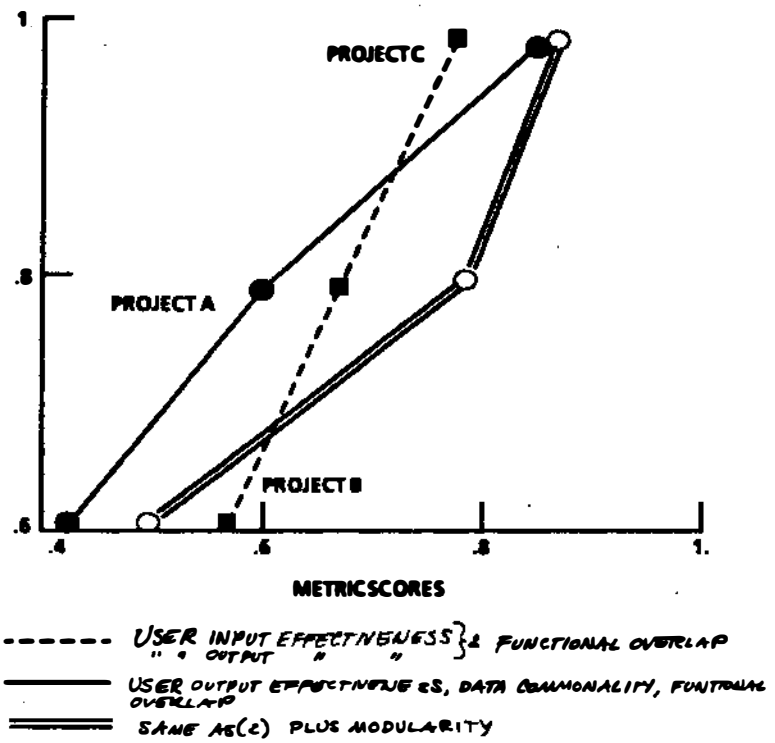
We changed the ratings slightly, which showed the solutions could swing wildly back and forth, so we really didn't know what was going on at least statistically we didn't know what was going on.

INTEROPERABILITY

Other Reasonable Ratings Can Produce Reasonable, But Highly Varying Results



That's the only way to count it. So we tried some different approaches. We said, well, let's not look at the criteria, which is what we're looking at here, let's look at some submetrics within criteria and see if those in combination seem to indicate the actual results.



INTEROPERABILITY RATINGS vs. METRIC PREDICTIONS

We did find a combination that looks pretty good, but I wouldn't wager a lot of money on it because, as we've seen, the data itself is very nebulous. That was kind of not too bad, unfortunately,what got left off was the combination of metrics that produced those results.

I can tell you roughly it was part of function overlap which is a way of saying how much of the functions done by this system is also done by another system. It's a combination of functional overlap and how well the system communicates with human beings it has to deal with. That's actually important for this. It turns out that as you start reading the literature and looking at how the military interoperates, one of the key interface points between 2 systems is a human being. So the ability of the system to communiate with a human in some cases is actually very important because he's the link.

Interoperability - Conclusions /Recommendations

- Unable to Completely validate Interoperability Metric Framework
- Some relationships were observed
- Review Working Definition of Interoperability
  - Effort May Not Be Best Characteristic to Measure
  - Need a Quantitative Measure
- Data Collection Should be a CDRL Requirement

We then made some conclusions and recommendations. We found some results. These have gone into a report that should be available now as an RDC technical report. It looks like maybe there's some relationship between these measurements and interoperability. We don't want to go out and try to predict interoperability based on those, what we're saying is "Check out some more, further work needs to be done.". When you're looking at a system measure like interoperability you need to look at a lot of systems before you can predict anything. Maybe we need a better measure of interoperability. Maybe the things we were working with and the way we were looking at it, weren't a true reflection of the things itself. That was a problem we had in this contract, especially with interoperability that you wouldn't have with say, reliability. Some of the factors that Tom talked about are pretty well defined. Everybody knows pretty well how you measure them. But if you're trying to predict something like interoperability where there's no real measure for it, it's hard to say whether your predictions are right or not because you can't prove it, you can't measure the thing itself. So, one definite recommendation then is that if you're going to try to validate a relationship or if you're going to

try to validate any kind of measure that predicts something, make sure you can measure the thing you're predicting, or otherwise you'll never prove your predictions are right.

If there is any real conclusion to all of this, it is that- beware if you see a paper that says I've got the answer and all you have to do is measure these things, and use this formula and you can predict exactly what you want, because it probably won't work. It needs the proper validation and is much more sensitive and much more complex than it first appears. And as we talked at this mornings meeting what we need to do is come up with better standards of what you measure and how you compute what you measure so we can all begin to compare results and speak the same language. In other words, we'll be more interoperable, I guess.

BIOGRAPHY: P Edward Presson is a software research engineer with Boeing Aerospace Company in Seattle, Washington. He has been with Boeing since 1962. His BSEE and MSEE are from the University of New Mexico Ed has been on the IEEE Metrics subcommittee for over a year and is familiar with achievements and future direction of their work.

A REAL METRIC SURVEY

William J. Moore
Engineering Quality Manager
Metheus Corporation

The text of this paper was transcribed from videotape. It has been edited for clarity in written form but purposely left in a distinctly conversational tone. Slides and pictures have been included where appropriate.

What I am going to be talking about today is a practical case study that I did some time ago. What I am hoping to do is share with you the experiences and good results that we had in conducting a software quality requirements survey. The division where this was done, is a fairly large division, and they literally make all kinds of software and all kinds of products. So, there was a lot of concern as to what was really important in this division and what we could do with respect to addressing software quality problems.

There was a report that came out in 1980 called the McCall Report. This was the basis for the survey I conducted. The 11 factors I'll be discussing were part of the report that McCall did. His report recommended that before you get into software quality metrics, you should conduct a survey on a given project to find out which things are important, and try to concentrate on measuring those types of things. Well, doing metrics on any project where I was working at t he time would have been pretty difficult. What I was interested in doing was expanding this and doing the survey over the entire division. I wanted to find out what people really thought about software quality and where they thought the problems were.

To that end, I had three objectives I wanted to accomplish with this particular survey. First, I wanted to determine what the real perceptions were and find out if there were any conflicts of interest or differences between different groups. In a large organization, you tend to push certain things, perhaps the group over in one corner have a different need than the group in the other corner. I wanted to find out if that was true. Second, I wanted to find out what was\really important, and then third, use that information as a focus on any improvement efforts that we wanted to do.

Now, a big problem we had going into this thing, was we wanted to make sure that we got valid responses from people. To avoid getting a motherhood type of response I worked with a number of market research people and came up with a really simple questionnaire. It's essentially two questions.

In the first question we simply asked people to rank the 11 factors in order of importance. We wanted people to make definite decisions about which factors were most important to them. In conjunction with that, we wanted to get an idea of how well people thought we were doing. For the second question we asked them to rank how well they

thought we were doing as an organization, on each of the factors. The same thing here, we limited it to only three possible responses. Sometimes these surveys come out with a scale of ten or something. We wanted to know if someone had inclinations definitely one way or another and we kind of forced them into doing that with a choice of only 3 responses. From the response profile, you can see how responsive different groups were to the survey.

RESPONSE PROFILE

Job description	
Software engineers	35
Software eval. engr.	4
All other engr.	31
Engr project/program mgr.	17
Engr Eval Mgrs.	5
Division/Bus unit staff	3
Marketing	5
Anonymous	4
	104

We also collected information on what type of software people were working on and whether they were really involved directy or indirectly. We also knew what projects they were working on, where they were working and who they were working for. I took all this information so I could form various categories from which I could make comparisons as to whether or not this group or this type of people that worked on this type of software were having different needs than other people.

Essentially, we sent the survey out to everybody. We did not use any discrimination whatsoever in sending it out. Marketing and sales had only five people return surveys. There were actually more like 150 sent out to marketing and sales. They also had a large sales organization in this area. One reason we sent it to everybody, we wanted to know if there was anybody in sales or marketing that had any strong feelings. We didn't get a lot of response there. Probably the most vocal response we got was from all other engineers, particularly electrical engineers. They seemed to have very strong feelings. I did not have a comment section on the survey, but they wrote all over the place. They had a lot of very strong feelings. The other response that was interesting was the division business unit staff. We were organized into business units which reported to a division staff. Of the three responses I got, they came from the business unit general managers. They were also very vocal. So it was interesting to see there was a lot of concern on varying levels.

After tracking the responses for about three weeks, the only conclusion that I could come up with, was, that everybody in this division thinks the same way. Now, you might say it could be the way the survey was constructed. The way I avoided that ~~was~~ with those questions was to have three different surveys. The particular things were arranged

randomly, so it gave three different samples. We used that to determine whether or not the way the survey was presented created difficulty. Sometimes the way you list the questions creates a mind set that kind of forces answers. There was no difference in the responses for the 3 surveys. That was very encouraging, because you are always hearing "They don't understand my problem" For whatever reason, everybody in this division that responded seemed to think almost alike. There is very little variance in the results that we got.

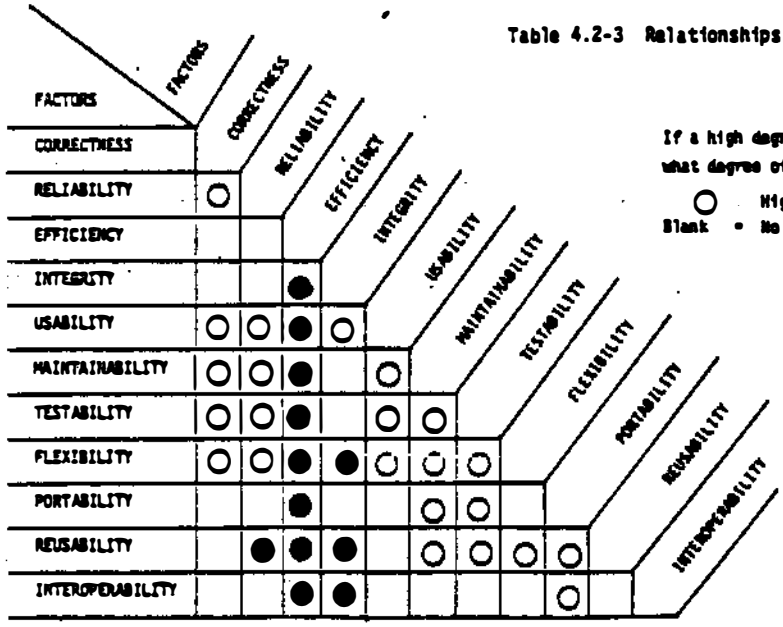
SOFTWARE QUALITY FACTOR RANKINGS

Rank	Factor
1	Correctness
2	Reliability
3	Useability
4	Maintainability
5	Efficiency
6	Testability
7	Flexibility
8	Portability
9	Reuseability
10	Interoperability
11	Integrity

The first thing was the ranking. This is just how it happened to work out for that group. A couple of things are of interest, correctness and reliability. If you've ever seen McCall's report those definitions look almost identical. In fact you could probably switch them in most cases. It was probably a poor choice of words. The interesting thing was that a lot of people mentioned that they seemed the same, but everybody consistently rated correctness over reliability. They didn't seem to have any doubt in their mind which was more important.

The other factor I'd like to mention is efficiency. Now efficiency had a pretty high rating compared to some of the other work that came out in McCall's paper.

Table 4.2-3 Relationships Between Software Quality Factors



This is a little bit difficult to interpret. But essentially, there are quality factors on each side and what they are saying is, if you put a lot of emphasis on factors coming down from the top, what effect should you see on these other factors. The dark spots indicate that it has a negative impact, and the white circles indicate it seems to have a very positive impact, and the blank dots seem kind of neutral. Now where you see the big row of black dots, that is the efficiency, if you put increased emphasis on efficiency. The only thing it didn't seem to affect was correctness and reliability. This was from the RADC study. A part of the reason this may have happened in this particular division, was, that a number of major projects somewhat prior to this had difficulty with running out of memory. And as some of the people said, maybe that caused a lot of consternation. The project managers for those projects rated the efficiency considerably higher, than even the number of problems allowed at release. That could have been a factor in the survey. It seemed to be important to a lot of people, but, what most of the studies are showing is this could have a very negative impact on an awful lot of other factors that might be important as well.

The next question was how well we did.

SOFTWARE QUALITY FACTOR SUCCESS RATINGS

Factor	Rating (1-3)
Correctness	1.88
Reliability	1.93
Useability	2.02
Maintainability	2.38
Efficiency	2.07
Testability	2.29
Flexibility	2.24
Portability	2.39
Reuseability	2.31
Interoperability	2.38
Integrity	2.38

The ratings were either 1, 2, or 3. 1 was very high success, 2 was moderately successful and 3 was not successful. Somewhat surprising was that the average all together, was not very good. It seemed to be moderately successful for correctness and reliability, not very high scores. Now, a couple of things are interesting about this. One, this particular organization did have very high customer acceptance of quality in the field, something that they are very proud of, and customer surveys tend to bear that out. But still the people on the line, the people from inside the division, voiced in the survey that they didn't think they were doing as well as they would like to do. Now, you may notice that there is somewhat of a linear relationship here between the top one and the bottom one. This in fact did hold

true. One other thing, was that the maintainability factor, number four which was a 2.38, even at 95% confidence levels sticks way out as being "We don't do this very well." And that was kind of interesting, because maintainability and efficiency, together can have a lot of impact on a lot of things. How you do maintainability is really a direct outgrowth of what your design implementation practices are. In one of the organizations I was dealing with we essentially made up a check list. We knew we weren't going to measure a thing but we had a checklist of the kinds of things the individual designer and testers should be looking for. A lot of maintainability things are really basic, things within design and implementation.

Essentially, there were three major results. There are a lot of other results, too, but I didn't want to get into much detail on them. The first result was that the perception of quality in this organization was very consistent between groups. If they wanted to address the problems, they'd all be addressing them from very similar perspectives. A lot of people found that somewhat interesting.

The second result is their general perception that the achievement of software quality had only been moderately successful. Exactly why that was, I don't know. It was kind of a surprise, we really expected to come out fairly middle of the road. There's a lot of proud people there that do a lot of good work.

The last one is probably the most significant. Achieving low maintainability and a lot of emphasis on efficiency were the primary obstacles for actually increasing the level of quality being achieved.

Essentially what I hope I've shown, is that you can do things with these metrics, you can find information out about your organization, and you can apply them in your projects. Even if you don't get down to counting lines of code and paths and things, there are certain things you can do with them.

BIOGRAPHY William J. Moore is the Engineering Quality Manager at Metheus Corporation in Hillsboro, Oregon. Before joining Metheus he was with the Information Display Division of Tektronix in Wilsonville, Oregon and SAI Comsystems in San Diego, California. He has a BS from the the U.S. Naval Academy and an MBA from National University in San Diego, California.

SOFTWARE TESTING

Nelson Mills
R&D Software Manager
Hewlett Packard

The text of this paper was transcribed from videotape. It has been edited for clarity in written form but purposely left in a distinctly conversational tone. Slides and pictures have been included where appropriate.

When Ken Oar asked me to come and speak at this conference, he told me I could choose a subject on something we were doing well here or something we were doing here that's not working well. At the risk of giving the impression that we don't know what we're doing here, I decided to talk about something we're doing here that's not working well. In fact, it's something that I don't think is working well in the industry in general. The subject that I want to talk about is software testing.

In a typical software project, approximately 50% of the time and 50% of the cost will be consumed in some type of testing activity. During the past decade we have made substantial progress in this industry in the area of improving software development methodologies and tools. We've just put a tremendous amount of energy into that area and really made some improvements in terms of things like design methodologies, structured design, some languages that support that, and compiler compilers that feed in the syntax and generate the front end of the compiler. I would like to maintain that similar investments improving our testing tools and techniques would have a tremendous payoff in terms of the quality of the final products we produce, our productivity and the time it takes to get new products to market. In the business that we're in here at the Portable Computer division at HP it's almost a commodity market. Timing for the market is extremely important. There's a window during which a product is right. A few month's later it's not right and it costs a tremendous amount of money if you miss that window.

I want to start out with some definitions. To begin with, testing is the process of executing a program with the intent of finding errors. A good test case is the one that has a high probability of detecting an as yet undetected error. A successful test case is one that does indeed find errors. These definitions are from the ART OF SOFTWARE TESTING.

What I want to do now is talk about the varied aspects of testing. The first one may be a bit strange and may not be regarded as a testing process, but I would like to maintain that it is. I would like to talk a little bit about design testing. Around here, design testing is usually done by some kind of peer review of your design concept, if it is done at all. Occasionally it is done by developing a bunch of code to implement a portion of a user interface. That's the most common area we would use this for here. It is code that we would throw away

later, because it's not the real product. What we're doing, is implementing enough of the front end of the product to give ourselves some confidence that this is going to be useable by our end user. Then we throw it away and go to work writing the real thing.What I would like to see in this area is the development of some rapid prototyping tools that would in particular support the development and testing of the user interface. This isn't terribly important if we're talking about operating systems, or compilers, but we're also talking about applications that are aimed at the non computer sophisticates. Frequently our peers aren't the best people to test those user interface concepts to tell us if we've done a good job. What we would really like is to be able to quickly implement that user interface with some stubs to support the rest of the program and try it out on some real users to see what their responses are.

The second type of testing I would like to address is module testing. I'm assuming a project is fairly well structured and broken down into some fairly small logical modules. And when the modules are completed, the modules will be tested. Around here again this is done by the development engineer, he does it by writing some throw away tests. He sits down at his terminal and he determines what he wants to test, he writes the test, executes it throws it away and goes on to the next one. His objective in doing this is to prove his program is correct. If you remember, I mentioned earlier that the objective of testing is to expose undisclosed errors. Here is this guy doing the testing, and he's the guy who designed the program, he's the guy who wrote it and his objective is to prove he did a good job. I think this is not being critical of program developers, I was one myself. I just think it's a very tough thing to sit down and create a program and then sit down and try to destroy it and actually approach it from that attitude. I'm finished with the process of creating this thing and now I'm going to sit down and prove I didn't do a good job. Since he is the designer and he has made mistakes in his program, he'll probably repeat that mistake when he does his test. Since I'm assuming that these are small modules, and therefore the logic is fairly well exposed, we want to concentrate on some white box testing. That is, I would like to see some logic driven auto test generators and test monitors. I would like to see a test generator that would not only generate test data, but generate the expected results then run the test and compare the result against the expected results. And also, flow path generators so we could validate that we have in fact covered all the logic in the program.

The next stage in my testing scenario is system testing, and by this I mean we've now tested one or more of several modules and now the people who wrote these modules are beginning to put them together and test parts of the whole system and the whole system itself. Around here, two of these programmers who tested the modules, sit down and test the system the same way they tested the modules. They think about what it is they want to check before they sit down at the terminal, they write the tests, they execute them and they're gone. They're objective is the same as I mentioned earlier, to prove that it all works. At this point the thing has gotten a little more complicated and the logic isn't quite as exposed and I think we need to move more into the area of functional tests, some input/output driven auto test generators and

test monitors.

The final stage in this process is the acceptance testing. This is typically done by the test team or the QA team, depending on the particular project and where the resources happen to lie. We do use test suites. These are suites of programs that are expected to remain around a while. So we run through them the first time and find all the errors the developers didn't find for reasons I just discussed. Then we put it back into development phase, fix those errors and then rerun those same set of tests to make sure when we fixed those errors we didn't create a new set of errors. All of us that have worked on programs know there is a high probability of that. If you go into a program to fix an area, the probability of creating an error is high. So what we would like is a test suite that is complete, that we can run over and over as we go through this process of correcting errors, and correcting the errors we created when we corrected the first errors. The problem around here is that this is a labor intensive manual process so the test suite is almost never complete. We're not sure we have complete coverage. We do the best we can but with limited resources we usually don't have complete coverage, so again a lot of the testing is done by individuals at terminals. It's not part of the test suite. A lot of it doesn't get saved. A lot of it is thrown away and if we have to rerun it, we typically have to depend on the people taking good notes as they ran the labor intensive manual part of the tests and reproducing exactly what they did the previous time they ran them.

I wouldn't like you to think that I'm picking on testing people, because I'm not. What I'm really trying to say, is the industry has essentially ignored this aspect of the development process. It hasn't gotten anywhere near the attention that we've given to developing tools to assist the developers in creating programs. Compared to that, the testing process has been fairly well ignored. I guess what I'm saying, is I don't think we can afford to do that any more. The Japanese, in their fifth generation computer plans make it very clear they are attacking this whole area of software development, including automatic program generation and automatic test development. And I think that if we're going to remain competitive, we have to do the same thing.

I feel that the quality in HP products has been high, but acheived at a pretty tremendous cost. I think the efficiency of creating products would be greatly enhanced by doing some of the things I have suggested. Sometimes we get really bad things out, but not very often. Sometimes we have projects that slip, one in particular slipped pretty bad. On this product they sent out a statement saying they were expecting some real problems, that was a year and half ago, and we're still waiting to hear. Sometimes we've had products in the testing phase a lot longer than they should be. We lost a lot of money on that product. We spent a lot of money fixing problems we may not have had to. It makes us nervous about our understanding of our customer needs. But, it's better than going the other way.

BIOGRAPHY: Nelson Mills received his B.S. in mathematics from Albion College in Albion Michigan in 1961. After five years in the Navy, he joined Control Data Corporation as a systems programmer. In 1976 he joined Hewlett Packard and worked on the HP-85 firmware. He is currently a software section manager at HP's Portable Computer Division in Corvallis.

TECHNIQUES AND TOOLS USED AT INTEL

Janice Cleary
Software Engineering Manager
Intel Corporation

The text of this paper was transcribed from videotape. It has been edited for clarity in written form but purposely left in a distinctly conversational tone. Slides and pictures have been included where appropriate.

The focus of my talk tends toward the large projects. The overall process is geared toward the large projects. Hopefully somewhere along the way these phases can be trimmed down. Someone asked me earlier today how I managed to get through all these steps that I'm going to describe here. I'll only ask you to pick up a copy of our new book and that will help you understand how we do it. I would like to also focus on using the quality indicators to the advantage of the individuals on the project team as well as the organization as a whole. My belief is that all of the members of the project team are equally responsible for the product quality. Typically the development engineers get saddled with the entire responsibility and the QA engineer comes along and says you didn't produce a quality product. All of the responsibility seems to reside with this one individual or group of individuals. I don't believe that you can really expect that one person to carry the entire burden. It takes all the members of the project team to produce the product and they all need to be responsible.

When I am discussing the project team I might include:

Project manager

Project leader

Marketing engineer

Development engineer

Technical writer

Evaluation engineer

Quality assurance engineer

Sustaining engineer

Support engineer

We have one or more project leaders depending on the size of the project. In projects that I have been working on recently we use 15 development engineers and evaluation engineers on the project. We always include the sustaining support and quality people in the project at a very early stage. We think it is really important that the entire project team agrees on the specifications and understand the goals for the project right from the beginning.

There are seven steps in achieving quality projects.

Requirements

Exploration

Definition

Architecture

Design

Implementation

Test

I break them down into three phases. The first stage I call the definition stage and it actually requires both exploration and definition. I think the requirements stage and exploration stage is an interim process that produces the definition and therefore I put them together in one phase. The architectural and design phase is also an indicator itself that the completion of those two steps produces design documentation. The implementation and the test is the final step in the process.

The requirements phase is a very initial definition of a priortized set of objectives with description of features and design constraints to be given to the project team. I think it's important that before beginning a project each member of a team understands what the objectives are and is able to review all of the documents produced with the same set of goals and understanding of what we are trying to achieve.

We then go into an exploration which should consist of defining several alternatives to solving the same problem. I believe that different members of the team might have different approaches and techniques. Each of the different alternatives defined can be explored in a free discussion environment and then each approach should be tested against the objectives. Typically we might go back and redefine some of the objectives based on the fact that none of the options that we've been able to come up with can address all of the features that the marketing group wants to put into the product. Some tend to be mutually exclusive. I'd like to get rid of those before we even start trying to specify the architecture of the software.

At that point we have the definition completed which is a finalized set of objectives and assumptions based on the feasibility study done during

the exploration stage. Achieving buy in from the project team on the goals and objectives will have major impact on the software quality. The important aspect here is the project team needs to believe they can complete the task they've been assigned to do.

In the overall design, the architecture specification is actually broken into two phases. The external phase and the overall design from a user point of view. It defines how the user will interact with this software and how all of the components might interact with each other. It's important to define what all the pieces are.

In this model I used an operating system. The system might be a major piece of the design. From the architecture level you would define how the device driver interface looks to the user of the operating system, but you wouldn't necessarily go into the description of each device driver in the system. So the architecture level is a high level design.

At the same level, the internal architecture specification is intended to imply the development environment and the language each designer should be using. The coding convention and methodologies are all defined in the internal architecture specification. I think that at this point we can now divide and carry on the overall project if you allow the component pieces to be developed independently.

The external design defines the "user" interface where "user" is defined as the person or program which must interface with this product.The distinction between architecture and design specification becomes more important when you get 3 or 4 modules to interface. Now you'll have three or four people in similar type modules, and again using a device driver as an example, the external specification for the device driver interface might be produced at one time and then used by four or five different engineers. I think that it's important that the specification be written in such a way that it is useful to the technical writer and the evaluation engineer as well as the development engineer. Being able to see the product from this point of view while developing their tests or writing their draft of the manual helps the project.

The internal specification would be the next phase and is pretty much a pseudo code representation of the software product. The intention here is that prior to writing code the logic is clearly and well thought through. Each procedure, subroutine and package is defined and all aspects of each module within a single program should be clearly specified at this point. The internal specification I like to use as the example for the engineers, is that when you're completed with this project, you should be able to give this specification and the code to the sustaining group and they'll never have to come back and find you again. If it's well written, I believe that can be done.

The implementation phase is where most of our current expense is. Time for coding, manual draft, test development and the review process for each of those activities is a large amount of the development environment.

The code reviews are intended to ensure compliance with the internal specifications. If you've done a good review of the internal specification, then you should have the logic of the program well understood and clean. Then all you need to do is be sure the code meets the specification and then it should be perfect. At times we go to other organizations within the company who might have worked on other operating systems that have a lot of expertise in a particular area, and use their help and experience to do the code reviews. The project leader on the evaluation team is generally the person who picks the review teams. It's that person's reponsibility to make sure that the reviewers are qualified to do the reviewing and give valid feedback.

Manual reviews are intended to ensure the eccuracy and useability of the product documentation. At this stage the primary focus is to assure that the documents are accurate and we do this by having all the members of the project team participate in the review. Not everyone participates in the code review and we pretty much try to select a group of people who understand the area being reviewed.

At the same time that the developers code the product being developed we also produce an evaluation plan which is written by the evaluation group. They describe what they are going to test, how they are going to test it and also a description of their test. We do test reviews as well as reviews of the product and this hopefully helps to get the developers mindset into the evaluators mind and hopefully helps that process by having the designers review the test engineers code as well as vice versa. Really the key here is that all members of the team participate in some aspect of the review phase.

In the testing phase, there is a design engineering testing phase that we go through before we get into evaluation. It is really the responsibility of the project leader to determine at which point in time all known problems have been resolved and the product is ready to be transmitted to the evaluation group. We really strive for removing the inner problems that the developer has found before transferring it to the evaluation group. The designer should feel that he is ready to release the product at that time.

During the evaluation phase the product is reviewed not only from a "Does this meet the specifications" point of view but also from a point of view of "This is the package I'm going to get in the mail when I release my product. I'm going to sit down at a system and plug in a disc, does that whole process flow? Do I have a manual that tells me what I need to know? Is the manual readily available to me or do I have to move to secondary documentation to figure out the first manual?" The evaluation test phase not only includes testing to the specification but the useability of the product.

And so I've consolidated that whole procedure for achieving software quality into three phases that I feel are a key and also are things that are equally as lengthy in the design process. Those are

1. Presenting a clear set of goals and objectives

2. A well thought through design
3. Methodical approach to implementation

This process is pretty much a standard process that we have at Intel.

During the next series of projects that we are going to be starting in our area, we are going to measure that process at each phase. I plan to use this measure at each stage of the development by documenting and logging every problem as it is identified. I believe it's very important to document our problems. One of our first presentations this morning noted that the Japanese DOA rate is based on any problem that you encounter no matter how serious it is. I think it's important to keep that in mind when you are documenting at each stage of the development cycle so we can understand how to improve our process. In addition to taking it through the development stage I'm also suggesting that we extend it out 90 days after shipping it to a customer. That window is an arbitrary 90 days that I set to try to put an end date when I am going to look at all the statistics that I've gathered. I am just picking 90 days arbitrarily as an end time to evaluate the data. If our process works to the point that we want it to work there shouldn't be any reported problems after the release of the product. That should always be the ultimate goal.

The development methodology that we use in our design is to ensure quality software. Therefore I believe the identified problems should decrease in number at each stage of the development cycle. During the design phase we should identify the most number of problems, and during the implementation phase we should shake out quite a few more. In the test phase there should be a limited number of problems identified and then zero when we ship the product. I'm going to measure that so I can achieve it. I don't know what it is and I don't know where I'm spending most of my time and I don't know what area I need to focus on. So I think it's really important to measure our problems. I think that the result from that will be very beneficial, for the team overall and for each individual in understanding the areas that they need to focus on to improve themselves.

I intend to use software quality measurement to benefit all memebers of the project team and the organization as a whole.

BIOGRAPHY Janice Cleary graduated from the Computer Programming Institute of Rhode Island in 1970. She joined Foxboro Company that same year, Digital Equipment in 1976 and Intel in 1979. She is currently the software engineering manager for the Interated Systems Operation.

TOOLS AND TECHNIQUES

Bruce Coorpender
R and D Manager
Tektronix

The text of this paper was transcribed from videotape. It has been edited for clarity in written form but purposely left in a distinctly conversational tone. Slides and pictures have been included where appropriate.

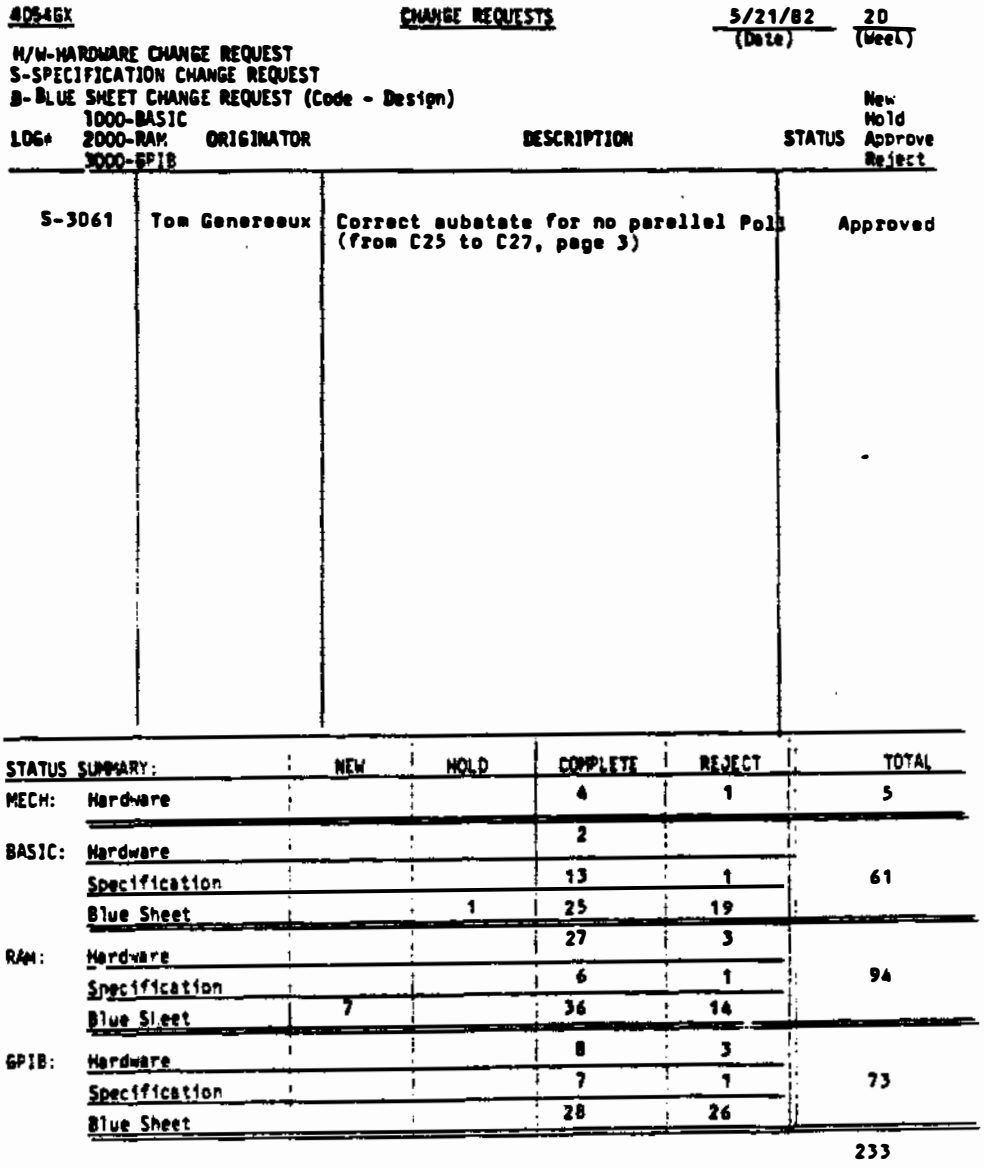
What I want to talk about today is tools and techniques. One of the slides that I picked off Steve Swerling's keynote speech this morning is "How do we measure quality?" There were four points to that. To determine whether the desired level of quality is met, to assess the quality of the test suites, to help make management decisions for whether to release the product or not, and to build a historical test base or data base. What I'm going to present is three tools or techniques and set up some metrics for performing those tests.

IMPLEMENTATION PHASE PROBLEM REPORTING

To define implementation phase in the context that I am going to discuss it here, is the time from where the internal design is complete until we ship the product to the customer. So it encompasses the coding and the testing phase. One of the first things you need to do is to track all the change requests and bug reports. All of them no matter what the source. I found that it is valuable for a project and it's not just software and firmware projects. It also works for electrical, mechanical, and microcode type things. I've managed typically all these things, hardware, mechanical, all the aspects of a project, and I use the same techniques. I must admit to a little surprise though, the hardware engineers are asking to do design reviews and actually do schematics review, a more appropriate name on a hardware project than code review.

When I talk about change requests and bug reports, one of the things that I find convenient to do is use a standard form. Don't reinvent a unique form for a new project. Most companies have some type of change request or bug report form. Use that same form so everyone involved doesn't have to repeat the learning curve. Define a review/approval method and responsibility for it. There are always going to be changes in any project. I can absolutely guarantee that every person requesting a change will have perfectly sound and logical reasons for requesting that change. But you can't make all the changes people request. Somebody has to have the responsibility for determining which ones get implemented and which ones get put in the bin labeled "This is a good idea, but we don't have time right now" or "We're not going to do that one at all." You need to set this process up. Typically my reaction is to delegate that responsibility to the design engineer. I believe in a high degree of autonomy in design engineering, because they are the ones who are ultimately responsible for the product. Everyone on the project team is responsible, but the ones that are

actually doing work are more responsible. So, they have the bottom line jurisdiction. I typically don't get involved in assessing those change requests over a project leader in a given area. They process all change request reports through the tracking function first. This ensures that the reports don't get lost. Handing them off to the reviewer for an approval process sometimes means they don't get in the tracking system. I brought some examples of some kinds of mechanisms I have used in the past.



40S4GX — CHANGE REQUESTS — 5/21/82 (Date) — 20 (Week)

H/W-HARDWARE CHANGE REQUEST
S-SPECIFICATION CHANGE REQUEST
B-BLUE SHEET CHANGE REQUEST (Code - Design)
1000-BASIC
2000-RAM
3000-GPIB

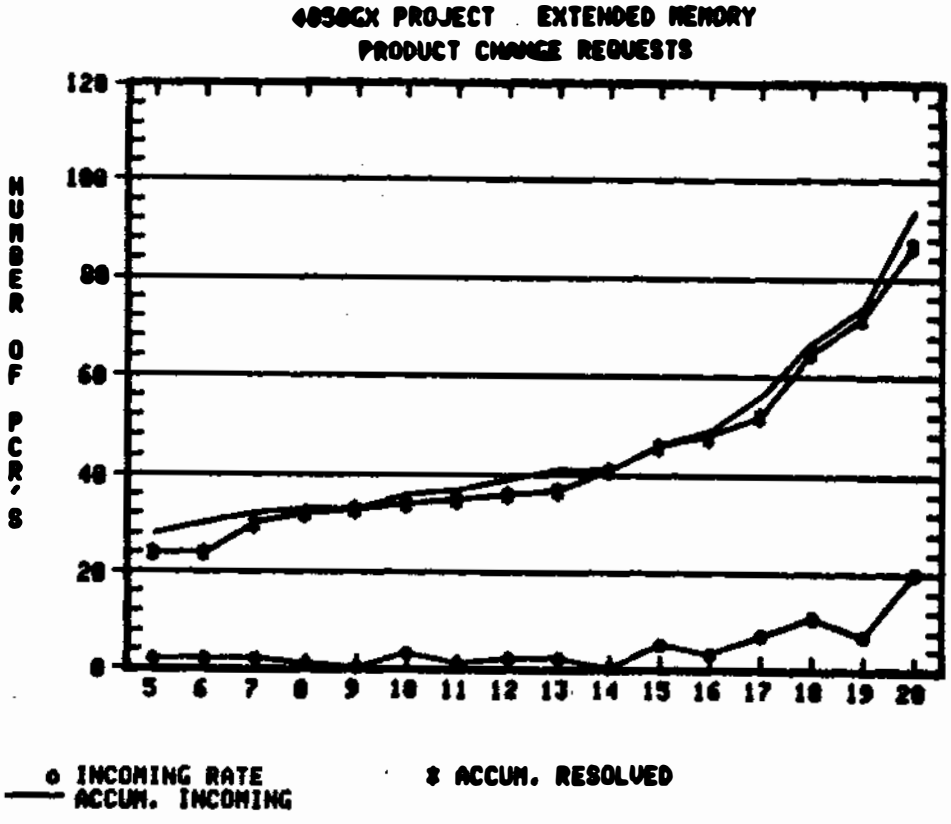
LOG#	ORIGINATOR	DESCRIPTION	STATUS (New / Hold / Approve / Reject)
S-3061	Tom Genereaux	Correct substate for no parallel Poll (from C25 to C27, page 3)	Approved

STATUS SUMMARY:		NEW	HOLD	COMPLETE	REJECT	TOTAL
MECH:	Hardware			4	1	5
BASIC:	Hardware			2		
	Specification			13	1	61
	Blue Sheet		1	25	19	
RAM:	Hardware			27	3	
	Specification			6	1	94
	Blue Sheet	7		36	14	
GPIB:	Hardware			8	3	
	Specification			7	1	73
	Blue Sheet			28	26	
						233

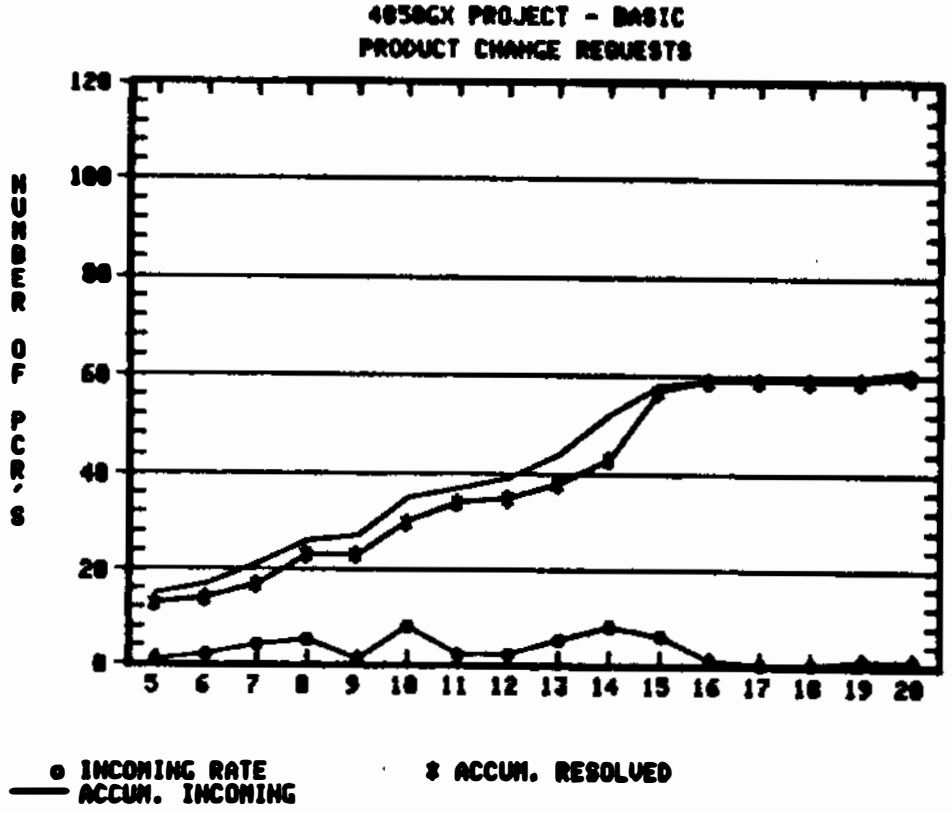
This is a summary report that I used on a project we completed a year ago. This data was summarized on a weekly basis and reviewed at a project team meeting. This would track the type of change request, (hardware, software, whatever) identify new elements that came into the system that week, what the status is and then some summary information on how the project is doing to date. You can see here that mechanical had only four requests completed and one was rejected, for a total of 5. The overall Blue Sheet (bug sheet) in the extended ram file manager summary had 36 that were completed, and 14 rejected, a total of 94, some of them were in hardware, and some in specification.

As I look at the data coming in, it's a curve for each one of the areas. Typically, it's an S shaped curve, such that I expect relatively few change requests and bug reports in the early phase of the project. As people start to move out on the learning curve they find out what the product is, how to use it, how to express it, and then they really start finding the problems. You will see a high rate of change in the product at that point, you get a lot of incoming bug reports, and then a leveling off as you gain some stability of the product.

This graph is a snapshot of one of the last programs in the earlier phases, where you still see this accumulated rate rising. The lower line on the graph is the instantaneous response on a weekly basis. The accumulated basis is still rising on a fairly sharp slope that tells me they haven't caught that upper knee of the curve yet. This program was not ready for release at that time.



By contrast, at a little later date, we look at another element of the project on the following graph. Although it appears that they have the same type of instantaneous data, (lower line),they had moved up the S shaped curve, past that knee, so this plot is looking fairly quiet. In fact, I see one of the people who was involved in this project in the audience today. We're looking at a fairly high level of confidence and a fairly stable product and one of the reasons is we have few incoming bug sheets.



A DESIGN REVIEW PROCESS

One of the ways to help enhance that is to adopt and enforce standard design rules (syntax, etc) so that you're not always learning the other person's hierarchical design language, pseudo code, syntax, or anything else. I don't dictate that, but I ask the engineers to get together and as a group process, set up something they all feel good about, and that they can live with. They tend to be self policing.

When you do a review, archive the fact that the review occurred. Who was a participant in that and when it occurred. Like the design I want the archives imbedded with code. The design should be in some kind of pseudo code, or some powerful design language that can be imbedded with code as a comment section. That gives you the only chance in the world that exists to update the design documentation when you update the code. It's not a guarantee, but at least you optimize the

opportunity that it will take place. Then review every design, or redesign. If someone has to go back and make major changes in their design, you need to review that redesign, as well as the original design.

In the code review process, again, use commonality, so you don't have to repeat the learning curve for everyone involved. Use commmon syntax to permit the optimum chance to catch syntax errors. Use common names, publish a data dictionary that refers to what those names are. Occasionally bring in outside experts, people with expertise in a given area that you happen to be dealing with. But I have found that the most value at the code review as well as design reviews, comes from the originator of the design of the code, walking somebody else through that, and all of a sudden, somebody will ask him a question. Oh yea!!! and the question didn't have anything to do with the sudden realization, it was just a trigger mechanism. The majority of the errors that were found in the design and coding phases, were found by the originators themselves, but typically triggered by a response to somebody else.

A secondary effect of the coding and design review process is that it increases communication to other members of the project team. Like " Oh you're doing it that way, that means, that I'm going to have to do this". So, it's a very valuable process for both the designer and participant. Like the design review we review all new code and major revisions to the code. You go in and fix the bug, let it be your judgement, but you don't always have to review the code.

In terms of corrections costs, anything that we can catch earlier is less expensive than things that are caught later. Design time is the least expensive time to correct errors. If you end up recalling a product to correct a problem in the field it may cost 100 to 1000times as much as in the design phase. Probably it's going to be in the 100 number if you are in a soft product such that your software is loaded from some kind of magnetic media. It tends to the 1000 number if your software or firmware is silicon based, and you actually have to send out for new parts and put them in the mailing or if you have someone out there in the field, physically installing the new product.

When you are going through the design or code review situation, you gain information as you resolve the errors. First of all the process is used to identify that some fault condition exists. Second, you can resolve the problem and track the error cause if its something that you caught in the coding review. Was it a coding problem? Was it a syntax problem? Or something inherent in the architecture? Capture that information, because you can use that to help in future code reviews. Feed that back to other projects as well, particulary if they are using the same kind of design and coding tools, because that may be something common to the particular language or tool. Document the number of errors by module to identify candidates for intensive evaluation and rewrite. That's not a salary performance type tool. I don't want to look at it from that kind of management sense, but if I had tracked that a particular module has three or four times the error rate of any other module, that's a good candidate for at least a very intensive

evaluation and possibly an entire rewrite. I don't look at the statistics, I expect the project leaders to do that, beause I don't want to get involved too much in a management sense.

In that historical data base, one of the tools we used was something called the post completion audit. I use that term rather than post mortem because I don't like the project to die. I like it to be complete. I think it's not a finger pointing exercise, "Gee, because quality assurance didn't do this, we have this problem." This is to indicate what went well and what did not go well. It's not to involve compensation or performance metrics or anything else, but to figure out how to do things better next time.

THE PROJECT PLAN OUTLINE

One way to do this is to follow the project plan outline. A typical outline of project plan that I like to see has these parts, maybe a few more.

TYPICAL OUTLINE

- INTRODUCTION
- OBJECTIVES
- CONSTRAINTS
- RESOURCE UTILIZATION
- SCHEDULE
- ASSUMPTIONS, RISKS & CONTINGENCIES
- PROJECT CONTROL PROCEDURES

An introduction to a project plan and to the post completion audit should include a description of what this project is all about, including a brief description of the product. If management reads it very likely all they have the time and inclination to read is the first two paragraphs. That's a good way to give a synopsis of the whole project. Describe the objective for the performance, user interface, marketing, etc. Then in the post completion audit, how well did you meet those objectives? Which ones were satisfied and which ones weren't satisfied? What were the constraints? What resource utilization did we expect and how well did we use those resources? Did we need additional equipment or additional people to meet the schedule goals? What were the schedule goals, and how well did we meet them? Did we hit our targets? The most important thing is product shipment time. The intermediate ones can vary a little bit, but the bottom line is to get your product shipped on time. This indicates whether you've had a very successful project.

There should be a statement of assumptions, risks and contingencies. What are we assuming in the development of this project? State those, particularly in the project plan. Everyone can see what our

assumptions are, and if they are one of the assumptions, they can either protest, or recognize a commitment to them. The risk contingencies are a simple expression of what can go wrong and what you will do about it if it does. The example I like to use here is in this particular program that we were doing. We were doing a module that we anticipated would take 12K of code so the electronic engineer put 16K worth of EPROMS on the board. We got to looking at this as a potential risk situation and the question came up "what will happen if we go to a little over 16K , say 17K?" Well, the design engineer said "Well, all I have to do is to put another chip on board, and I could put the pads on there so we could just drop it in." That was incorporated into the design as a result of the risk analysis. The code went to 17K and finally ended up at 22K. But, the only thing it accomplished was to add $12 in manufacturing costs. No schedule impact and no impact in any other design difference because it was all in place. That happened only because it was asked at the front end. Otherwise we would have been in a panic mode, redesigned the circuit board to put in another set of pads for the extra EPROMS. It cost absolutely nothing in the first design.

In the post completion audit, what did we track there? There's going to be some risks that come up that we didn't anticipate. Capture those in the post completion audit so you can look at those the next time we do a project and plan for how we might get around that.

How effective was our project control procedure? How did we run the project? Did it get in the way? Did we need more control than we had? I tend to use a weekly status report for each group involved in the project. And then a weekly status meeting report to allow everyone to review that defect. Did we need more, did we need less, how does that all work?

The value truly gained out of those audits is central archives to provide history. In fact, Chuck's organization at Tektronix acts as a repository for those things and several hundred post completion audits, design specifications that are good examples, project plans, all kinds of other documents to allow historical usage. You base estimates for time and resources on what is archived. You can also avoid potential problems by looking at the historical data base.

BIOGRAPHY: Bruce Coorpender received his BS in Math in 1974 at the University of Texas at Arlington and a MSCS in 1977 at the same University. He's held many different positions and has many honors. Who's Who in the West, Who's Who in Technology Today, and Epsilon Pi Epsilon a computer science honor society. He currently works for Tektronix as an engineering program manager.

COSTS OF SOFTWARE QUALITY

Paul Nichols
Reliability Engineering Manager
Information Networks Division
Hewlett Packard

The text of this paper was transcribed from videotape. It has been edited for clarity in written form but purposely left in a distinctly conversational tone. Slides and pictures have been included where appropriate.

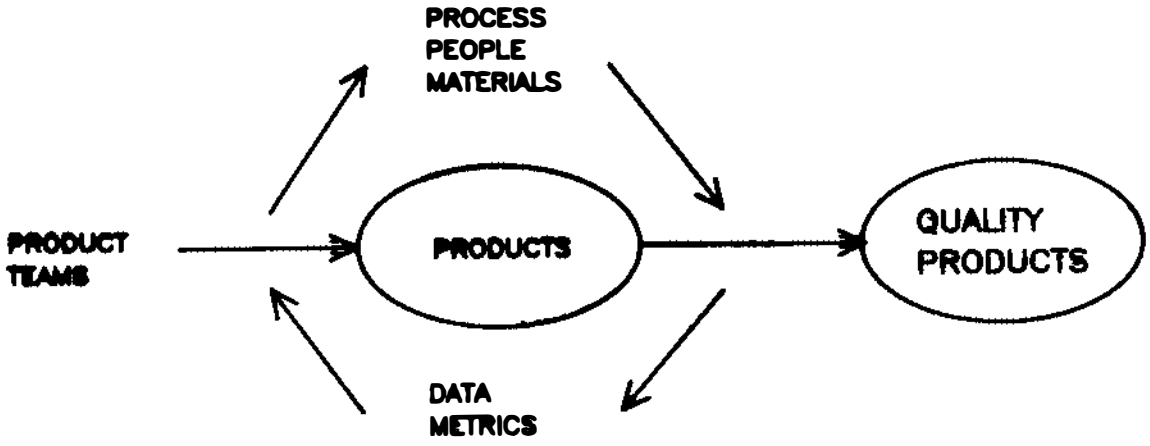
One of the things you find out when you talk to HP people is you get a lot of different answers. The reason is that we are a very diversified company. We have a lot of different computers such as desk top computers and interactive computers. We're at a lot of different locations. We build all kinds of different software with medical applications, software imbedded in our instruments, interactive software, desktop software, networking software and office software. There are lot of different problems and lots of different users. Then to make matters even more complicated, we have what we call the HP environment or the HP way, which says we trust people's judgement. We don't rely on bureaucratic rules although we do have standard software product lifecycles that we use at different divisions. They are used but they are tailored for those divisions which means they are used in different ways. We do emphasize contribution. HP has long been noted for innovation.

Where do I fit into this maze? Information networks division produces data bases and data communication products for the HP 3000. Essentially we have two software R&D labs, one for databases and one for data communications. We have about 180 engineers in R&D and a support staff. It's not only the HP 3000, it's also the HP 1000 today and we currently have about 25 other software products. That's the group of products our division is responsible for. I'm not the overall software QA manager, for the division. I represent the group of people that develop internal tools for use in the R&D facility and we support those tools.

We have established a reputation for excellence in technical innovation, and yet we face the same problems that everybody else does. Our reliability varies widely from product to product depending on a lot of factors, and we are still trying to understand what those factors are. The software does become critical path from time to time. We do experience scheduling problems and we have been doing some things to try and change that. Our software development costs are high. This workshop is going to be on software costs. One of the things I'd like to try to do is to put the business aspect of software into perspective. I don't think we've looked at it from a business point of view. In other words, we have a small group of product engineers that are managed by a project manager who has complete control over the project. He makes the decisions in a team environment, but still he is the one who is responsible for the project leadership. The product team is made up of people from all the functional areas, including,

marketing, manufacturing, quality assurance and R&D. What goes into that process is obviously people, the resources they consume and the capital equipment. There is a process that we use called product lifecycle. The role that Quality Assurance plays is gathering data, using metrics, analyzing that and giving feedback about the process and how it can be improved. Essentially our division manager, Andre Schwager has made a statement that QA owns the process. We're reponsible for analyzing that process.

QUALITY IMPROVEMENT PROCESS



"QA owns the process"
A. SCHWAGER 11/81

Let's look at it from a cost standpoint. If you could clear your mind and think of yourself as owning a software firm, your primary concern would be your return on investment. You're in business primarily to stay in business and that means you need to make money. So in order to be in business and to protect your return on your investment, you have to predict what your costs are going to be. How much is it going to cost me, how big is it going to be, how many people am I going to need, what are my resources, that type of thing. Then you'll have to track expenses as you're building this product to see if you're within cost estimates. Those costs feed into pricing for the product. You need to

know where the risk areas are, where you're going to be very tight, and what kind of risk issues are going to be coming up.

To manage that process we use lifecycles.There are various types of lifecycles that you can use. We use the traditional waterfall lifecycle. We do manipulate that lifecycle to a certain extent. We have milestones and checkpoints that produce feedback loops to see if we've met our goals. We try to automate the process as much as possible, and I notice a lot of discussion has been taking place today in the area of tools. That is my particular interest, too. We try to use tools as extensively as possible to try and automate the process whereever we can.

There are a lot of different things that are tracked in hardware.

FINANCIAL MODEL-HARDWARE PRODUCTS

Phase	Costs Estimated	Actuals Tracked
Investigation	Staffing	Staffing (R&D)
	Materials	Materials
	Preliminary ROI	
Design	Staffing	Staffing (R&D)
	Materials	Materials
	Manufacturing	Testing
	Support	
	Failure	
	Warranty	
	Testing	
Manufacturing	Staffing	Staffing
	Materials	Materials
	Process	Equipment
	Failure	Failure
	Testing	Testing
Introduction	Documentation	Documentation
		Travel
Support Life	Warranty	Warranty
	Production	Field Support
		Repair
		Production

If you look at a financial model for hardware products, the kinds of things you're going to be interested in looking at are estimating and tracking. Before I talk about that I'd better talk about the different phases in our lifecycle. Investigation phase at HP basically means that we have an idea for a new product or that we want to invent a follow on product. In this phase we take a look at the technology we need to use for that product. We estimate staffing requirements, the materials and do a preliminary return on investment. We look at the process we need to use to get that done. Then we write a proposal which is reviewed by management. If it's a good idea to staff that project we move into the

design phase. For software, the design occurs in two phases, external specification and then internal specifications. On the hardware side we look at a lot of different estimations of cost. And I think this is really different from any other company. There are a lot of different things to look at when you're estimating hardware costs. What is it going to cost you to manufacture this product? What are your support costs? What is your warranty, if you're going to have one? HP products have a one year warranty on instrumentation and a 90 day warranty on computer products. We try to estimate what the warranty costs are going to be. We try to estimate what the testing costs are going to be. The staffing estimates only include R and D staffing. The testing is done by a separate group for hardware and we track our testing expenses separately. That does include salaries. We also estimate material and equipment. In the manufacturing process we look at the actual manufacturing process itself and what it's going to cost us, the bills of materials, try to predict a failure rate on the line during the test, and the equipment we need to do that with.

The software slide is from my own experience at HP.

FINANCIAL MODEL-SOFTWARE PRODUCTS

Phase	Cost Estimated	Cost Tracked
Investigation	Staffing (R&D)	Staffing (R&D)
Design	Staffing (R&D)	Staffing (R&D)
Introduction	Documentation Cost	Documentation Cost Travel Cost
Support Life		CPE Costs (R&D)

Typically, software is not driven by cost but rather driven by the marketplace. In other words, by virtue of the market that you are in, you have to add some software to make it attractive to sell in order to create the complete solution for the customer. That's where the industry is at today. And so typically we don't really worry about software costs up front like we do for hardware. We worry about them after the fact when schedules slip and it starts to cost a lot more than you expect. For software you're mainly looking at staffing costs and documentation costs. When we price software, we don't price it against what it cost you to produce that software. The rest of the market hasn't priced their software at what it cost them to produce that software. If you price your software at your costing algorithm you'd never sell your software. The point that I'm trying to make is that the software industry hasn't matured yet like the hardware industry has. The conclusion is that hardware is cost based and software is a question that depends on the business you're in. For HP it certainly is the marketing base. Not necessarily cost base. So our ability to maximize profit and make decisions against different kinds of software is limited because we're not making the same kinds of business decisions that we're used to making with hardware.

I'd like to give you some ideas about my thinking on what we need to do.

1) Track our costs during development.

The way to do that is by using a project management system and tying it back to our accounting system. That is a way of creating a mapping of predicted costs against what your actual costs are.

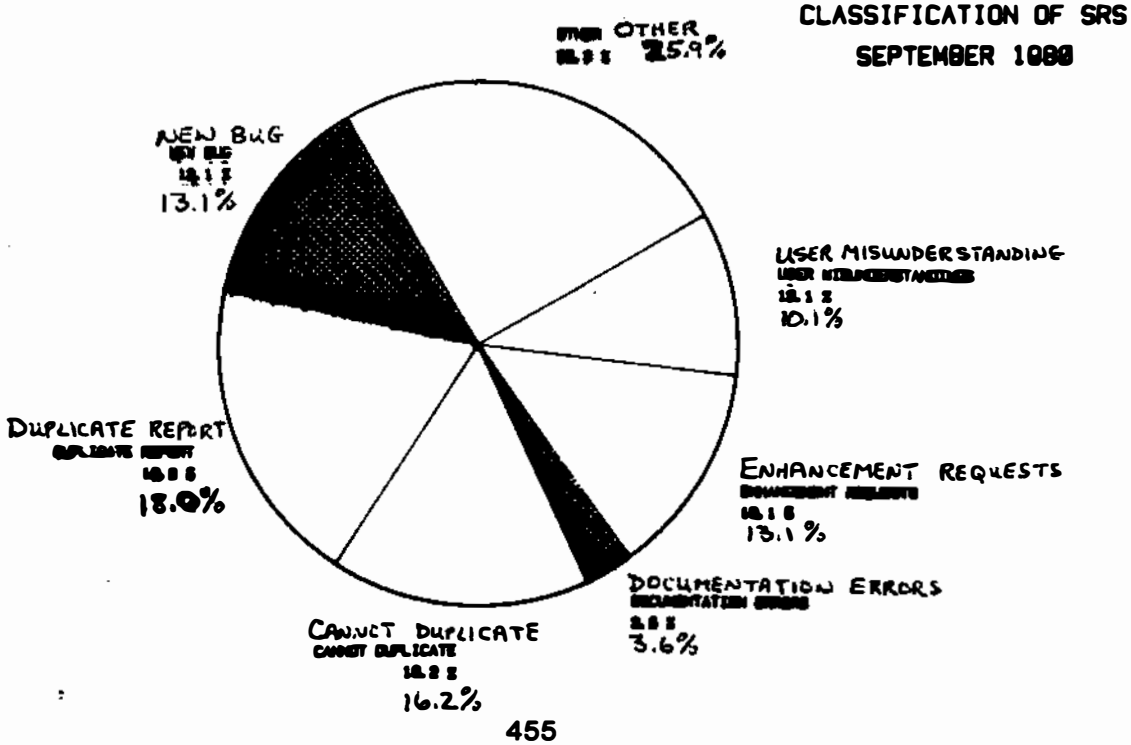
2) When software fails in the field we would like to tie it back to the originating entity.

In this case the particular division that produced that software. We should make that cost failure show up on that division's profit and loss statement.

3) We need to make more systematic buy vs make decisions.

In other words, you don't necessarily have to build all the software, there are a lot of OEMs that can do it. Software is expanding rapidly, especially in the personal computer area. There are a lot of opportunities to buy software and provide support for that software. HP is starting to do that with the HP plus program where we make a purchase of software through a third party vendor and supply a certain level of support for that software.

4) Tracking the quality costs which is really the root of what we want to talk about today. We do track some costs at HP. To track software cost you need data. Without the data you can't analyze or determine what it's costing you and you don't know where you can reduce your costs. Like the old adage, if you can't measure it you can't manage it. One of the ways we track software is through a system called STARS (software tracking recording system). All the service requests from our customers come into this data base. This gives us an opportunity to capture everything in one place and analyze it.



When we get a request from the field, we classify it. The service request goes through two levels of classification, one by marketing people beyond my support area and then secondly the lab R and D team. The lab engineer responsible for maintaining that software then classifies it. Then there's negotiating back and forth. Sometimes an R and D engineer says that "this is not a problem" or "it's an enhancement request and not a bug". Then there'll be negotiation back and forth until they determine what that really is. But essentially we're interested in all these different areas. A new bug is a new problem that we recognize as a software failure that we need to do something about. We need to fix it. A duplicate report has a cost associated with it. In fact every service request that comes to the factory has a cost associated with it and just minimizing the amount of paper that comes into the factory is an important cost reduction. We can reduce the duplicate reports by providing timely information to our customers about problems in our software. Sometimes we can't duplicate it. If there's a problem in the software that a customer says he or she has, and the field has verified that they can reproduce that problem, the fact that we can't reproduce it tells the factory they don't have the right test scenarios set up or that they don't have the right equipment set up to duplicate that problem. Then maybe they need to look at something in that area. Documentation errors include when the documentation isn't adequate to explain how our software works. Enhancement requests mean that we missed the target market possibly or the target user, or that we haven't communicated to the field and the customer base our intentions to follow up releases with certain kinds of enhancements that we were going to provide. User misunderstanding is really a reflection back of documentation that's poor. Typically a feature that was misinterpreted by a customer or a product that was not as well designed as it could have been and a customer had different expections of what that software should do. So we do some tracking and some analysis right off this broad data saying where can we improve the process.

Another way we measure ourselves is with bug rates. We count the absolute number of bugs at any point in time. We know there is a cost associated with fixing those bugs. The bugs are weighted by severity. We have 4 severity levels, critical, serious, normal and low. You can get an idea of how long it takes to fix certain level of bugs so we can associate cost with that kind of bug. We normalize those things by thousands of lines of code. We've defined a line of code as a line of executable code that does not include comments or blank lines, that allows us to standardize these measures across divisions. That allows us to do trend analysis on the bugs and say "Are we getting better or worse?" We also use weighted numberings against severity codes. In other words, a severe bug is worse than a low bug and so we can do some weightings. Those are useful in comparing products. They are also useful during development for making milestone decisions. During the development we produce a document called a quality plan. In the quality plan we have goals and objectives that we set for each of the different milestones we have to meet. It would say what our bug rates

have to be before releasing a product. We can use this information for determing if the bug rates are coming down during the testing phase and if we feel comfortable with release.

We can also use this information for deciding on manpower. For instance if we have a very large back log of bugs to fix then we know we need to put more people on maintenance. It helps us decide if we should hire more people to do that or do we shift emphasis from new product development to product maintenance. In planning for the future we need to look at how we can do a better job. One of the things we have implemented recently is tracking the true cause of the error. We've developed about fifteen or twenty different categories of errors, such as an initialized variable, bad internal design, bad external design, etc. We also list what phase of the development that the error was introduced. We're very interested in knowing that.

Another measure we use is a user report card. We distribute the user report cards at our users group meeting about twice a year. We have our external users send report cards back to us. They essentially evaluate us on areas of useability, functionality supportability, performance and reliability, and give us a grade. We look at that data and see how we can make improvements. The field is probably one of our best critics. The report cards are great in telling us now we are doing against our competitors, how our customers feel about our software and the amount of integration our field force goes through sometimes to put a software system to work, especially on the HP 3000 where we offer a wide variety of software products. Sometimes the field has to go through some gyrations they would rather not go through to put together a customized software system for a customer. And then we also get suggestions, how can we improve?

Our own support group internal to the factory gives us feedback on hot sites. Hot sites means, basically a site that has run into a critical problem where the factory needs to get involved to solve that problem. They also can tell us how we improve our training. The lab reviews each other. We have quarterly reviews and we evaluate each other across lab areas. That is a real good way for peers to evaluate each other. Also when they're doing maintenance they determine how easy a product was to use. These are all costs. I'm not saying that at this time we are tracking all these costs, but these are all costs we could track. We do sometimes track things by number.

There is an associated average cost---a phone call for instance. We have an average cost of what a typical on line support phone call costs. We keep track by product of what those phone calls are. We track memos written to the field or to other areas inside the company. A commgram is a notice you basically put out to the field when you've found a problem. It's like a broadcast, mailgram to everybody, we know what that costs. Patches, to software products and visits to site, those things are pretty well defineable. Training requirements are tracked too. Typically we put software in a test site. We have two levels of test sites. One is called an Alpha test site which is internal to HP. The second is a Beta test site which is external to HP. There are training costs associated with each type of test site. We know what it costs to keep STARS running. The field charges back things like

training specialists, phone calls, letters and things like that. We are also looking at tracking things inside of STARS and associating a cost for those.

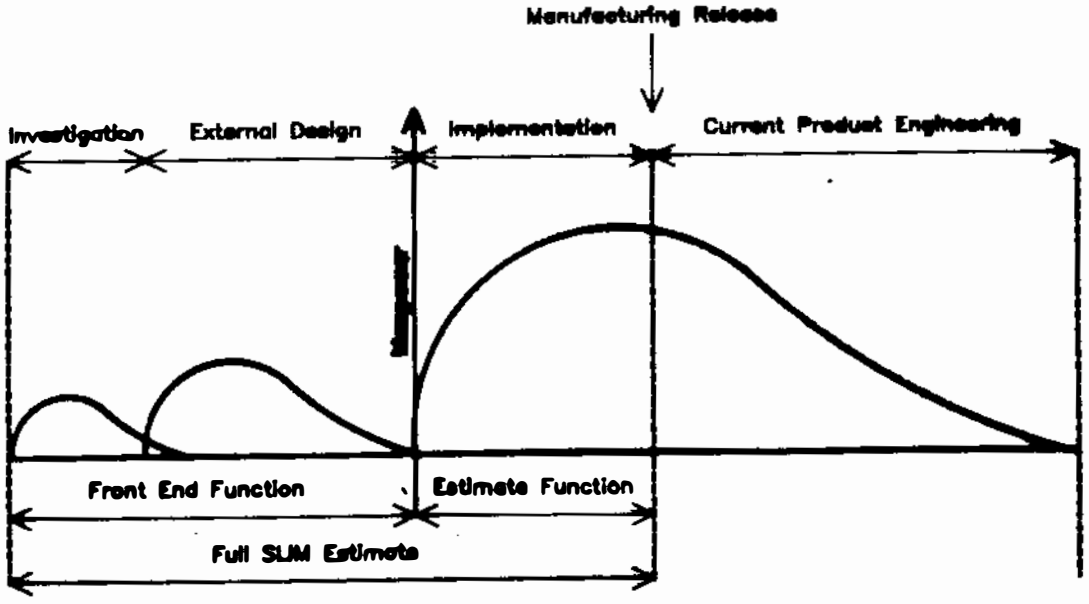
I wanted to define some terminology here so we can discuss cost of quality later on. If you talk about costs of quality I think you have to talk about costs in terms of what you're going to spend on prevention vs. what you are going to spend on appraisal vs what have you spent on something that has failed. What does a hot site cost you, and how could you avoid that hot site. What would you spend in prevention to avoid it?

A lifecycle should definitely define the roles and responsibilities of the members of our product teams. We all have a role to play and we know what our responsibilities are. Each functional area has those defined for each of the different phases. Definitely you have to have check points and associated expectations. What I mean is that when you start a project you set your objectives and a lot of times those objectives will conflict. For instance, if one of your objectives is to create a very maintainable piece of software, that may conflict with it's performance requirements. You need to priortize those objectives so you do that up front, and then you do that for each phase in the lifecycle. Then you can measure whether you've met your objectives and know whether you've completed that milestone. The way we do that is through the quality plan. We have a set of objectives, we say what our measures are, and then how we are going to verify them. That's all that document provides, but it provides it up front in the development cycle and it provides it for each phase. We find check lists very handy for quick reviews. We hold weekly reviews for each project team and typically go over the check list of things that have to be done and make sure that people have completed their action items. The quality plan provides identification authorization, that is, who has sign off capability to answer some questions and permit the team to go to the next phase. Three people have to sign off the checklist before the project proceeds to the next phase and quality assurance. They need those three signatures. Like I said we don't like to get to the point where there's a big fight over not signing and hopefully our process keeps that from happening.

Other things the quality plan can optionally provide is an organizational philosophy, how you measure progress, how you know progress has been made, cultural attributes, etc.

One of the things we found we had a real problem with was schedules and costing of the project. What happens when that becomes a problem is you typically cut out some things. If your goal is to get the product to the market at a certain time but you underestimated the resources required or you underestimated the schedule, you get near the end of the cycle into the testing phase or documentation phase and decide to cut some activities out. We discovered we had some unrealistic schedules so we decided to do something about it.

Software Life Cycle

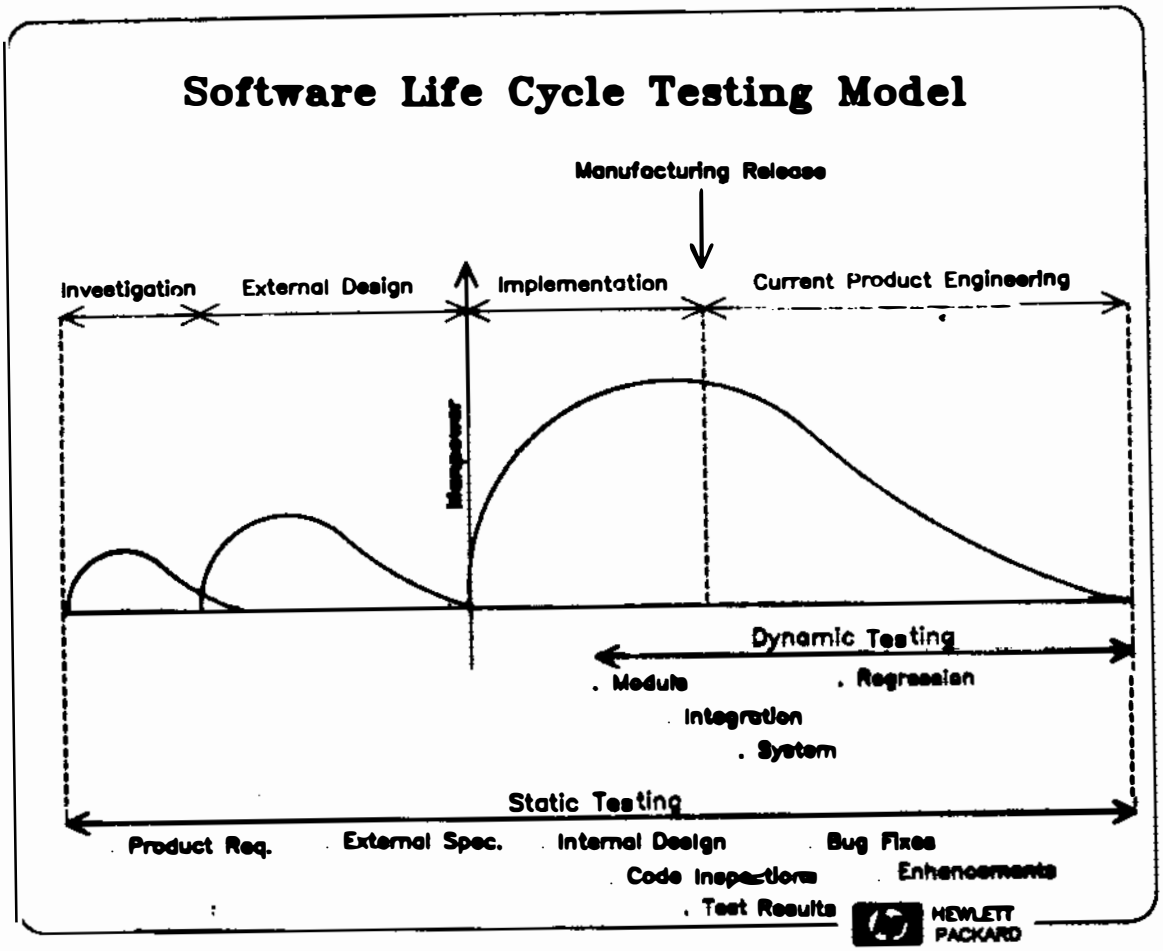


We started using a tool called Slim for estimating software costs. I guess the point I am trying to make is that you need to use some kind of estimation algorithm. It shouldn't be the only thing you use to figure out project schedules, but it's very important that you do a good job estimating schedules. If you don't I don't care how much committment you have to quality and costing, you'll be forced to get the product out in some reasonable time frame. You're never going to be able to do that if you don't have good estimates.

The point here is to put up the phase of the lifecycle showing that there is a manufacturing release point and that all the area under each of these curves represents the effort required during the entire life of the product. After manufacturing release there is a lot of effort left underneath the curve which we call current product engineering. It's kind of a nice way to get engineers to do maintenance.

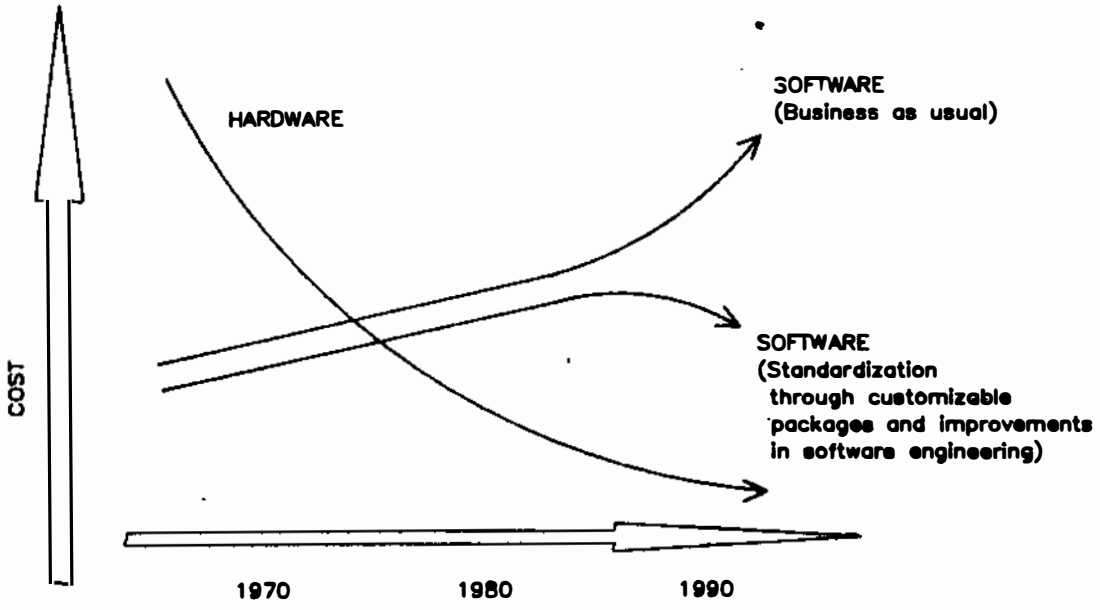
Those are all costs that are estimated.

We do a similar thing for testing. We have a testing model of the lifecycle. We test our product against the requirements. Sometimes we test them against actual customers and sometimes we test them against internal users. We find that we have a lot of users of our products inside the company so we have them test the product. That way we don't run into new product disclosure problems. We test internal specs against the internal design. The visual, dynamic testing is done on the integrated system level. We have regression testing for each successive release of the software.



The economics for software is changing. In the next ten years we're going to see some profound changes. If we continue to move along our current path, software is going to cost us way too much to provide. A lot of companies aren't going to be able to afford the investment that it's going to take to build software products. I think that there's a lot of things that can be done to reduce the cost and improve the quality of the software.

THE CHANGING ECONOMICS OF INFORMATION SYSTEMS



- Continuing improvements in hardware price/performance
- Inflation (labor) cost and productivity (need for improvement) drive software costs

At this point the group began a workshop and suggested the following ways to reduce or measure the cost of software quality.

1. Formally and rigorously track functional requirements all the way through the design process.

2. Use rapid prototyping to test the user interface and functionality before investing too much development money.

3. Use the 3 party approach ie. Have the following 3 parties reach consensus that a product is ready for release:
   a) design team developing the product.
   b) Someone (QA or Marketing) representing user interests
   c) V and V ((verification and validation)

4. Determine the minimum acceptable feature set early and write the user manual at that time.

5. Find errors as early as possible by using design reviews, code reviews and code walkthroughs.

6. Track development costs vs maintenance costs.

7. Track the performance of the people who manage and develop new products. Give credit to those who produce good quality products and penalize those who don't.

8. Have development engineers spend internship time in QA. Former QA people always produce better specifications for their products.

9. Provide documentation support tools for the specs, code and tests.

10. Carefully archive software, documentation, tests and tools.

11. Gather information on and validate one metric during the coming year. Organize your entire shop to take advantage of the metric. That would speed up the guidebook considerably.

12. Provide adequate training for managers, engineers and the field. Use job rotation as part of the training.

BIOGRAPHY: Paul Nichols spent eight years in the banking industry as a programmer/analyst and systems programmer. Typical reponsibilities included installation and support of new operating system releases and special projects, such as setting up a Computer Aided Data Entry (CADE) system for data entry clerks and an automated teller network.

Since joining HP four years ago, he has worked in Software Quality Assurance including experience with the HP 3000 and HP 300. During the last three years he has been a reliability engineering manager for software QA support of business application tools and office products on the HP 3000.