FIFTH ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

October 19-20, 1987

Sheraton Inn - Portland Airport
Portland, Oregon

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

WELCOME AND INTRODUCTORY REMARKS

I am very happy to welcome you to the fifth annual Pacific Northwest Software Quality Conference. I have actively participated in the Conference since its second year, and I have watched it grow and become a more significant event each year.

The primary purpose of this conference is to help you in your career as a software professional. Help you by presenting new ideas that you can adapt to your own work. Help you by providing technical workshops where you can acquire new skills. Help you by providing access to vendors with useful tools. And help you by providing a forum where you can meet fellow software professionals and share ideas.

I sometimes think back to our first conference. At that time it was not called the first annual conference because neither of the two organizers knew if it would be held on an annual basis. In fact, when they first discussed the concept, they were mentally prepared for the possibility that the only people in the audience might be themselves and the people giving the presentations. As it happened, it was very successful and that conference marked the beginning of something new and important to you, the software professionals of the Pacific Northwest.

Each year since then has been marked by some important new initiative. Last year we added workshops and vendor exhibits. This year, for the first time, we sent out a national call for papers. We were rewarded with a large number of high-quality papers. So many in fact, that we felt compelled to increase the number of parallel tracks from 2 to 3, and increase the number of presentations from 15 last year to 26 this year. In a similar vein, we have added a fourth workshop. I think you will find that the quality as well as the quantity has increased.

The theme of this year's conference is Effective Software Practices. This is a broad umbrella, and the three areas we have chosen to focus on are Project Management, Reusability, and Meeting Customer Needs.

On behalf of the organizing committee, I sincerely hope that you enjoy the conference and that you come away a more competent software professional.

Steve Shellans
Conference Chair

Conference Committee

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Steve Shellans - President and Chairman; *Tektronix, Inc.*

Dale Mosby - Technical Program; *Sequent Computer Systems, Inc.*

Paul Blattner - Workshops; *Quality Software Engineering*

Bill Edmark - Exhibits; *Intel Corporation*

Sue Strater - Vice-President, Public Relations; *Mentor Graphics Corporation*

Sue Bartlett - Secretary; *Engineering Software Products*

Dennis Schnabel - Treasurer; *Mentor Graphics Corporation*

CONFERENCE PLANNING COMMITTEE

Sandy Baldrige, *Sequent Computer Systems, Inc.*

Dave Dickmann, *Hewlett-Packard*

Ron Edwards, *ELDEC Corporation*

Richard Hamlet, *Oregon Graduate Center*

Elicia Harrell, *Intel Corporation*

G. W. Hicks, *Test System Strategies*

Monika Hunscher, *Floating Point Systems*

Arun Jagota, *Intel Corporation*

Bill Junk, *University of Idaho*

Marne McIvor, *Tektronix, Inc.*

Kent Mehrer, *Mentor Graphics Corporation*

Kenneth Oar, *Hewlett-Packard*

Craig Thomas, *Mentor Graphics Corporation*

Nancy Winston, *Intel Corporation*

PROFESSIONAL SUPPORT

Lawrence & Craig, Inc. - *Conference Management*

Liz Kingslien - *Graphic Design*

Decorators West, Inc. - *Decorator for Exhibits*

Authors

Richard Anderson
Pacific Bell
2600 Camino Ramon, #2N750
San Ramon CA 94583

Barbara Beccue
Applied Computer Science Department
Illinois State University
Normal IL 61761

Ronald Benton
Hewlett-Packard
Lake Stevens Instrument Division
Everett WA 98205-1298

Johnny M. Brown
Schlumberger Well Services
P.O. Box 200015
Austin TX 78720

Robert Cameron
Simon Fraser University
Burnaby BC V5A 1S6
CANADA

John Cherniavsky
Computer Science Department
Georgetown University
Washington DC 20057

Robert Dea
Hewlett-Packard
3500 Deer Creek Road, Bldg. 26U
Palo Alto CA 94304

J. P. Delatore
AT&T Bell Labs
200 Park Plaza, 1HP 1F-508
Naperville IL 60566-7050

Frank Hall
Hewlett-Packard
1000 NE Circle Blvd.
Corvallis OR 97380

Mark A. Johnson
Mentor Graphics Corporation
8500 SW Creekside Place
Beaverton OR 97005-7191

Peter Lempp
Software Products & Services
14 E. 38th Street
New York NY 10016

Dick LeVitt
Hewlett-Packard
8000 Foothills Blvd.
Roseville CA 95678

Patrick H. Loy
Whiting School of Engineering
The Johns Hopkins University
Baltimore MD 21218

Jerry Luengen
J. Luengen & Associates
8000-D NE 58th Avenue
Vancouver WA 98665

Jennifer Marden
Hewlett-Packard DSD
1266 Kifer Road, MS 101N
Sunnyvale CA 94086

Chuck Martiny
Tektronix, Inc.
P.O. Box 500, MS 19-245
Beaverton OR 97077

Millard J. McQuaid
AT&T Info. Sys. Labs
11900 N. Pecos, #1d09
Denver CO 80234

Larry J. Morell
College of William and Mary
Department of Computer Sciences
Williamsburg VA 23185

Thomas J. Ostrand
Siemens Research Labs
105 College Road E.
Princeton NJ 08540

S. G. Perelgut
Holt Software Associates, Inc.
203 College Street
Toronto ON M5T 1P9
CANADA

Authors (continued)

Debra Richardson
Computer Science Department
University of California
Irvine CA 92717

Alan Snyder
Hewlett-Packard
1501 Page Mill Road
Palo Alto CA 94304

Cathryn Stanford
Hewlett-Packard
Lake Stevens Instrument Division
Everett WA 98205-1298

Ambrish Vashishta
Department of Computer Science
Wayne State University
Detroit MI 48202

John P. Walter
1240 Seventeenth Street
Hermosa Beach CA 90254-3304

Elaine Weyuker
New York University
251 Mercer Street
New York NY 10012

Nancy Willard
Attorney at Law
296 E. 5th Avenue, #302
Eugene OR 97401

Exhibitors

Expertware, Inc.
Contact: Gary W. Furr
3235 Kifer Road, #220
Santa Clara CA 95051-0804
408/746-0706

IDE, Inc.
Contact: Irene Kazakova
150 4th Street
San Francisco CA 94103
415/543-0900

Index Technology
Contact: Elaine Cincotta
One Main Street
Cambridge MA 02142
202/494-8200

Informix Software, Inc.
Contact: Pattee Singer
4100 Bohannon Drive
Menlo Park CA 94025
415/322-4100

KnowledgeWare, Inc.
Contact: Diana Felde
3340 Peachtree Road NE, #2900
Atlanta GA 30026
404/231-8575

PEI, Inc.
Contact: Bob Poston
4043 Highway 33
Pinton Falls NJ 07753
201/918-0110

Promod, Inc.
Contact: Thomas L. Scott
23685 Birtcher Drive
Lake Forest CA 92630
714/855-3046

Softstar Systems
Contact: Dan Ligett
28 Ponemah Road
Amherst NH 03031
603/672-0487

Software Research, Inc.
Contact: Edward Miller
625 Third Street
San Francisco CA 94107-1997
415/957-1441

Tektronix, Inc.
Contact: Marty Boyesen
P.O. Box 4600, MS 92-635
Beaverton OR 97076
503/645-6464

Keynote

THE JAPANESE VIEW OF SOFTWARE QUALITY

Donald J. Reifer
Consultant

The Japanese have a national goal of becoming a leader in the information technology industries. They believe that software quality is of strategic importance in achieving this goal. We identify the approaches the Japanese use to engineer quality into their software products, analyze the Japanese approach in order to highlight its strengths and weaknesses, and recommend ways in which American firms can take advantage of the Japanese work in the context of the American way of doing business.

Biography

Donald J. Reifer is an internationally recognized expert in the fields of software engineering and management, with experience in both industry and government. Mr. Reifer has successfully managed major projects, served on source selections, written winning proposals, and led project recovery teams. While with TRW, Mr. Reifer managed an 800-person avionics organization and was the Deputy Program Manager for Global Positioning Satellite (GPS) system projects. While with the Aerospace Corporation, Mr. Reifer managed over $800 million in software deliverables for the Space Transportation System (Space Shuttle). These efforts involved managing the development of over 20 million instructions (many reused) produced by 18 contractor teams. Currently, as President of his own consulting firm, Mr. Reifer directs efforts aimed at helping many Fortune 500 companies and government agencies get their arms around software.

Mr. Reifer is author of over 100 papers on software engineering and management. He holds a BSEE from Newark College of Engineering, an MSOR from the University of Southern California, an MBA from UCLA, and two honorary PhDs. He has been a member of the Air Force Scientific Advisory Board, the Space Station startup team, and the NASA Director's Select Committee on Software Management. He serves as an advisor to the Export Control Administration, the General Accounting Office, the Nuclear Regulatory Commission, and the U.S. Intelligence Board. His many honors include being listed in *Who's Who in the West*, and receiving the NASA Distinguished Service Medal and the Hughes Aircraft Company Fellowship.

Mr. Reifer is involved professionally, being a member of the ACM, IEEE, ISPA, NSIA and SSCAG. As a member of the NSIA's Software Committee, he helped prepare their 1987 position paper on large-scale software project management. He is also active in the ASQC, being on their software section's Executive Committee and Chairman of their Strategic Planning Subcommittee. He is an associate editor of *The Journal of Systems and Software* and a contributing editor to a number of trade publications.

THE JAPANESE VIEW OF SOFTWARE QUALITY

19 OCTOBER 1987

Reifer Consultants, Inc.

25550 Hawthorne Boulevard, Suite 208/Torrance, California 90505

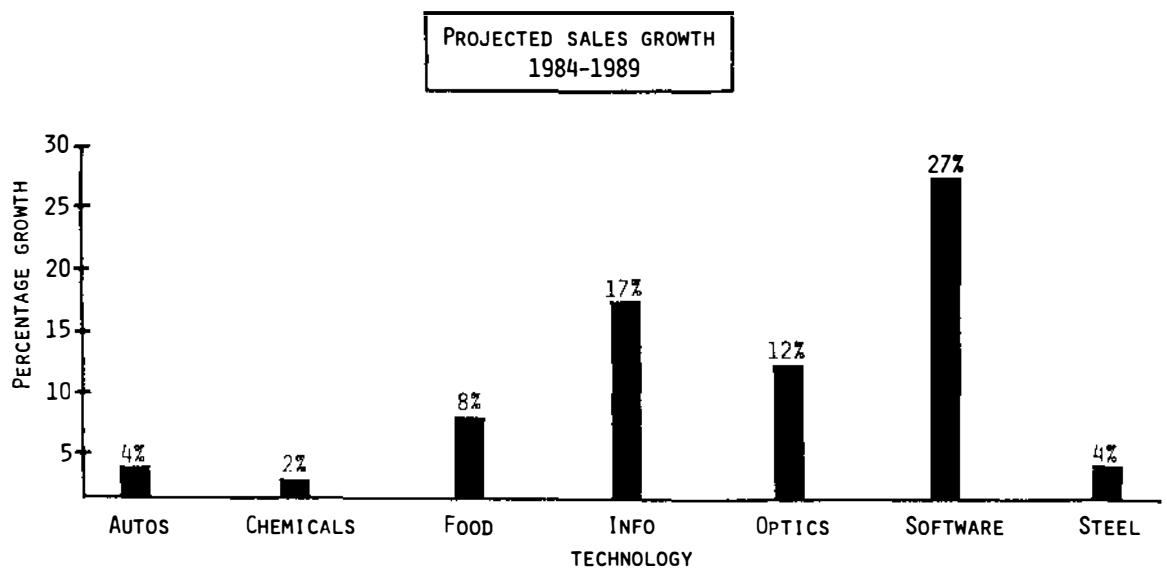
PURPOSE OF BRIEFING

- To describe why the Japanese believe that software quality is of strategic importance
- To identify the approaches the Japanese use to engineer quality into their software products
- To analyze the Japanese approach competitively to determine its strengths and weaknesses
- To recommend things that U.S. firms can do to take advantage of the Japanese work in the context of the American way of doing business

JAPANESE COMPUTER STRATEGY

- ESTABLISHMENT OF A NATIONAL GOAL TO BECOME A LEADER IN THE INFORMATION TECHNOLOGY INDUSTRIES
- CREATION OF STRATEGIES AND A NATIONAL ORGANIZATION DEDICATED TO ACHIEVING THE GOAL
  - JSDC AND VARIOUS GOVERNMENT LABORATORIES ESTABLISHED
- ACQUISITION OF THE PRODUCTION CAPACITY AND TECHNOLOGY TO REALIZE STRATEGIES
  - CREATION OF A NATIONAL SOFTWARE TECHNOLOGY PROGRAM
  - CREATION OF SOFTWARE FACTORIES AND STRATEGIC ALLIANCES WITH SOFTWARE HOUSES
  - SHARING OF TRADE SECRETS WITH IBM
  - SELECTION OF UNIX AS A NATIONAL STANDARD
- DEVELOPMENT OF MARKETING CHANNELS AIMED AT PENETRATING A WORLD-WIDE CUSTOMER BASE
  - DEVELOPMENT OF PARTNERSHIPS AND SUBSIDIARIES

SOFTWARE AS A GROWTH INDUSTRY

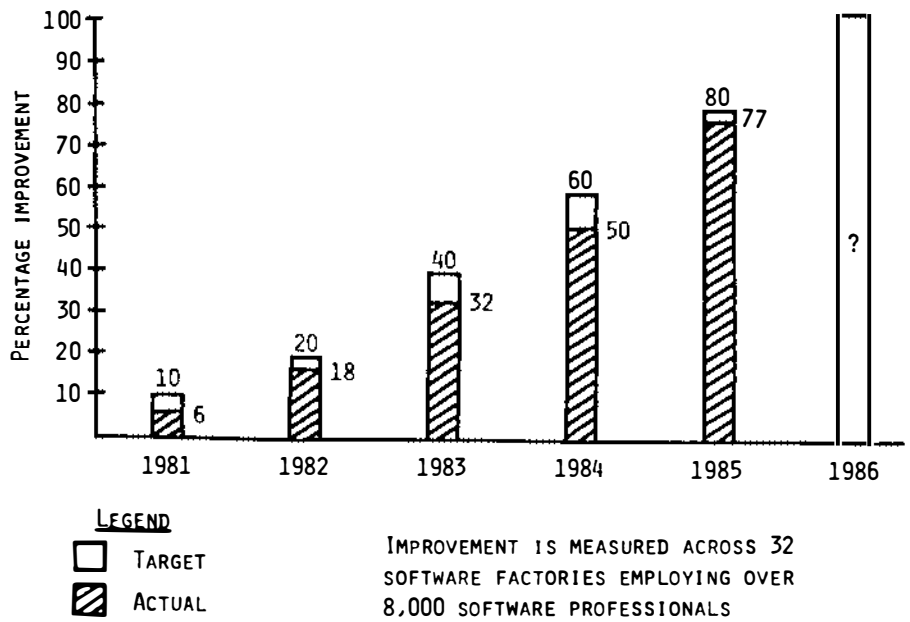


SOURCE: SOFTWARE ALERT, 04.86

UNDERLYING JAPANESE SOFTWARE STRATEGY - 1981

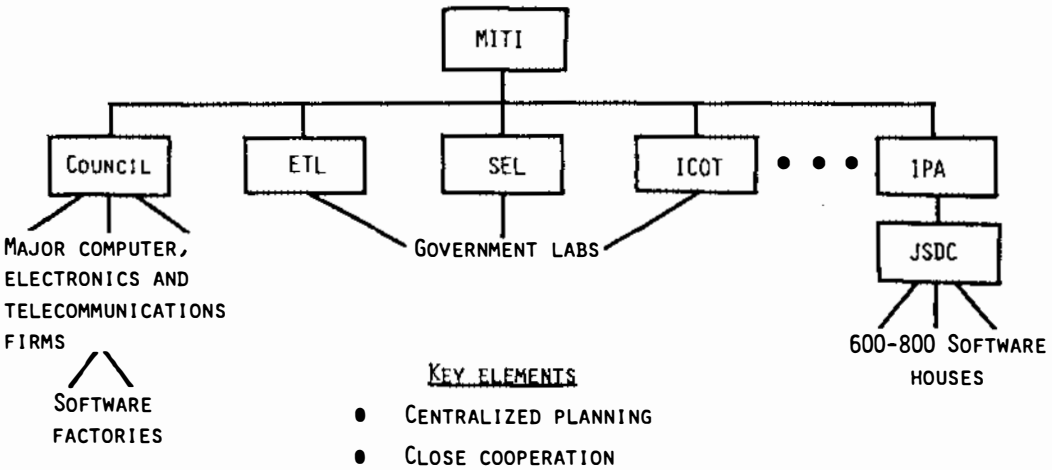
- SOFTWARE SELLS HARDWARE AND IS ITSELF A MULTI-BILLION DOLLAR INDUSTRY IN THE U.S. AND ABROAD
- HIGH TECHNOLOGY MARKETS ARE QUALITY-DRIVEN RATHER THAN PRICE-DRIVEN
- FOR MARKET ENTRY:
  1. ESTABLISH A REPUTATION FOR QUALITY
  2. PRICE SOFTWARE WITHIN 10% OF COMPETITION AND WARRANTY IT
  3. CREATE NICHE MARKETS TO CREATE FOOTHOLDS
  4. GRADUALLY EXPAND BASE
- CREATE STRATEGIC PLAN AND SET REASONABLE TIMEFRAME
- CREATE STRATEGIC ENTERPRISES AND TECHNOLOGY BASE TO ACHIEVE STRATEGIC PLAN

QUALITY IMPROVEMENTS TO DATE



IMPROVEMENT IS MEASURED ACROSS 32 SOFTWARE FACTORIES EMPLOYING OVER 8,000 SOFTWARE PROFESSIONALS

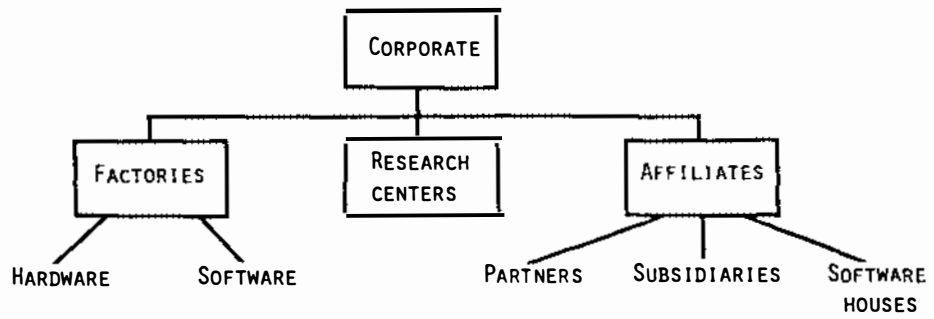
SOFTWARE INDUSTRY IN JAPAN



KEY ELEMENTS

- CENTRALIZED PLANNING
- CLOSE COOPERATION BETWEEN COMPETITORS
- STRATEGIC ALLIANCES WITH SOFTWARE HOUSES
- EMPHASIS ON QUALITY AND PRODUCTIVITY

JAPANESE COMPANY X



KEY ELEMENTS

- CENTRALIZED PLANNING
- CLOSE COOPERATION WITH MITI, GOVERNMENT AGENCIES AND UNIVERSITIES
- STRATEGIC ALLIANCES WITH SOFTWARE HOUSES

JAPANESE SOFTWARE FACTORIES

1. STRATEGIC CORPORATE ENTERPRISE
2. SEPARATE DIVISION PRODUCING SOFTWARE FOR VARIED USES
3. ORGANIZED LIKE A MANUFACTURING FACILITY
4. A PRECISE AND CONSISTENT METHODOLOGY FOR ALL
   - DEVELOPMENT
   - MAINTENANCE
5. AUTOMATED AND INTEGRATED SOFTWARE SUPPORT SYSTEMS
   - DEVELOPMENT
   - MANAGEMENT
6. MORE THAN 1,000 PEOPLE

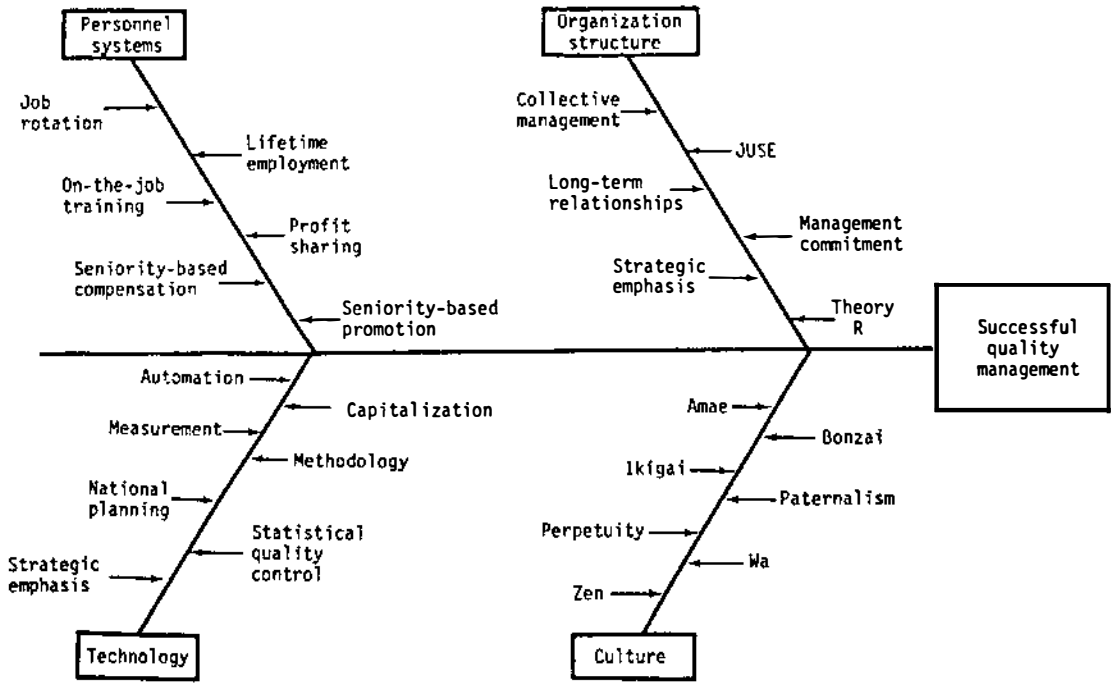
SOURCE: MCNAMARA, GE

JAPANESE FACTORY TACTICS

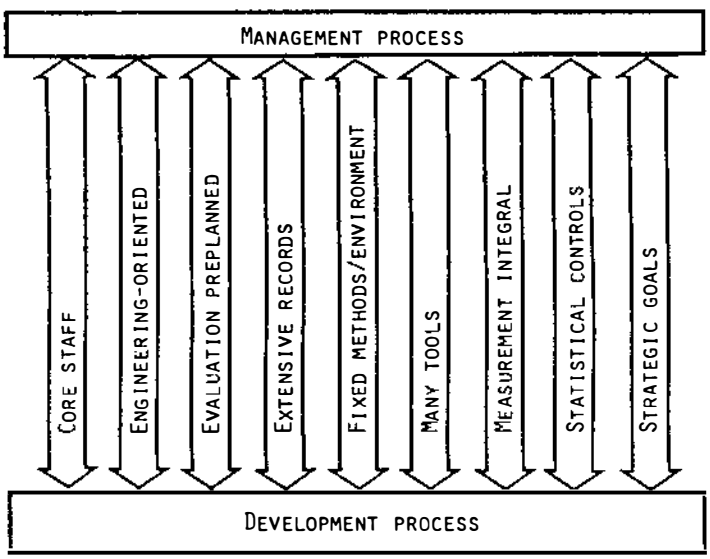
- TOP MANAGEMENT COMMITMENT
- FIXED CADRES/ORGANIZATIONS
- EMPHASIS ON TEAMWORK/PARTICIPATION
- FIXED DEVELOPMENT METHODS/ENVIRONMENT
- HIGHLY CAPITALIZED AND AUTOMATED
- FOCUSED ON SPECIFIC APPLICATIONS
- EMPHASIS ON PACKAGING AND REUSE
- QUALITY CONTROL AT EVERY STEP
- MEASUREMENT INTEGRAL TO THE PROCESS
- EMPHASIS ON QUALITY AND PRODUCTIVITY

SOURCE: MCNAMARA, GE

QUALITY SUCCESS FACTORS



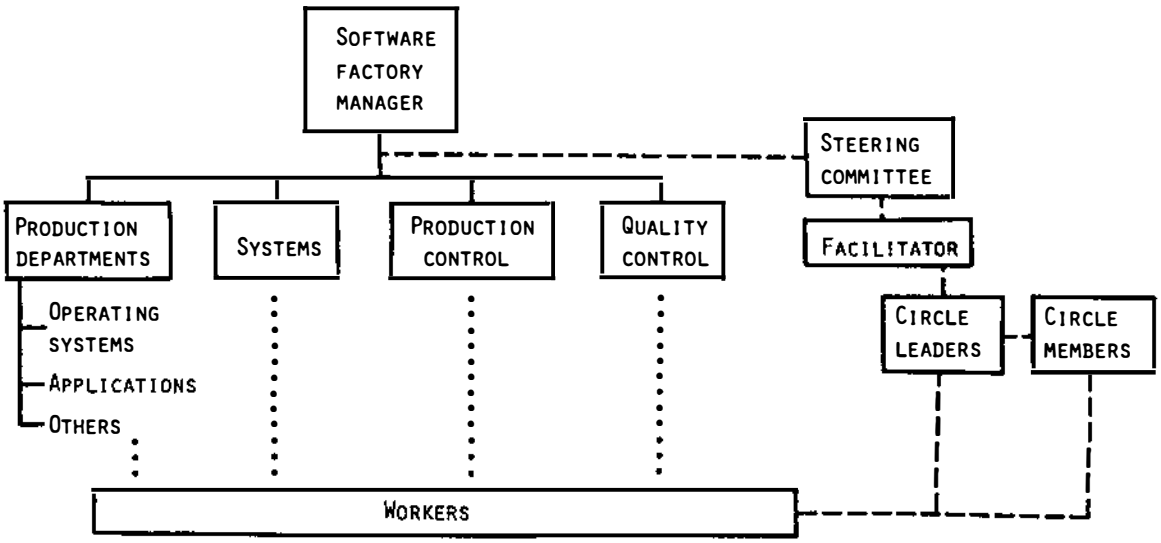
QUALITY CONTROL CONCEPTS



KEY ELEMENTS

- EVERONE PARTICIPATES
- ENGINEER-IN QUALITY
- REMOVE DEFECTS EARLY
- MEASURE DURING EVERY STEP

CORE STAFF



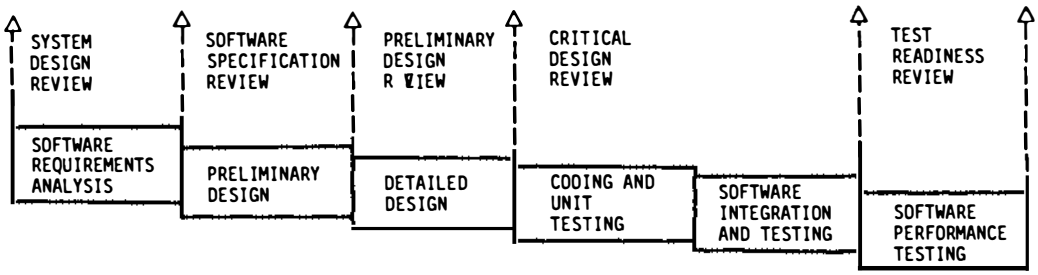
ENGINEERING - ORIENTED

USA		JAPAN
SQA AS A RUBBER STAMP - CHECKS COMPLIANCE WITH STANDARDS	1970	SQA AS A RUBBER STAMP - CHECKS COMPLIANCE WITH STANDARDS
SQA AS A TEST ACTIVITY - EVALUATES SOFTWARE QUALITATIVELY FOR "GOODNESS"	1975	SQA AS A LIFE CYCLE ACTIVITY - HELP ENGINEER QUALITY INTO PRODUCT
SQA AS A METRICS ACTIVITY - EVALUATES MEASURES OF QUALITY	1980	SQC AS A STATISTICAL ACTIVITY - EVALUATES QUALITY AND USES FEEDBACK TO IMPROVE BOTH PROCESS AND PRODUCT
SQA AS A LIFE CYCLE ACTIVITY - HELP ENGINEER QUALITY INTO PRODUCT	1985	SQC AS A SELF-METRICS ACTIVITY - INSTRUMENT ENVIRONMENTS SO PERFORMERS CAN EVALUATE OWN QUALITY

ENGINEERING - THE ART OR SCIENCE OF MAKING PRACTICAL APPLICATION OF KNOWN SCIENTIFIC KNOWLEDGE

EVALUATION PREPLANNED

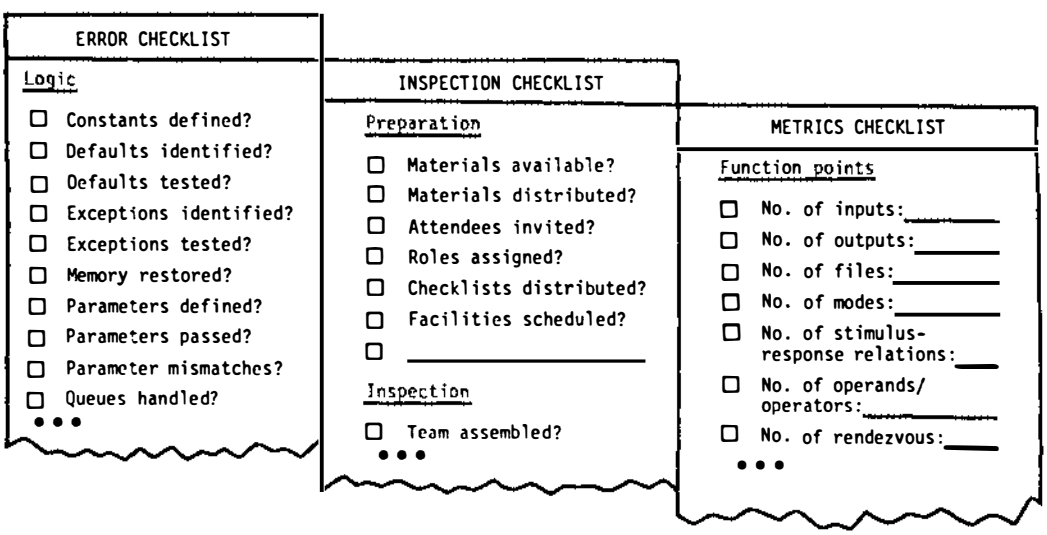
- Sets quality goals and establishes criteria to measure success
- Establishes roles and responsibilities
- Creates the tactical plan



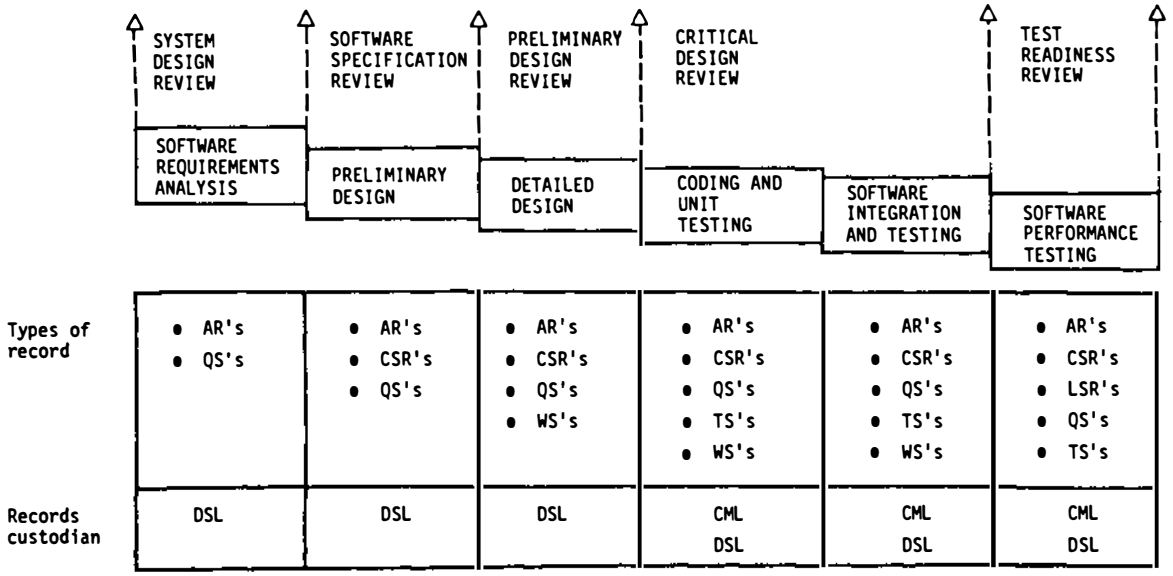
EXAMPLES:

Methods	• SA • SREM	• OOD • SD	• CPD	• SP • SV	• THREADS	• THREADS
Models	• DFD • R-NET	• MFD • SC	• FC	• Control graph	• C-and-E graph	• Trace matrix
Metrics	• Stability rate	• Fan-in • Fan-out	• Cyclomatic complexity	• Coverage	• Error rate	• Retest rate
Standards	60%	5	6	90%	10	5

WIDESPREAD USE OF CHECKLISTS

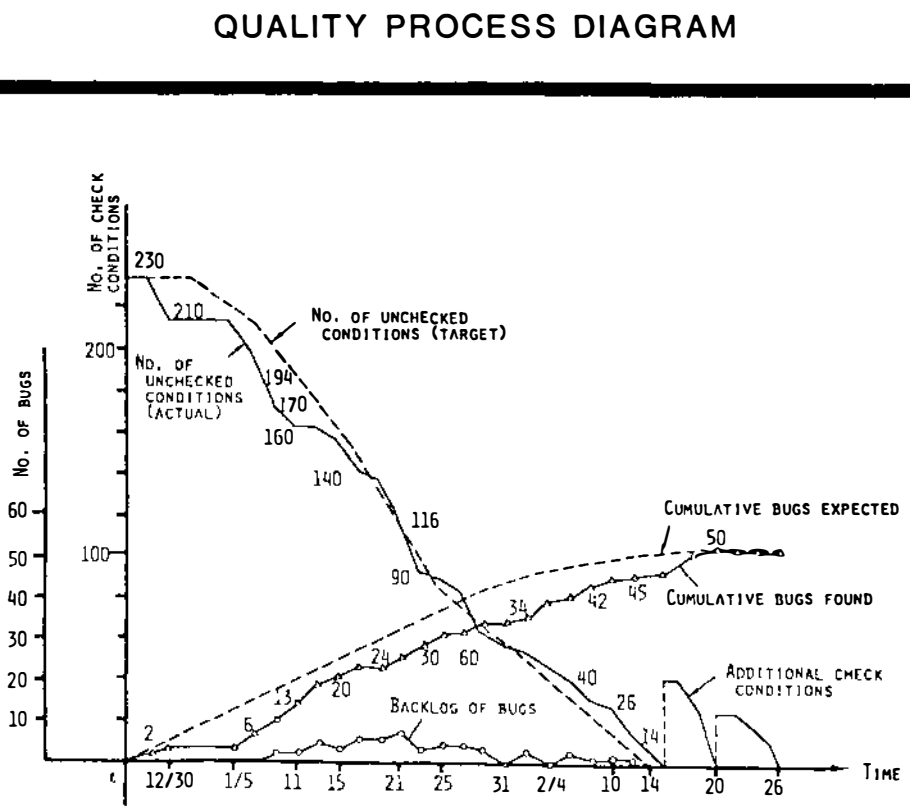


EXTENSIVE RECORDS



AR - Anomaly Reports
CML - CM Library
DSL - Development Support Library
CSR - Change Status Reports
LSR - Lien Status Reports
QS - Quality Summaries
TS - Test Summaries
WS - Walkthrough Summaries

QUALITY PROCESS DIAGRAM

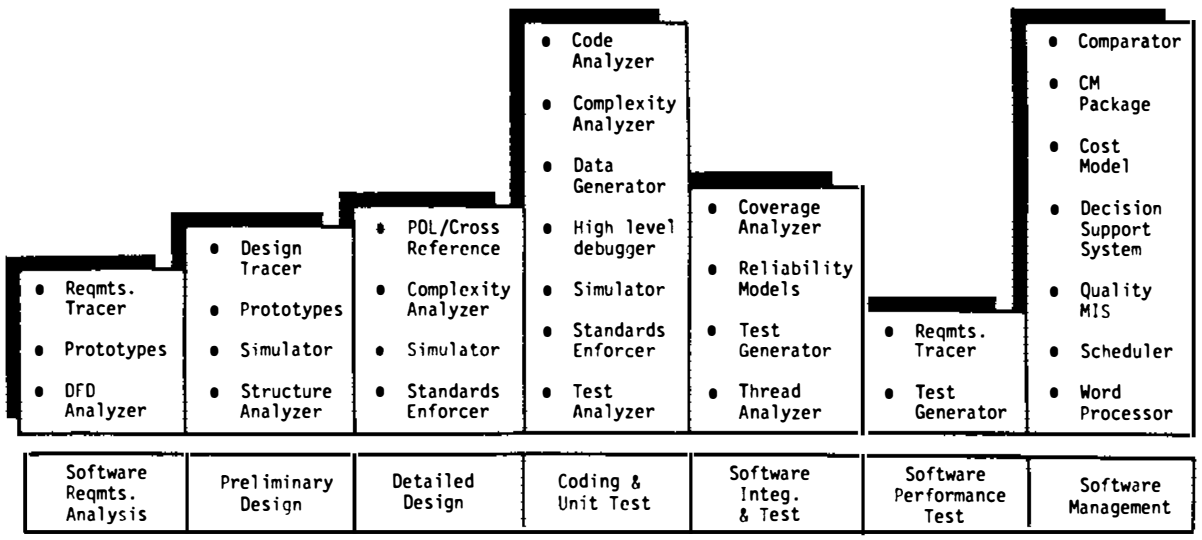


FIXED METHODS/ENVIRONMENT

Life cycle (phase)	Needs Analysis	System Planning	System Design	Software Design	Programming	Test	Operations & Maintenance
Development policy	• Establishment of integrated development method of application systems • Improvement of productivity, reliability, maintainability and manageability						
Methodologies	Management technique of system development (project management/integrated development steps and documents)						
	Problem investigation and ordering technique		Structured description technique		Structured programming technique	Systematic testing technique	Editing of documents
	Structuring of problems		Structuring of system requirement specifications	Structuring of software modules		Test case design Sufficiency of test	Japanese diagrams
Tools and languages	• PPDS-HDS		• RDL/ROA • GDSS	• DFDS • ADDS • POL-PAD	• HPL • SPL	• AGENT	• Various
		• PDSS-FIT	• DBDS				
Environment	• Common command language • UNIX-based			• Workstation hosted • AI front-end			

MANY TOOLS

Types and relative proportion of tools used by Quality Control across life cycle phases and activities

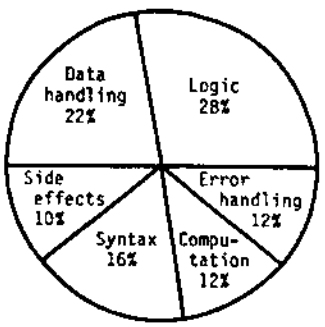


SPECIALIZED QUALITY TOOLS

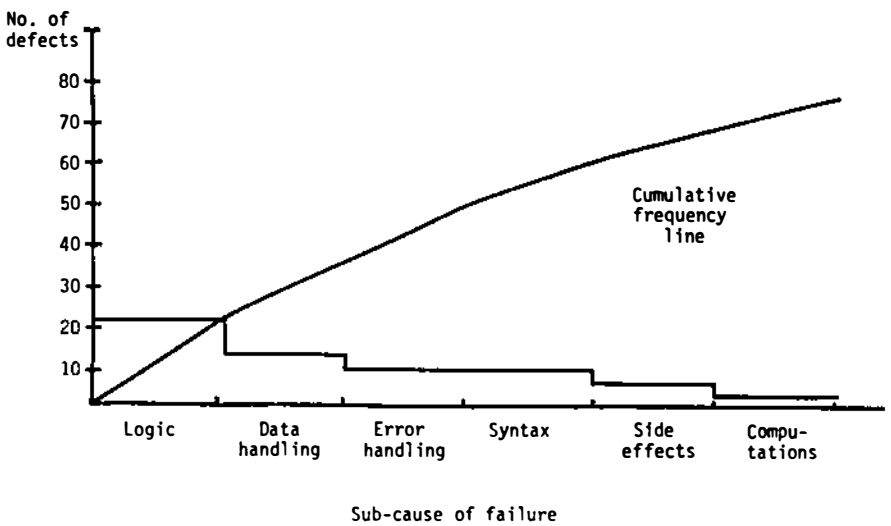
- BRAINSTORMING SESSIONS
- CAUSE AND EFFECT GRAPHS
- CHECKLISTS
- ERROR LISTS/HISTOGRAMS
- FISHBONE DIAGRAMS
- PARETO DIAGRAMS
- QUALITY CIRCLES
- QUALITY METRICS
- STATISTICAL/TREND CHARTS

PARETO ANALYSIS (70/30 RULE)

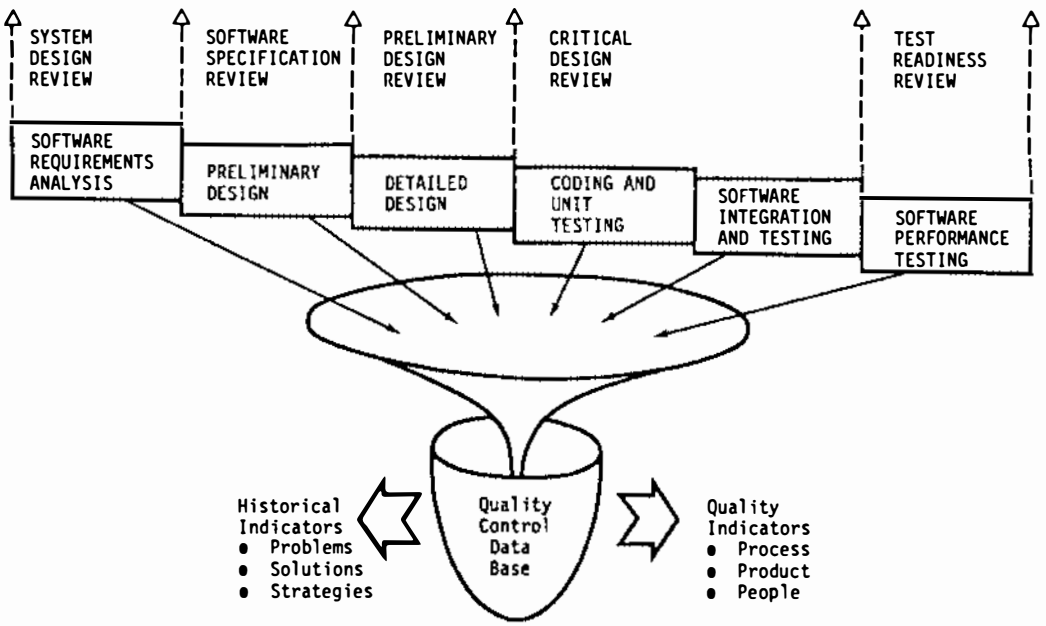
Defect circle graph



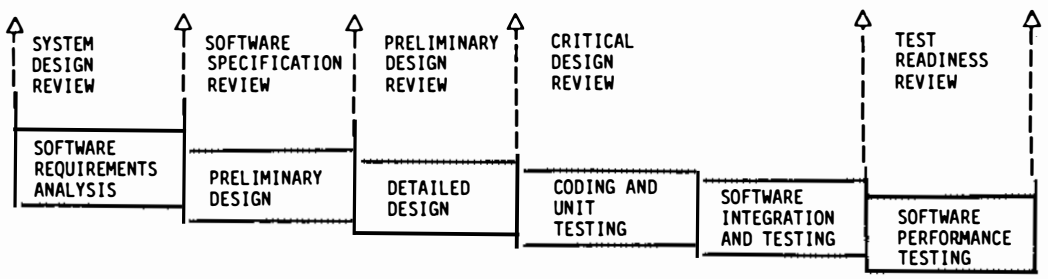
	No. of defects	Cumulative count
Logic	21	21
Data handling	15	36
Error handling	12	48
Syntax	12	60
Side effects	9	69
Computation	6	75



MEASUREMENT INTEGRAL

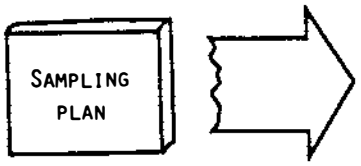


EXAMPLE QUALITY INDICATORS



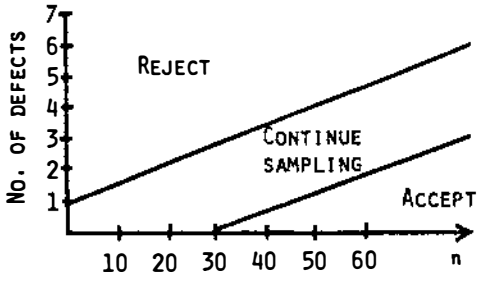
Types of Indicators						Types of Reports
• Process • Product • People	• Function point count • Information entropy • Reqmts. stability • Rework rate • Size • Staffing rate	• Defect rate • Fan-in and fan-out • Module density • Reqmts. stability • Rework rate • Size	• Defect rate • Design complexity • Growth rate • Sample rate • Rework rate • Turnover rate	• Cyclomatic complexity • Defect rate • Growth rate • Sample rate • Rework rate • Turnover rate	• Change rate • Defect rate • Coverage • Retest rate • Rework rate • Test success ratio	• Acceptance rate • Change rate • Defect rate • Retest rate • Rework rate • Test success ratio	• Actuals vs. predicted • Actuals vs. standards • Trend lines

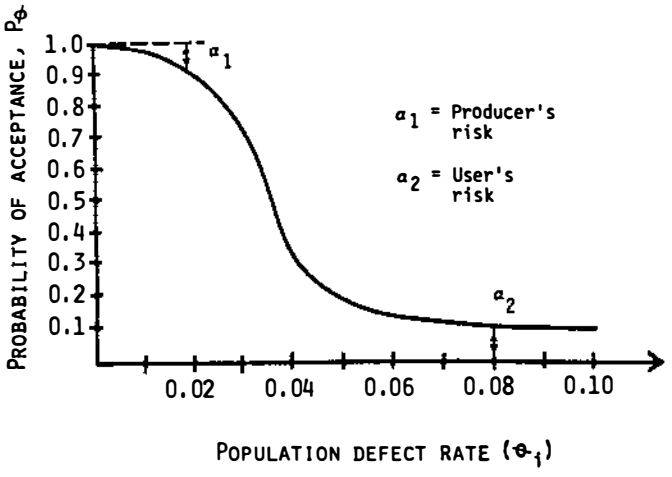
STATISTICAL CONTROLS



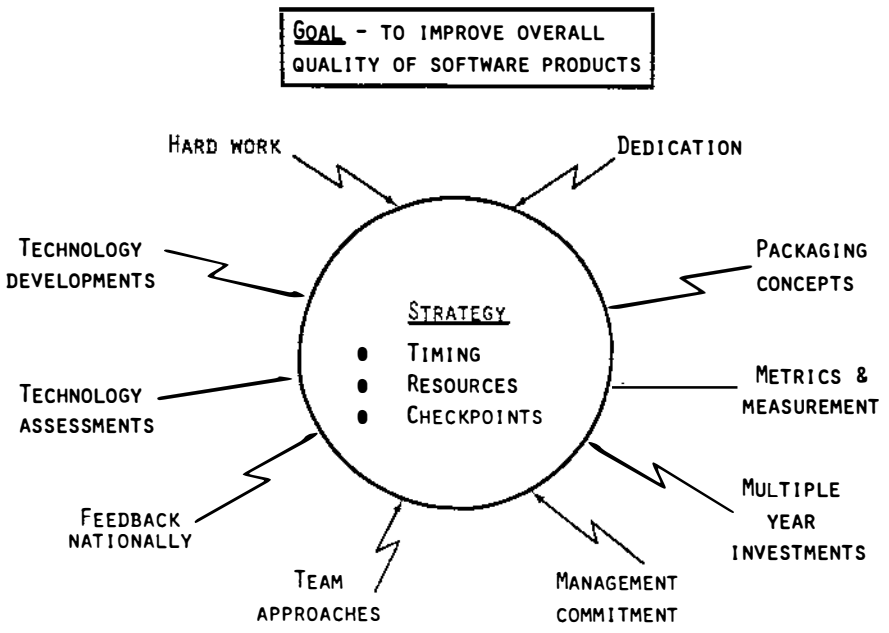
PRODUCER'S RISK = A GOOD LOT CAN BE REJECTED BY THE SAMPLING PLAN

USER'S RISK = A BAD LOT CAN BE ACCEPTED BY THE SAMPLING PLAN





STRATEGIC GOALS



STRENGTHS AND WEAKNESSES

	U.S.	JAPAN
STRENGTHS	• EMPHASIS ON MARKETING • INDIVIDUALISTIC • INVENTIVE • STRONG LEADER-SHIP	• INNOVATIVE • OPEN TO CHANGE • PATERNALISTIC • STRATEGIC • TEAM-ORIENTED
WEAKNESSES	• ADVERSARIAL • CONFLICT-ORIENTED • INDIVIDUALISTIC • RESISTANT TO CHANGE • TACTICAL	• EMPHASIS ON PRODUCTION • GROUPISM • MOBILITY LIMITATIONS • RIGIDITY OF STRUCTURE

WHAT CAN WE LEARN FROM JAPAN?

- BE MORE STRATEGIC IN OUR ORIENTATION
  - FUND IMPROVEMENTS USING MULTI-YEAR PLANS ACROSS MULTIPLE PROJECTS
- MAKE BETTER USE OF THE TECHNOLOGY BASE
  - DEVELOP AND USE QUALITY INDICATORS
  - MAKE OUR ENVIRONMENTS SELF-METRIC
- MEASURE AND CONTROL PROGRESS IN THE AREA OF QUALITY
  - STRESS ENGINEERING IT INTO THE PRODUCT
- BUILD ON OUR STRENGTHS AND COMPENSATE FOR OUR WEAKNESSES
- CHANGES OUR LAISSEZ FAIRE ATTITUDE AND BECOME AGGRESSIVE IN OUR PURSUIT OF EXCELLENCE

ACHIEVING GOOD SOFTWARE QUALITY

ENGINEER-IN QUALITY	REVIEW-OUT DEFECTS	MAKE-EVIDENT QUALITY
• COMMITMENT OF ALL CONCERNED	• SELF INSPECTIONS	• EXTENSIVE RECORDS
• MODERN SOFTWARE ENGINEERING METHODS/PRACTICES	• PEER REVIEWS - WALKTHROUGHS - WALKBACKS - INSPECTIONS	• PROCESS AND PRODUCT ORIENTATION
• PARTICIPATORY TEAM APPROACH	• FORMAL REVIEWS AND AUDITS	• PUBLISHED QUALITY INDICATOR REPORTS
• QUANTITATIVE QUALITY GOALS	• INDEPENDENT EVALUATION OF QUALITY	• QUALITY GATES
• QUALITY AS THE TEACHER PHILOSOPHY		• STATISTICAL QUALITY CONTROLS
		• WARRANTIES

FUTURE CHALLENGES

- HOW DO WE ENGINEER QUALITY INTO:
  - EXPERT SYSTEMS WHOSE KNOWLEDGE BASE AND/OR RULES ARE STOCHASTIC IN NATURE?
- HOW DOES QUALITY CONTROL INTERFACE WITH PROJECTS USING NEW PARADIGMS, METHODS AND ENVIRONMENTS?
  - SPIRAL AND INCREMENTAL BUILD MODELS OF DEVELOPMENT PROCESS
  - RAPID PROTOTYPING AND REUSABLE LIBRARIES
- WHAT MEASURES PROVIDE YOU WITH STATISTICALLY VALID MANAGEMENT INDICATORS OF QUALITY AS A FUNCTION OF LIFE CYCLE PHASE AND RISK?
- WHAT IS THE ACTUAL COST OF QUALITY AND HOW CAN IT BE AMORTIZED ACROSS MULTIPLE PROJECTS TO MAKE IT AFFORDABLE?

IN SUMMARY

- WE'VE DESCRIBED HOW THE JAPANESE ENGINEER QUALITY INTO THEIR SOFTWARE PRODUCTS
- WE'VE ANALYZED THE JAPANESE APPROACH COMPETITIVELY AND POINTED OUT ITS STRENGTHS AND WEAKNESSES
  - THEIR CULTURE, TEAM APPROACHES AND STRATEGIC ORIENTATION POSITION THEM WELL IN COMPARISON
  - THEIR LACK OF INVENTIVENESS AND INABILITY TO MAKE DECISIONS QUICKLY ARE RECOGNIZED HANDICAPS
- WE'VE RECOMMENDED THINGS THAT U.S. FIRMS CAN DO TO TAKE ADVANTAGE OF THE JAPANESE WORK IN THE CONTEXT OF OUR WAY OF DOING BUSINESS
- THERE IS A CHALLENGE BEING MOUNTED, BUT I DON'T THINK IT IS INSURMOUNTABLE IF WE APPLY OUR RESOURCEFULNESS

IN CONCLUSION

ALTHOUGH THERE IS NOTHING NEW, MAYBE THERE IS SOMETHING WE CAN LEARN FROM THE JAPANESE. WHAT DO YOU THINK?

Session 1

MANAGEMENT TOOLS

"How to Organize a Software Project"
Jerry Luengen, J. Luengen & Associates; and Ronald Swingen, Mentor Graphics

"Delivering on Commitments: Process Measures to Improve R&D Scheduling Accuracy"
Dick LeVitt, Hewlett-Packard

"Managing Software Projects Through a Metrics-Driven Lifecycle"
Cathryn Stanford and Ronald Benton, Hewlett-Packard

HOW TO ORGANIZE A SOFTWARE PROJECT

BY
J. LUENGEN & R. SWINGEN

ABSTRACT

This paper considers the portion of a software project between project inception and detailed planning. This span of time is very critical to project success. The key to a successful software project is project organization. A project should be organized in a systematic manner that can be adapted to any project: small or large. The concept of "organized" can be defined by a list of criteria which becomes a foundation for subsequent detailed planning and design activities.

The "organizing" tools and procedure, and how they should be applied, are discussed. The concept of a "quality hierarchy of definition" is introduced which is used to control this aspect of a software project. The procedural steps for efficiently organizing a project are shown, along with examples of a project manager's "toolkit" - a new efficiency tool. The components of project organization's primary deliverable, an initial Project Plan, are explained. Altogether, several new concepts, tools, and a systematic development approach to project organization make this a powerful aid to project success.

Author Biography

Jerry Luengen is a principal of J. Luengen & Assoc. located in Vancouver, WA. Mr. Luengen has 24 years of project management experience in a varietey of industries. He has held every position available in the field of project management. Currently, he is a Project Management Consultant, presenting project management seminars and an advisor to several Fortune 500 companies. He founded the Alaska Chapter of PMI and regurlarly presents project management workshops and papers at regional symposiums.

Ronald Swingen is a Senior Project Engineering Manager with Mentor Graphics Corporation in Beaverton, Oregon. Mr. Swingen has over 20 years experience in software engineering projects. He has held a variety of software positions with IBM, INTEL, and Tektronix before joining Mentor Graphics.

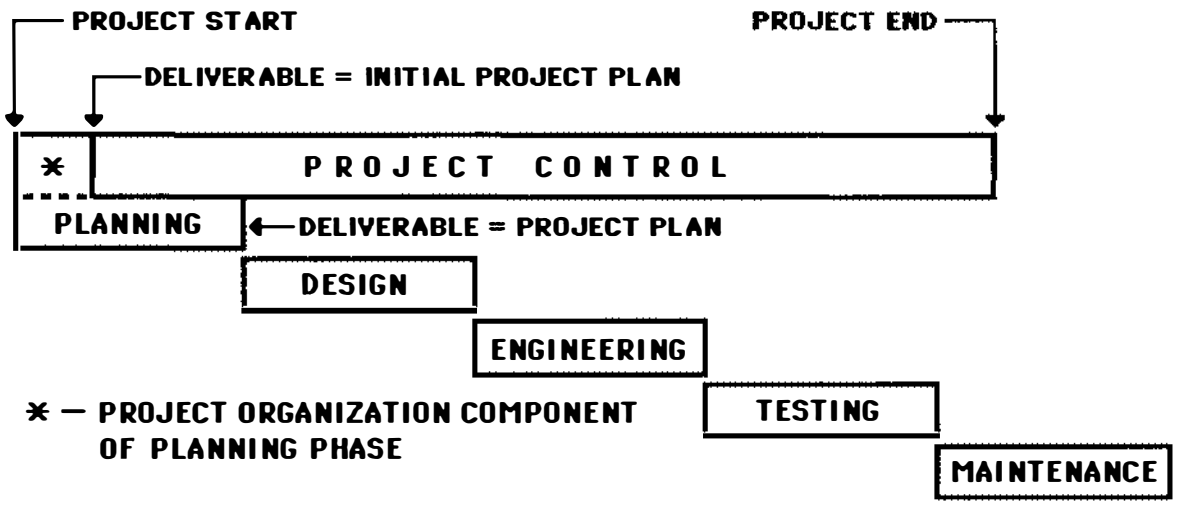
HOW TO ORGANIZE A SOFTWARE PROJECT

The key to a successful software project is being organized ... starting from project inception: the project has been authorized and the project manager named. The initial project organization sets the pattern, the tone, the development efficiency for the rest of the project. Project organization accomplishes the following:

1. Immediately imposes order and controlled development.
2. Establishes project control over the entire project.
3. Crystallizes the project goals, objectives, and expected end product.
4. Documents all of the above.

Software projects differ from other projects in their deliverables, their quality metrics and, to some extent, their lack of resilience to schedule delays. The organization component of project planning removes ambiguities in deliverables, defines the metrics by which the project will be judged successful, establishes schedules, identifies dependencies and specifies contingency actions.

The figure below shows the conceptual project management approach superimposed over a typical software lifecycle diagram.

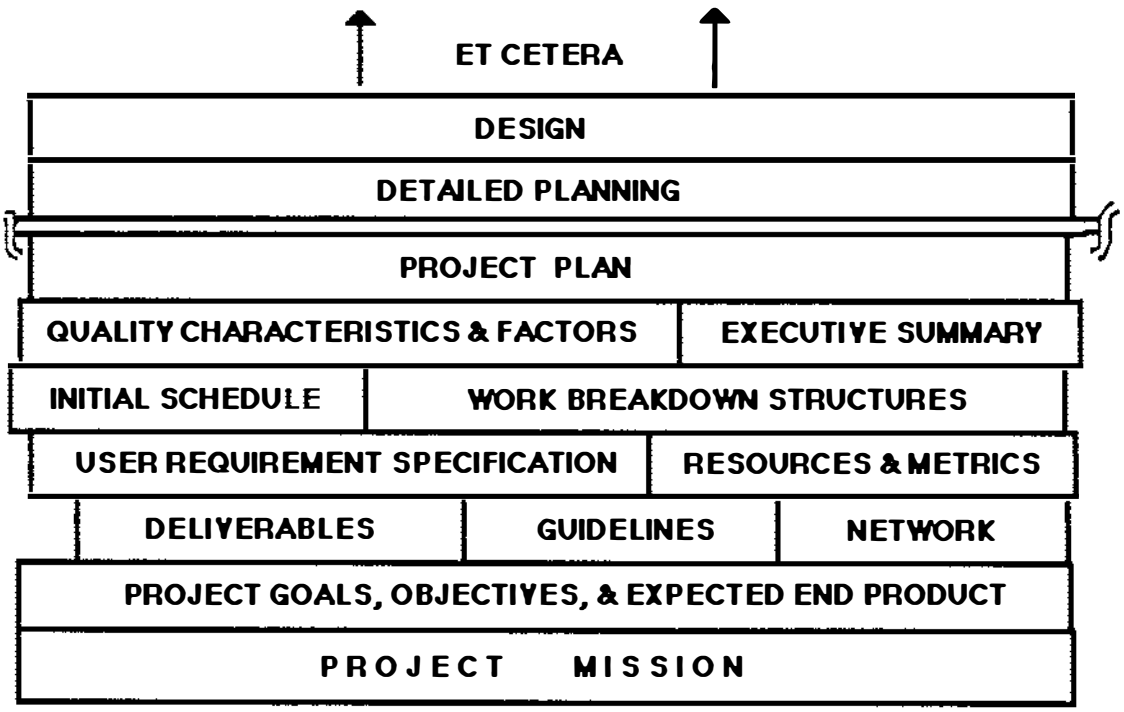


As shown above, project organization is the first part of the Planning Phase. The organization component of the Planning Phase documents the project goals and objectives, and establishes the operational context that will be in effect for the life of the project. The end product of the project organization component of the Planning Phase is an initial Project Plan. While the work done during project organization does not guarantee a

successful project, following a Project Plan based upon a proven systematic approach reduces potential problems and is the major determinant of project success. Project organization immediately imposes a systematic approach to, and control over every part of the developing project. The proposed organization concept is not a tedious or ridgid fill-out-a-form exercise, but a generic, flexible approach suited to all projects.

A BUILDING BLOCK CONCEPT IS THE KEY

Every project organization step expands and builds on the previous step similar to a building's foundation as illustrated below.



Similarly, each project organization step generates a document, which cumulatively, become the initial Project Plan. Some or all of the initial Project Plan documents will undergo iterative revision as part of detailed planning in the Planning Phase. The Planning Phase deliverable is a final Project Plan consisting of updated versions of initial Project Plan documents.

OVERVIEW OF PROJECT ORGANIZATION

It is important to note three things that occur during the project organization portion of the Planning Phase. One, virtually all activities are

project management oriented as opposed to engineering oriented. Two, an orderly & systematic development approach during project organization is introduced that will be used throughout the the remainder of the project. And three, control is imposed on all project events. Nothing happens or proceeds unless it is in accord with the Project Plan. This approach precludes projects starting in a chaotic manner or out-of-control projects.

Project organization starts when the project has been authorized and assigned to a Project Manager. Project organization ceases as a formal activity when its primary deliverable, the initial Project Plan, has been completed and approved. The Project Plan serves as the master control document during subsequent development phases until the project is finished or cancelled.

When the initial Project Plan comes into existence, the Project Sponsor, Functional managers, Project Team and Project Manager can easily review the Project Plan and verify that the project mission is understood and that project goals and objectives are consistent with the mission. There should be sufficient documentation in the initial Project Plan to demonstrate that the project remains feasible within high level quality, cost and time parameters. At this point, pure project management activities diminish, detailed planning proceeds, and software engineering becomes an ever-increasing part of project activities.

PROJECT ORGANIZATION CRITERIA

While each software project is unique, there is a commonality with respect to the documents that contribute to the Project Plan. These documents are described in the Toolkit and Exhibits sections of this paper. Not all documents described in these sections are contained in the Project Plan; some are only temporary documents to facilitate organization and are not part of the Project Plan. However, the collection of individual documents constitutes the Project Plan and, altogether, must meet the criteria for an organized project.

Project organization is complete when the following criteria are satisfied.

1. The project mission is clearly stated.
2. The authority of the project manager is clearly stated.

3. The project's goals and objectives are clearly defined.
4. Project guidelines are established.
5. Time frame and target dates are established.
6. Project Manager or Sponsor is operating
7. A logic diagram or PERT chart of project development exists.
8. Summary milestone bar chart is completed.
9. Cost guidelines are established.
10. Preliminary statement of work describing who,what, when, where, how, and how much is prepared.
11. A User Requirements document exists.
12. End item or product oriented Work Breakdown Structure (WBS) is prepared.
13. Risks, their probabilities, impact, and contingency actions are identified.
14. Project change procedures are defined.
15. The Project Plan, containing the above documents, has been published and approved by the Project Sponsor and the Functional Managers affected by the project.

If these criteria are met, then the project is organized. This doesn't mean planning is complete, or that this is all of the organization or planning required. Project organization is only the first part of planning. Organization establishes an environment in which further detailed and accurate planning can flourish.

PROCEDURAL STEPS FOR PROJECT ORGANZATION

The assumption is made that a project manager is appointed and concurrently given a new project and project mission. Sometimes, the project mission is vaguely worded and sometimes the project manager receives a voluminous folder of previous studies, estimates, notes, etc. Whichever, is the case, the organization actions are the same. First individual organization, then "braintrust" (a group of individuals who possess information relevant to the project) organization, then project organization. Generally, it should happen that way. There are times, however, when all three organization steps happen concurrently. Here is an example of the procedural steps involved.

Individual organization:

1. Set up project file.
2. Put in basic documents

A. Action item control form (from "toolkit")
B. Change register form (from "toolkit")
C. Commitment register (from "toolkit")
D. Project Correspondence log (from "toolkit")
E. Add previous project proposal, project authorization
F. Add all previous logic diagrams, cost estimates and correspondence.

"Braintrust" organization

3. Identify and acquire commitment from members of the "brain trust"
4. Use a "brain trusting" tool to define initial project plan. (Optional)
5. Use "brain trust" to prepare rough WBS - responsibility and end item oriented.

Project organization

6. Document and clean up "brain trust" output - format into pieces of initial project plan.
7. Prepare preliminary Statement of Work (SOW)
8. Identify global project risks, their probability and contingency action for each risk. .
9. Complete organization package (recognizing parts may be revised before approval).
10. Prepare project review notice (from toolkit) - present results to project team, then Project Sponsor - obtain approvals.
11. Continue with detailed planning.

COMMENTS ON THE PROJECT ORGANIZATION "SYSTEM"

As shown above, the imposed systematic approach integrates step-by-step procedures and the application of project management "tools".
It is a very deliberate, but flexible, approach. Never deviate (at least not without forethought and good reason); never rush off to start coding or start sketching out the high level system design. First things first, according to the "system".

An important task is to identify and acquire the personnel that make up the project's "brain trust". The "brain trust" is a group of individual contributors and managers who possess information and expertise pertinent to the project being planned. They usually represent the major operational components of the organization participating in the project, e.g. Marketing, Engineering, Quality Asssurance, Manufacturing, and Technical Publications. The number of "brain trust" core members should not exceed ten and 4 to 5 would be ideal. Additional personnel for "brain

trusting" activities can be added for short term activities.

The "brain trust" accomplishes the following:

1. Refines the mission and develops the project goals and objectives.
2. Describes the project results in sufficient detail so the criteria for success or failure of the project can be determined.
3. Develops project guidelines (instructions) to assure that the proper mechanisms will be in place to keep the project on track.
4. Defines milestones to assure that the project can be tracked and that timely corrective action can be applied when necessary.
5. Identifies risk areas and possible courses of action to minimize the adverse impacts of such risks.
6. Prepares all of the initial documents necessary to communicate, manage, and control the project.
7. Uses the project manager's "toolkit" for preparing and imposing control mechanisms and documents.
8. Uses the appropriate project management "tools" for organizing the subsequent phases of the project.

How and when to specify quality metrics that should apply to a project has been on on-going development problem. As a remedy, consider using a hierarchial development concept. There are four quality catagories that should be developed during project organization as quality guidelines. One of these, quality factors, is part of a conceptual process that merits further explanation later. The four quality catagories that appear during project organization are:

1. Quality control mechanisms to be used.
2. System characteristics.
3. Relative importance of quality factors as development guidelines.
4. High level management guidelines for creation and control of project quality.

The key to developing the quality aspects of a project is to start with a hierarchy of quality considerations. This concept orders quality considerations into a descending hierarchy of detail. Each level of the quality hierarchy should be developed after the preceding one and at the appropriate point in the project. There is no point in discussing a level five quality item during project organization when level 1 and 2 items should be specified. To do so would be premature because the proper hierarchial foundation would not exist. For example, consider this hierarchy of quality definition.

1. High level guidelines (4 catagories mentioned above)
    A. System characteristics
        1) Quality factors
            a) Attributes
                (1) Specific metric to be applied

Examples of system characteristics listed below have been adapted from Cooper & Fisher, Software Quality Management.

1. Human lives affected
2. Long life cycle
3. Real time application
4. Classified information processed
5. Interrelated with other systems

The above examples of system characteristics affect the basic project quality requirements and provide a schema for analyzing the proposed system characteristics of each unique software system.

Each fundemental system characteristic may be further analyzed in terms of quality factors that are intrinsic to the system characteristic. Examples of quality factors, again adapted from Cooper and Fisher, are:

-Human lives affected (characteristic)
    -Reliability
    -Correctness
    -Testability
-Long life cycle
    -Maintainability
    -Flexibility
    -Portability
-Etc.

In similar manner, attributes can be chosen for each critical quality factor and metrics can be specified for each attribute. However, this sort of detailed analysis is done later in the planning phase and is not developed during project organization. It is only the concept of decending orders of quality detail that is important during project organization. The intent is to reduce the quality considerations to a checklist for easy reference and to construct a conceptual framework when tradeoffs need to be evaluated. Tradeoff analysis of various quality factors and attributes are not done during project organization, but merely noted as a follow-on task to be done later in the Planning Phase.

Coming back to specifying quality control mechanisms, some examples may help to explain the kind of guideline desired.

1. Reviews
   -Plan and output approvals at key development points.
   -Walkthroughs
   -audits
2. Status reports
3. Interim documentation review
4. Program source code
5. Etc.

Most of the quality control quidelines will come from corporate quality policies and procedures which are integrated into the project guidelines.

Other examples of high level quality guidelines may deal with aspects of quality as the project develops. Here are some typical guideline expressions.

1. Prepare plan to achieve attributes for each critical quality factor
2. Target dates for milestone quality events: functional specification, alpha testing, beta testing, etc.
3. Tools & techniques to be used
4. Who and how expressions

In summary, the above discussion illustrates that using the concept of a hierarchy of quality concerns provides perpective in developing the project. During project organization, emphasis is placed on specifying the high level quality guidelines that provide the foundation for further detailing during the Planning Phase. Using the quality hierarchy pattern of development forces quality planning, at the appropriate level of detail, at the proper point in the development of the project. More perspective is provided in the discussion of quidelines below.

THE ORGANIZING "TOOLS"

A tool is defined as any technique, methodology, form, document, aid, hardware, or software which can be employed to improve efficiency or productivity of project development. Hence, a project manager's "toolkit" is a collection of all useful project management tools organized to be handy and ready. Once developed, a manager, project manager, or project leader should be able to write a project prescription for use of particular

tools. An example of a typical project organization "tools" checklist is shown in the Exhibits.

BRAINTRUSTING TECHNIQUE

An aid to efficient project organization is the use of a "braintrusting" technique. The front end of a project requires the quick distillation of wisdom from the project staff and associated functional support members. The idea is to get all of the pertinent advice out of the minds of the project team and onto paper. One "braintrusting" tool that does this efficiently is called LOGPAD. LOGPAD is an acronym standing for list of activities, objectives, guidelines, plan-of-attack, alternatives, and data sources. A detailed script for employing the LOGPAD technique is shown in the Exhibits.

GOALS, OBJECTIVES, AND GUIDELINES CONCEPT

The concept and use of goals, objectives, and guidelines is a useful project management tool. Although the project management literature is not consistent in definition; goals, objectives, and guidelines must be developed on every project. Here is a set of definitions to further our explanation.

GOAL: a proposed achievement that can be quantified, measured, or demonstrated in some fashion.

OBJECTIVE: a non-quantifiable achievement.

GUIDELINES: a grouping of instructions, objective statements, or criteria that states desired actions, restraints, or guidance for project development.

Goals and sub-goals usually represent the overall and secondary tangible project achievements. Objectives are used to give subjective direction or instruction and usually modify how a goal is obtained. Guidelines are a convenient grouping of all the instructions pertinent to the project. Guidelines may act as a repository for all of the corporate project development guidance. For instance, Division or Corporate Project execution instructions are integrated into the project guidelines. What that means is that the Project Manager is obligated to take policy, standard procedure, and specific boss-imposed instructions and translate them into guidelines that guide (until changed) the developing project.

There are four main benefits to using the concept of goals, objectives, and

guidelines.

1. It provides a common language for typical project activities.
2. In the event of project staff changes, the basic project direction and instruction are a part of the project documentation - easily referenced for continuity of management and planning.
3. Articulation of qoals, objectives, and guidelines forces better organization and control.
4. Project documentation of goals, objectives, and guidelines provides a succinct and easily located statement of the most important project information for managerial review.

These are some examples of typical areas that would be covered in the project quidelines.

1. Project Manager authority and decision latitude
2. High level cost and time targets : dates, dollars, etc.
3. Quality aspects such as:
   -quality standards to be followed
   -quality reviews
   -peer reviews
   -configuration management requirements
4. Resources available
5. Funding strategies
6. Timing
7. Approvals When? By whom?
8. Planning detail
9. Priorities
10. Coordination requirements
11. Reporting requirements
12. Potential problems
13. Critical dates
14. User-client-customer concerns
15. Etc.

PROJECT PLAN

The most important Project Manager's tool is the Project Plan. This grouping of documents is a relatively concise compendium of all project documents which describe how, who, how much, when, where, and what shall be done as part of the project. Atthough every organization has its own version of a Project Plan, here is a representative list of contents of that portion to be developed during project organization. Previously

described as an initial Project Plan, it includes:

- Mission statement
- Statement of project goals and objectives
- Description of desired end product/output/service and major deliverables
- High level project guidelines for quality, cost and time
- Broad statement of work
- End item oriented Work breakdown structure
- Milestone summary barchart
- Project logic diagram or Network
- Risk evaluation - do on global basis

WORK BREAKDOWN STRUCTURES

The next most important tool is probably a work breakdown structure (WBS). This is also the primary scope management tool used on a project. A WBS may be oriented in several helpful ways:

-by responsibility
-by reporting hierarchy
-by activity
-by end item
-by cost account
-other

However, the end item orientation is most useful in the organization of a project. The only dictum is be consistent in orientation and try to develop as much detail as time allots.

RISK ANALYSIS

Risk evaluation is a very important part of project organization or planning activities. Most Project Managers do not spend adequate time in evaluating what could go wrong, where the high risk areas are, where the project's greatest exposure to problems are. A good guideline is to spend 20% of available planning time in evaluating risk. Basically, evaluating risk involves asking a series of questions similar to the ones posed above.

A good risk evaluation technique is one described by Kepner & Tregoe in their book, The Rational Manager. Their technique, Potential Problem Analysis (PPA) uses a series of questions and a procedure paraphrased below.

What can go wrong?

What are the major problems?
What are the technical risks?
What are the political, managerial, and environmental risks?
Then ask for each answer: What is the impact on the project? - low, medium or high? What is the probability of occurrence? - low, medium, or high?
Then ask, What are the likely causes?
Next, what are the possible actions to resolve each possible risk?
Finally, on a global project basis, determine for each possible risk judged to have high impact on the project and a high probability of occurrence the best resolving action. When this procedure is completed, integrate the problem resolving action into the project plan in such fashion that problems are prevented.

OTHER TOOLS

Other tools in the project manager's toolbox are in common use and have already been mentioned:

- -Networks or logic diagrams
- -Summary barcharts - preferably event oriented
- -Project budget or preliminary cost estimate
- -Standard forms, standards, and document layouts

The above tools vary in format and application with each organization. However, it is the application and use that counts. Proficiency in using the toolbox concept comes with practice.

SUMMARY OF PROJECT ORGANIZATION

The end product of project organization is an initial Project Plan. Its parts or pieces may be embryonic or very complete depending on the size and complexity of the project. Starting at project inception, documents come into existence, one at a time, as project organization proceeds. Documents may be generated through use of standard techniques or may be a completed standard form from the "toolkit".

After the initial Project Plan is approved, individual documents are updated and others are added as detailed planning proceeds. During the remainder of the Planning Phase, the initial Project Plan evolves into a Project Plan which contains the expected achievements, baseline cost,

time and quality parameters for the entire project.

Generating an initial Project Plan for approval serves as a quality and performance checkpoint. There has been some work done and tangible output to review. Operating Managers and the Project Manager can easily review the Project Plan, and verify that the project team has interpreted the basic mission correctly. Has a terse mission statement been expanded into documents which verify a straight march to the intended project goal? Typically, a cover letter requesting approval of the initial Project Plan also provides evidence that the project is still feasible - within high level cost, quality, and time constraints. This approval point is also a point of confirmation. If results are satisfactory, the project proceeds. If not, revise the project or kill it at a point where investment is at a minimum.

THE BENEFITS OF SYSTEMATIC PROJECT ORGANIZATION

A systematic approach to organizing any project, using the "tools" described, is the most efficient way of mass producing successful projects. Some of the benefits of this systematic organizing approach are not immediately apparent. Consider this listing of benefits.

1. From project start to approval of initial plan, the project is developed according to a systematic, efficient method. This approach never lets a project get out of control or ever start in a chaotic manner.

2. Using a project manager's "toolkit" and a systematic approach reduces project organization to hours and days, not days and weeks. Many small projects can be organized in less than a day. It takes more time to type the output documents than it takes to do the organizing.

3. In a very short amount of time, documents are developed which help communicate, control, and manage the project. The same documents can be used to obtain commitments, conduct briefings, and portray status.

4. This is a standard approach to project development that can easily be taught and integrated into the corporate project execution manual.

5. Systematizing project organization is an efficient method of getting a project started and provides a conceptual basis for further efficiency improvements. For example, manual methods are described herein. The

same process can be automated on a computer for greater efficiency.

6. Systematically organizing a project builds a foundation for subsequent project development with minimum rework. Identified problems should decrease at each stage of subsequent design and development because of the thoroughness of project organization. In so doing, the probability of project success is greatly enhanced.

GLOSSARY

PROJECT: Any unique, one-time, one-of-a-kind undertaking concerned with achieving specific objectives.

PROJECT MANAGER: The individual with responsibility and requisite authority for the planning, directing, leading and controlling of human and material resources to achieve specific project objectives, i.e., on-target technical performance, within budget and time constraints, meets required quality criteria, and provides a desired level of user satisfaction

PROJECT LEADER: An individual responsible for leading a functional team tasked with meeting the technical objectives associated with a project.

FUNCTIONAL MANAGER: The individual responsible for the administrative support and assignment of personnel doing specific functional duties, e.g., technical publications, manufacturing, etc.

PRODUCT: A tangible item produced as part or wholly from project activities.

PLANNING PHASE: That portion of a project which encompasses the majority of project planning and documentation that defines and directs subsequent phases of the project.

EXHIBITS:

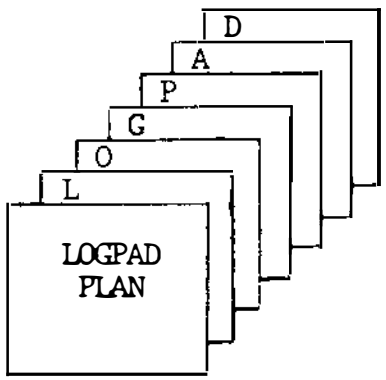
LOGPAD description
Sample project manager's toolbox contents
References

L O G P A D P L A N N I N G T E C H N I Q U E

Here is a simple planning technique that you can use to get the initial planning of a task, job, problem started under your control.

The technique is best suited for easel presentation. But, can be done on blackboard or on 8½" X 11" sheets.

Here is the overall plan. You are to take 1 minute to define the problem. Then ask leading questions to draw out your "Brain Trust". You will accumulate 6 pieces of paper that will form a "Plan", developed in a maximum time of 30 minutes.



LOGPAD is an acronym that stands for:
List of work items
Objective & output
Guidelines & assumptions
Plan of attack
Alternatives
Data sources

STEP 1 BRIEF YOUR "BRAIN TRUST"

Generally describe your mission, how you're going about it & what you want to accomplish.

Check the time; 30 minutes from now you're through - absolutely - positively quit! This is a very disciplined method & you as group leader have to be conscious of time - you cannot afford to tie up endless manhours on planning details.

This procedure works because 80% of a problem or task can be detailed quickly within 30 minutes. The other 20% will be added as we work the plan & will usually cause no significant change in planning direction.

STEP 2 LIST OF WORK ITEMS

A - Label top of easel sheet: (whatever fits)
Work list
List of work
List of work to do

B - Ask for work items & put all suggestions down.
Do not screen or evaluate. You can do that later. Get the major ideas down quickly. You want to make use of "piggybacking" concept as used in brainstorming.

C - Ask leading questions like:
- What else is involved?
- Any other items?
- Is this all the work?
- Any work we left out?
- If we did all this would we have completed the project or solved the problem?

D - When discussion sounds like the're really searching for added work items cut it off & move to next step.

STEP 3 STATEMENT OF OBJECTIVE

Normally, if you follow textbook theory, you state your objective & then talk about all the work that has to be done to achieve the objective. However, 80% of the time you cannot state a very clear objective to start. You then spend 30 minutes, alone, just arriving at a definition of what is to be accomplished.

That is why you talk about work items first. Everybody can think of pieces or parts of the work that needs to be done. This gets everyone warmed up & then you'll find you can arrive at a better definition of objective in short order.

A - Either write down the objective or ask "what is it we're to accomplish?" "How should I word that?"

B - Listen to the answers & formulate a concensus.

C - Tidy up the statement of objective & then move on to output.

D - Define what output shape & form should be, if it is not clear from objective statement.

E - Be prepare to leave this subject quickly & fill in some ideas as plan of attack is generated.

STEP 4 GUIDELINES & ASSUMPTIONS

A - Label top of sheet Guidelines & Assumptions

B - Ask questions like: "What assumptions should we make?" "What guidelines should apply to this _____ work?"

C - Quickly list answers & continue prompting.

D - Draw out additional guidelines & assumptions by pointing to a work item & saying: Are there any assumptions connected with this piece of work? Should we set any guidelines for this item/work/area/activity? Ad infinitum

E - Also ask: "Is there anything that should be included or excluded as a part of this work?" That question will help to define the parameters, dimensions & boundaries of a problem or task.

F - As soon as discussion dies out move to next step.

STEP 5 PLAN OF ATTACK

Now prepare a modified Pert Plan, CPM, Flow Chart, or list of Priorities.

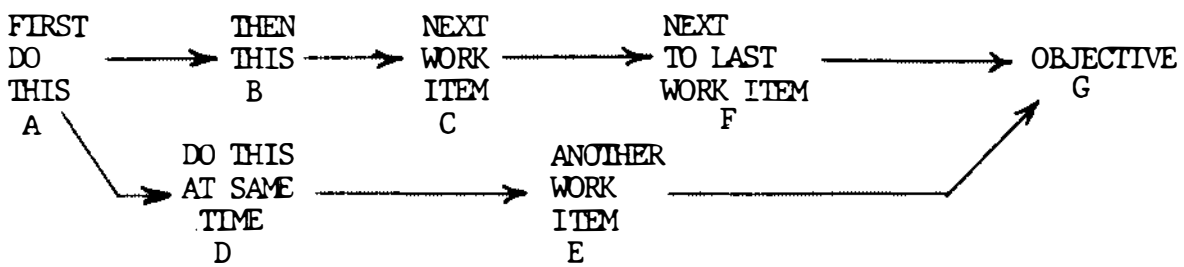
A - Point to list of work items & ask:
"Which of these should be done first?"

B - Label that - # 1 Priority

C - Continue on with priority 2, 3, 4, 5 etc.

D - If 2 items same priority, usually means can be done at same time or concurrently by two different people. Otherwise, one must be done before the other.

E - Try to arrive at a simple flow chart, arranged from left to right, that expresses the priority of accomplishment & should look like this in format.



F - Paste two easel sheets together if more room needed.

STEP 6 ALTERNATIVES

A - Label a sheet Alternatives & Evaluations

B - Ask the question: "What alternatives will we have to consider as a part of this work?" Follow up with: "What things do we need to study or evaluate?"

C - This is the area where method studies or cost evaluations need to be done so ask questions in relation to each work item.

D - Be aware that this step may generate additional major work items so go back & add to list of work items or plan of attack.

E - As discussion dwindles go immediately to next step.

STEP 7 DATA SOURCES

List all references, books, papers, organizations, people, sources, or any other possible place where you need to check for pertinent information. (Note: A good listing can start a new man off running & being efficient first day on job.)

A - Start listing most obvious people to be contacted & documents published.

B - Solict comments from "Brain Trust" with "any other sources of information?" "Who else do we need to talk to?" "What other papers should we read?"

C - As discussion dies down you check time & notice 30 minutes exactly has passed.

D - At this point you hang up your ink pen & quit. Positively!

E - Summarize results of major steps briefly in 1 minute.

F - "If there is no futher discussion relevant to this plan, I will have these charts typed on 8½" X 11" paper & distribute to you. Subsequent work will amplify & expand the plan in greater detail - but, at least we have a plan of attack!" At that statement heads will nod in agreement. That is your cue to thank your 'Brain Trust' for their fine help & dismiss them.

THE PROJECT MANAGER'S "TOOLKIT"

The project manager's toolkit includes forms, documents, software templates, standard techniques such as Investors Rate of Return analysis, etc. that are used regularly on projects. The intent of formally defining a toolkit is to make the concept and content handy and easy to use. By doing so, project development efficiency is promoted.

Here is a partial listing of a sample toolbox with a few examples attached.

Project organization checklist *
Document tracking form *
Action item tracking form *
Change Register form *
Phase completion form
Correspondence log *
Responsibility matrix form
Summary barchart form - project schedule *
Project cash flow analysis form
Budget form - several variations
Commitment register form
Project plan approval form *
Risk evaluation form
Project kickoff meeting checklist *
Project manpower planning form
Status reporting format *
Narrative Problem Analysis form

* = sample form attached. Samples chosen to illustrate the type and variety of toolbox content. Many others possible. The concept and use of a project manager's toolbox should be a part of the corporate project execution manual.

PROJECT ORGANIZATION Rx

Check if Req'd		Check when done
_____	Project misssion statement prepared?	_____
_____	Define project goals	_____
_____	Define end product/result	_____
_____	Define project guidedlines	_____
_____	Prepare list of activities and durations	_____
_____	Do logpad planning exercise	_____
_____	Prepare end item oriented WBS	_____
_____	Prepare project milestone summary bar chart	_____
_____	Prepare project logic diagram / network	_____
_____	Setup project Files	_____
_____	Institute project change register	_____
_____	Set up action item tracking system	_____
_____	Define Project manager resposibilities	_____
_____	Setup commitment register	_____
_____	Setup document tracking system	_____
_____	Status reporting system setup	_____
_____	Project estimate available?	_____
_____	Prepare project budget	_____
_____	Do global risk evaluation	_____
_____	User requirements specification prepared?	_____
_____	Prepare project plan	_____
_____	Hold project kickoff meeting	_____
_____	Project organization approval	_____

VENDOR/CONTRACTOR DOCUMENT TRACKING

DRAWINGS - DOCUMENTS

SHEET _____ of _____

PROJECT _____ PROJECT LDR _____

ITEM NO.	CONTRACTOR/ VENDOR NUMBER	NUMBER		DESCRIPTION	STATUS	SUBMITTALS						FILE REFERENCE
						REV. 0	REV. 1	REV. 2	REV. 3	REV. 4	REV. 5	
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							
					REC RET / STATUS							

ACTION ITEMS

AS OF: _____

PROJECT: _____ PROJECT No.: _____ AFE: _____ DATE: _____ Page: _____ of: _____

No	RESP	ACTION ITEM / PROBLEM / ITEM TO BE RESOLVED	SOURCE	DUE DATE	DATE	REMARKS

PROJECT CHANGE REGISTER

PROJECT _____ AS OF _____

BY _____

CHANGE DATE	REGISTER		CHANGE DESCRIPTION	IMPACT OF CHANGE			REASON FOR CHANGE -REMARKS-
	CHANGE NO.	REFERENCE		COST	TIME	OTHER	

PROJECT INCOMING/OUTGOING CORRESPONDENCE LOG

REC'D/ SENT	DATE	FROM	TO	SUBJECT	REMARKS	FILE #

PROJECT SCHEDULE

PROJECT: ______________________ PROJECT LEADER: ______________________

PROJECT No.: ______________________ DATE: ______________________ Page: ______ of: ______

WORK ITEM	WORK ITEM DESCRIPTION	RESP.												

PROJECT PLAN APPROVAL/ACCEPTANCE

(PROJECT SPONSOR/GM) _____

(PROJECT MANAGER) _____

(ENGINEERING MANAGER) _____

(MARKETING MANAGER) _____

(QUALITY ASSURANCE MANAGER) _____

(CUSTOMER SUPPORT MANAGER) _____

(TECHNICAL PUBLICATIONS MANAGER) _____

(MANUFACTURING MANAGER) _____

(INTERNATIONAL OPERATIONS MANAGER _____

PROJECT KICKOFF MEETINGS

One of the more important events, early on in a project, is to hold what's known as a kick-off meeting. The purpose of a kick-off meeting is to orient all of the assigned project staff with the scope-of-work and plan of execution.

Here are some of the topics discussed in a kick-off meeting:

- Briefing on project development, execution and current status.
- Work relationships, responsibilities and roles of individuals and organizations.
- Obtaining work commitments.
- Policies and procedures that will apply.
- Project Manager approvals.
- Commitment to project schedule.
- Reports, types, updates and input.
- Correspondence control.
- Purchase Requisition and expense control.
- Expectations of supporting organizations.
- Who's going to do what...when.

The Project Manager usually organizes and announces the kick-off meeting as soon as practical (get organized first). Invite representatives from supporting organizations. Get the details hashed out and your project is up and running.

STATUS REPORT

PROJECT: _____

Date: _____

PROJECT MANAGER ASSESSMENT OF PROJECT:

STATUS MAJOR MILESTONES:

PROBLEMS AND PLANNED RESOLUTIONS:

Project Manager _____

REFERENCES

Abramson, B.N. & Kennedy, R.D.; Managing Small Projects, January 1969, TRW Software Series (TRW-SS-69-02).

Cooper, J.D. & Fisher, M.J.; Software Quality, 1979, Petrocelli Books, New York, N.Y.

Kepner, C.H. & Tregoe, B.B.; The Rational Manager, 1965, McGraw-Hill, New York, N.Y. (Note: a later edition, the New Rational Manager, is also on the market, but not available to the authors.)

Luengen, J.L.; Practical Techniques for Small Project Management- Seminar notebook, 1987, J. Luengen & Assoc., Vancouver, WA.

Macro, A. & Buxton, J.; The Craft of Software Engineering, 1987, Addison-Wesley Publishers Linited.

Price, S.; Managing Computer Projects, 1986, John Wiley & Sons, New York, N.Y.

DELIVERING ON COMMITMENTS: PROCESS MEASURES TO IMPROVE R&D SCHEDULING ACCURACY

Dick LeVitt
R&D Process Section Manager
Roseville Networks Division
Hewlett Packard Company

ABSTRACT

Metrics for tracking R&D schedule accuracy are ordinarily applied to individual projects. These metrics are effective for project management, but they typically yield little information on the accuracy of schedules for the enterprise as a whole. In a large engineering organization there are frequently many projects competing for resources in a common development environment. If schedule estimation and execution are regarded as a continuous process in such an organization then observation of the process can provide insight into common causes of schedule overrun and point to corrective actions. This observation requires a metric which responds rapidly and sensitively to schedule changes as they occur in a set of active projects.

Two candidate metrics have been developed and used to improve scheduling accuracy in Hewlett-Packard's Roseville Networks Division R&D lab. Over the past year they have been key elements in a program which has dramatically reduced our incidence of delayed project completions.

This paper describes the derivation and application of these metrics in the Roseville Networks Division lab.

BIOGRAPHICAL SKETCH

Born in Placerville, California and raised in Nevada, Dick LeVitt received his BSEE degree from the University of California at Berkeley. He worked for several companies as a project manager and section manager developing industrial data acquisition and control systems prior to joining HP in 1981. At HP, he has been a reliability engineering manager at the Portable Computer Division and a project manager at the Roseville Terminals Division. He currently works as the manager of the R&D Process Section in the Roseville Networks Division lab. Dick lives near Auburn in the Sierra Nevada foothills and enjoys photography.

INTRODUCTION

R&D product development teams at Hewlett-Packard typically commit to a project completion date at the conclusion of requirements definition, just prior to implementation. Meeting the commitment date has been a perennial challenge, particularly for software engineering teams. Last year, the Roseville Networks Division lab began a major campaign to improve the accuracy of project schedules. Key to the effort is regarding scheduling as an ongoing process subject to continuous, objective measurement.

Roseville Networks Division is a member of HP's Information Networks Group which supplies networking products for HP's technical and commercial computer systems. At any given time, approximately two dozen R&D projects are in progress at RND. Each project may suffer one or more unanticipated setbacks during the course of development. Though the particular events affecting a single project cannot in principle be predicted, the cumulative effect of all sources of delay of projects can be measured and used to improve the accuracy of future schedule estimates. Unfortunately conventional measures of scheduling accuracy, such as Tom DeMarco's EQF [1], are not suitable for real time measurement of a set of project schedules. Hence we developed two novel process measurements which provide sensitive and timely indices of project progress in the RND lab.

One of these measures is a modified form of DeMarco's Estimation Quality Factor. By recording schedule adjustments on a month-by-month basis, we are able to make an accurate running estimate of the aggregate EQF of all lab projects. Another measure is Project Progress Rate which provides an ongoing prediction of the average actual project durations normalized by the average of the estimates. These metrics provide rapid feedback to schedule estimators on the accuracy of their project plans. Since we began the measurements a year ago, the effective projected EQF of the RND lab has improved by a factor of three. Our projected average Lab-phase schedule duration overrun has been reduced from 70% to about 20%.

PROCESS PERSPECTIVES

Like other HP entities, RND uses formal software and hardware lifecycles to define the internal process steps required for new product development. The most important milestones of the process are the Investigation-to-Lab (I-L) checkpoint and the Manufacturing Release (MR) checkpoint.

o Investigation-to-Lab (I-L) marks the transition from the definition and project planning phase to the implementation (Lab) phase. At this checkpoint, the project manager is expected to have prepared a detailed project plan with dates for the intermediate milestones and for the MR milestone. This is the time at which the formal commitment is made by the project team to deliver on a particular date.

o Manufacturing Release (MR) is the official end of development; the point at which all lab engineering and beta test work is complete. The product is suitable for customer shipments at this time.

Of the approximately two dozen projects active at RND, somewhat less than half are in the Investigation phase and the remainder are in the post I-L Lab phase. This is a large enough number that the projects can be considered collectively from a process perspective: process measurements of scheduling accuracy can be applied and process improvements identified which have benefits across the organization.

The central issue in R&D for delivering on commitments is the match between ACTUAL Lab phase durations and I-L ESTIMATES of duration. Historically, the ratio of actual Lab phase project durations to the I-L estimates has been about 1.7 at RND. Our process objective is to improve the ratio to 1.1 or better.

PROCESS MEASUREMENTS

Techniques in common use within HP for project tracking work well when applied to individual projects. A "Brunner diagram", for example, gives a good picture of time and effort expenditure in comparison with estimates as a project progresses. But it is difficult to combine information from Brunner diagrams to get an overall sense of how well an organization is managing projects.

Likewise, DeMarco's EQF has been used to evaluate the quality of schedule and effort estimates for individual projects. DeMarco suggests that project EQF's be averaged to obtain an overall value to represent an organization. Unfortunately the average can be made useless by the presence of one well planned project with very high EQF. EQF has a second drawback as a process measurement. The metric is normally calculated upon project completion, so it does not provide rapid feedback to aid improvement efforts while projects are in progress.

These considerations prompted the Roseville Network Division to develop new metrics to monitor the scheduling performance of the lab. For the past year, we have applied the "Process EQF" (PEQF) and "Project Progress Rate" measures to all active Lab phase projects with good results. Though our focus has been on tracking project calendar time, the metrics could be equally well applied to tracking engineering effort.

A feature of the RND lab environment has made data collection for the metrics relatively simple. Each month, all project managers are required to enter schedule updates in a central database for status reporting purposes. The database currently contains information covering over four years of lab history. Data from the status reports is entered once a month into a Lotus 1-2-3 spreadsheet for the production of the metric graphs.

PROCESS EQF

Process EQF (PEQF) is a modification of DeMarco's Estimation Quality Factor specifically developed for application to a set of ongoing projects rather than to individual projects. PEQF provides a sensitive real time index of scheduling accuracy which is free of the potentially misleading effects of EQF averaging.

DeMarco's EQF

DeMarco defines EQF as a ratio of areas on a chart of estimates versus time maintained for a project. Referring to Figure 1, let $E_{i,1}$ be the first or I-L estimate of, for example, L phase project duration for project 'i'. $E_{i,k}$ is the estimate for the kth interval (of a total n intervals) during the project. A_i is the corresponding actual duration recorded for project i on completion.

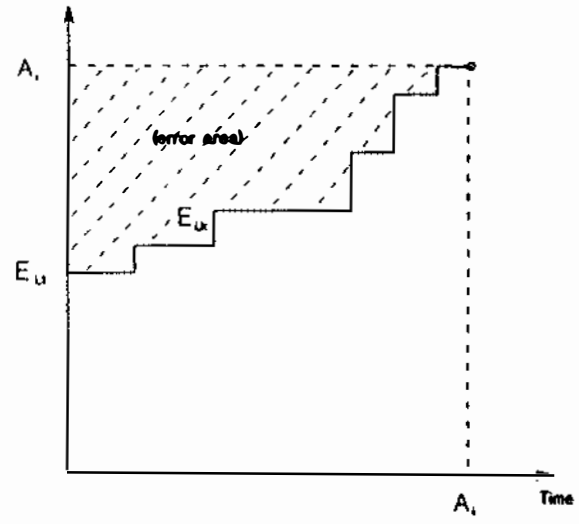


Figure 1

EQF for project i is then

$$EQF_i = \frac{\text{total area}}{\text{error area}} = \frac{A_i^2}{\frac{A_i}{n}\sum_{1}^{n} |(A_i - E_{i,k})|}$$

Consider the EQF for scheduling of a project i which experiences a constant rate of slippage. $E_{i,1}$ and A_i are the I-L estimate and actual Lab phase durations respectively. The history of this special case is diagrammed in Figure 2.

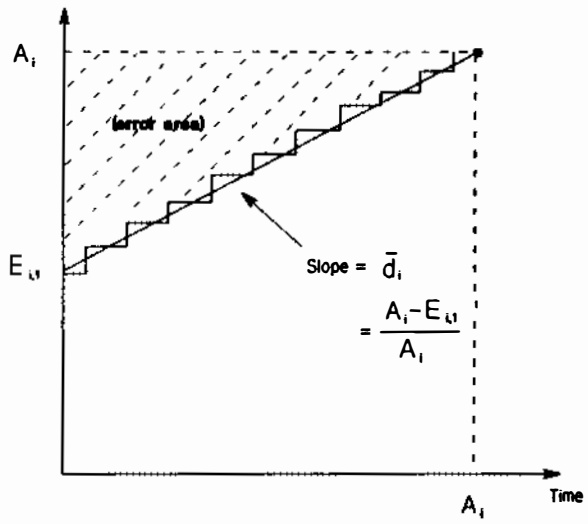


Figure 2

The average rate of slip $\bar{d}_i$ is just the slope of a line drawn from $(0,E_{i,1})$ to (A_i, A_i). This line forms a right triangle with the vertical axis, and it is clear that the error area can be represented by the triangle area.

Therefore,

$$EQF_i = \frac{\text{total area}}{\text{error area}} = \frac{A_i^2}{\frac{1}{2} A_i \, |(A_i - E_{i,1})|}$$

Substituting the slope $\bar{d}_i = \frac{A_i - E_{i,1}}{A_i}$ in the above equation we get

$$EQF_i = \frac{A_i^2}{\frac{1}{2} A_i \, |(\bar{d}_i \times A_i)|} = \frac{2}{|\bar{d}_i|}$$

Note that when the slip rate is constant we do not need to know the initial estimate $E_{i,1}$ or the actual duration A_i in order to calculate EQF.

Definition of PEQF

Consider now a set of projects collectively slipping at an average rate $\bar{d}$, where the average rate of slip is approximately constant. This situation is diagrammed in Figure 3.

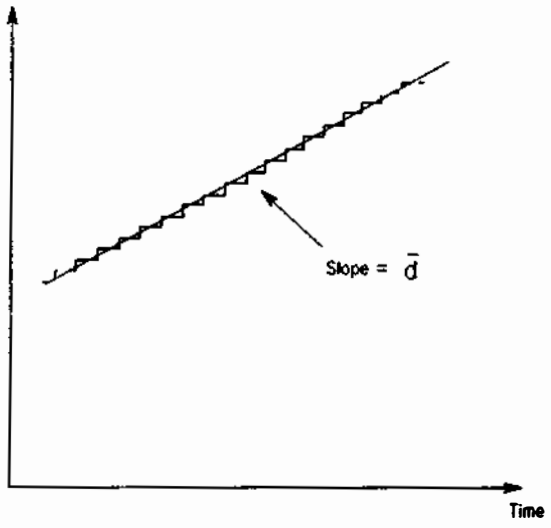


Figure 3

In analogy with the case described above, we define PEQF as

$$PEQF = \frac{2}{|\bar{d}|} \quad \text{where } \bar{d} = \frac{\text{cumulative slip (for all projects in the set)}}{\text{cumulative elapsed project time}}$$

Figure 4 illustrates the general case in which the average rate of slip varies slowly with time. A projection of Process EQF can be made at each calendar interval k using the average slip rate at k.

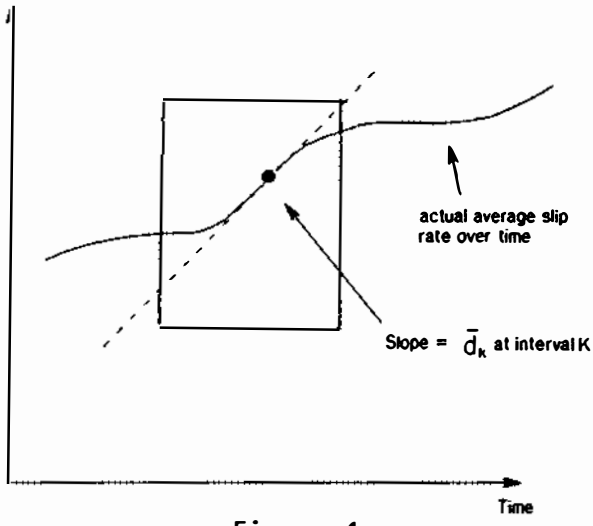


Figure 4

Thus,

$$PEQF_k = \frac{2}{|\bar{d}_k|}$$

where

$$\bar{d}_k = \frac{\text{sum of project slip times in interval k}}{\text{sum of project elapsed times in interval k}}$$

As shown in the figure, the projection is made with a tangent line to the curve at k. Observe that the tangent line forms a right triangle within the box in the diagram. It can be seen that the PEQF projection has a geometric interpretation as a ratio of areas which is similar to the original definition of EQF.

PEQF Results

Figure 5 is a plot of actual values of cumulative slip versus cumulative elapsed project time for L phase projects in the Roseville Networks Division lab. The curve includes nearly four years' project history at RND and is indeed slowly varying like the curve of Fig. 4. The interval k size in this plot is one month.

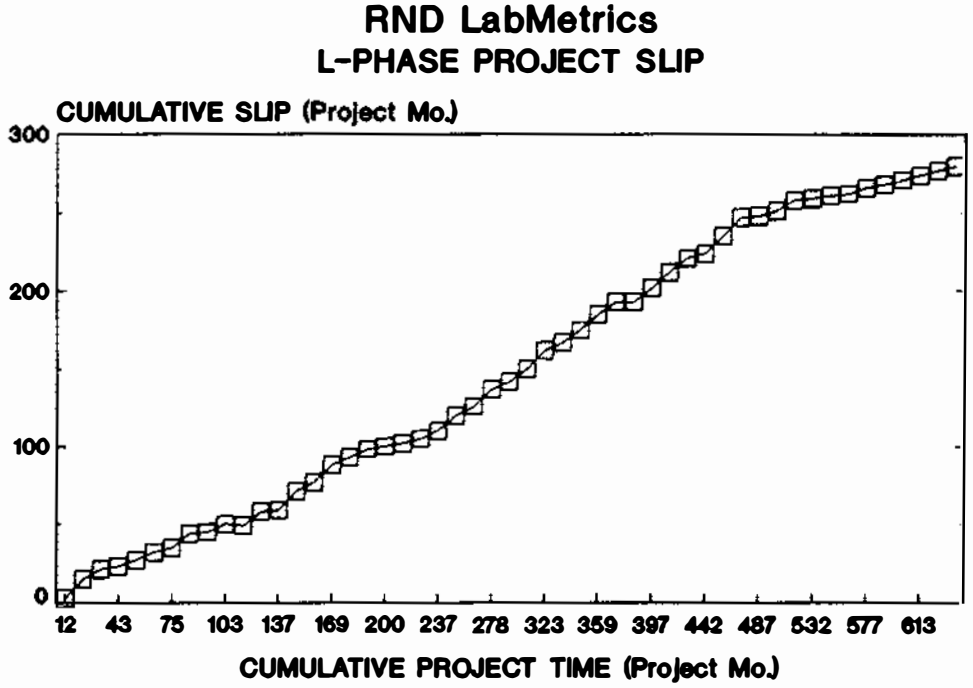


Figure 5

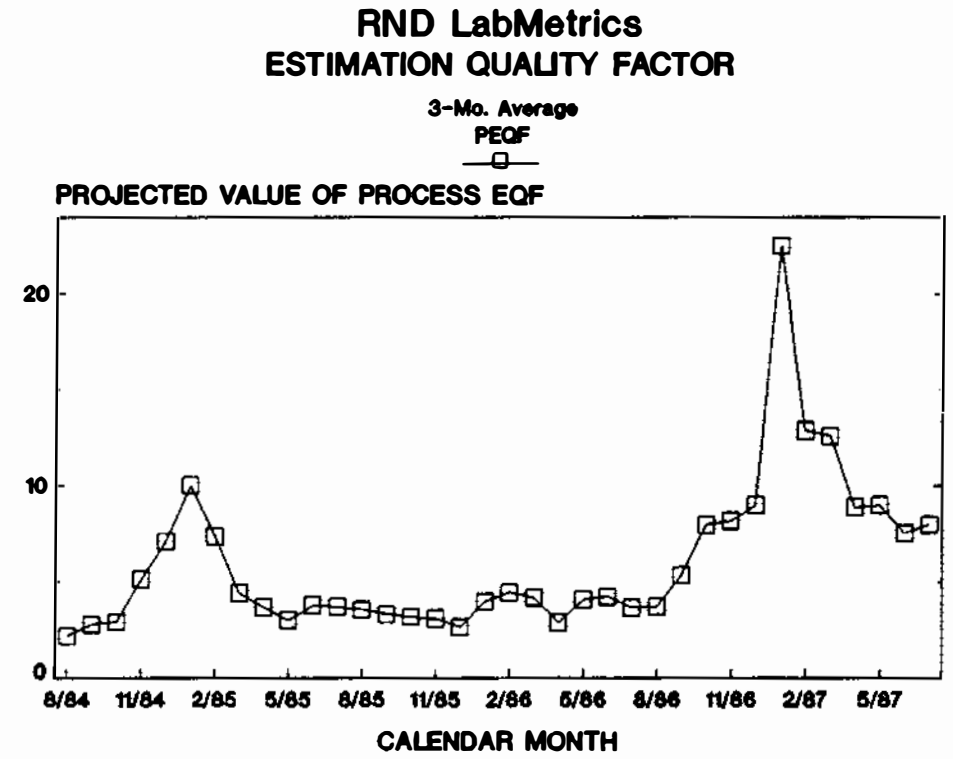


Figure 6

Figure 6 is a plot of historical $PEQF_k$ values for the RND lab. For this diagram the interval size has been increased to 3 months to reduce the effect of random fluctuations which occur in the timing of schedule adjustments. This chart has been available to Lab management each month since July 1986. Prior to July, the average PEQF was about 4. Since then PEQF has improved substantially with recent peak values exceeding 22.

PROJECT PROGRESS RATE

Although PEQF is useful, it, like the original EQF, is an abstract number having no simple or direct connection to schedule performance in management's terms. Management is less concerned with a ratio of areas than with the idea of completing a project on the commitment date.

RND has found it instructive to watch the ratio of the actual durations of projects to the I-L estimates of duration. A two month slip represents a much larger scheduling error for a 4 month project than for an 18 month project. Normalizing the actual durations by the estimates allows projects of differing lengths to be directly compared on a percentage basis.

This ratio is also helpful to evaluate the scheduling accuracy of a set of projects. Figure 7 is a scattergram of estimated project durations versus actual durations. The slope of a best fit straight line through the origin and the data points can be given by

$$m = \frac{\Sigma A_i}{\Sigma E_i}$$

where A_i is the actual duration of the ith project and E_i is the I-L estimate.

(Note to statisticians: Though this is not a least squares line, it is an unbiased estimator. The variance is somewhat larger than with least squares, but a test on RND project data shows the difference to be insignificant.)

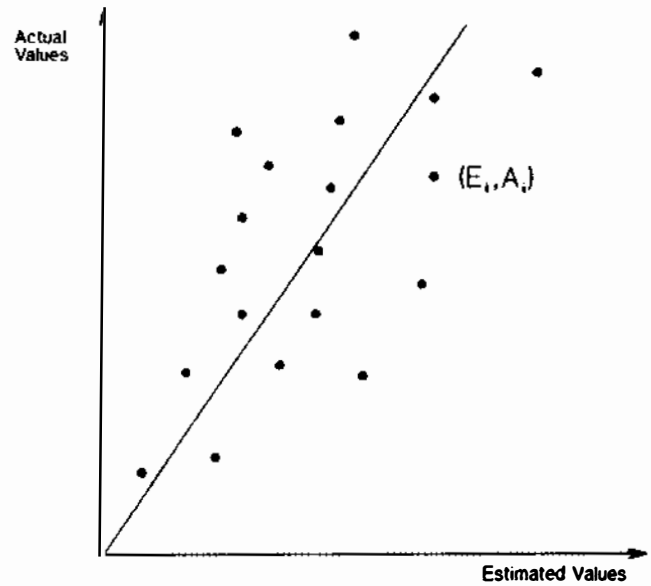


Figure 7

Unfortunately, 'm' is only available after a number of projects have been completed and hence is slow to respond to process improvements. What is needed is a real time predictor of m which is sensitive to process changes as they occur. PROJECT PROGRESS RATE is just such a predictor.

Definition of Project Progress Rate

The predictor of m we require can be developed as follows:

$$m = \frac{\Sigma A_i}{\Sigma E_i} = \frac{\Sigma A_i}{\Sigma (A_i - \Delta_i)} = \frac{1}{1 - [\Sigma \Delta_i / \Sigma A_i]}$$

where Δ_i is the total slip of project i: $\Delta_i = A_i - E_i$.

To estimate m at a time interval k we need an estimate of $\Sigma \Delta_i / \Sigma A_i$ at k.

Let

$$\bar{d}_k = \frac{\sum_i \Delta_{i,k}}{\sum_i A_{i,k}} = \frac{\text{sum of project slip times in interval k}}{\text{sum of project elapsed times in interval k}}$$

and define the Project Progress Rate at interval k as

$$r_k = 1 - \bar{d}_k \simeq 1/m$$

Project Progress Rate relates to the inverse of m rather than m directly for psychological reasons. We prefer a metric which increases like the Dow Jones index when things get better. 'Perfect' on this scale is a value of 1.0 sustained over time. Lower Project Progress Rate values correspond to higher projected values of m and hence longer average schedule overruns.

Project Progress Rate Results

An 'm' value of 1.7 was calculated a year ago using RND historical data from 21 completed projects. For comparison purposes, the Project Progress Rate metric was calculated for the same set of projects. The two results agree remarkably well: the discrepancy is just 1%. This is a strong empirical confirmation of the relationship between overall scheduling accuracy (expressed as a ratio of actual and estimated durations) and the Project Progress Rate metric. The metric has increased through the year and now forecasts an m value of about 1.2 for currently active L phase projects. If this performance is sustained, RND will deliver the next generation of products with an average of 20% or less schedule overrun.

Figure 8 is a chart of the Project Progress Rate metric as it is used at RND. The chart is updated monthly to show I phase projects (reflecting slips of the promised I-L dates), L phase projects (reflecting slips in the MR dates), and the combination of the two. The interval size for metric calculation is three months to reduce fluctuations. A lab with a smaller number of active projects may need to select a longer interval to ensure a clean and readable graph.

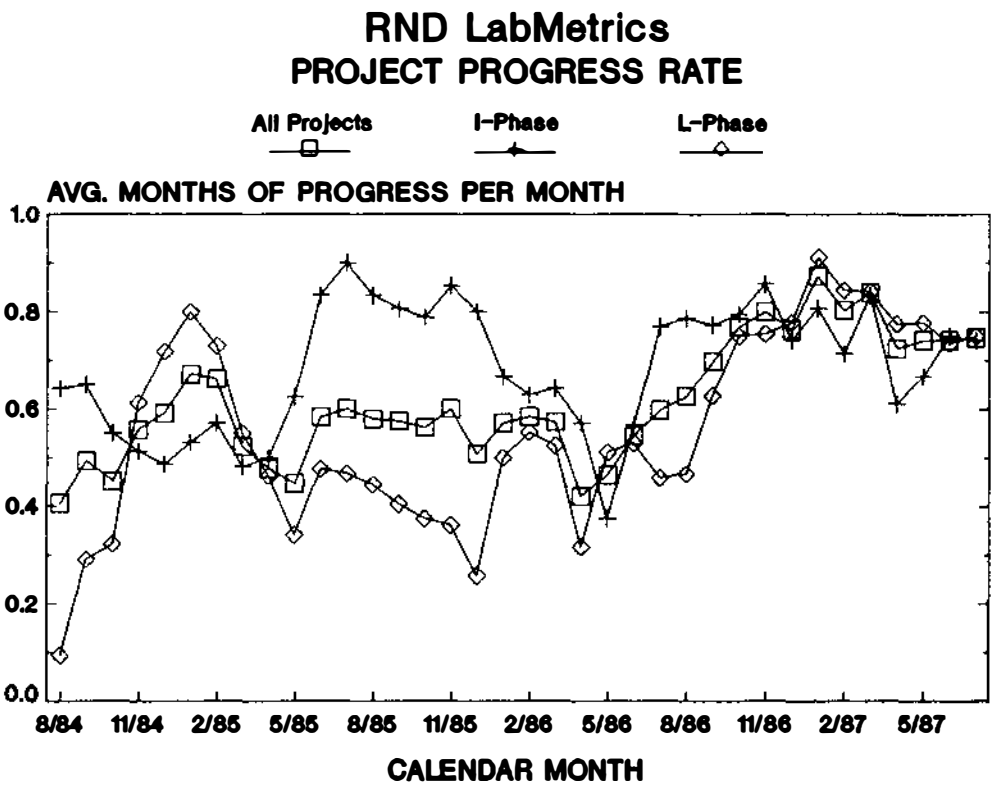


Figure 8

Even with 3 month intervals, there is noise in the RND graph. Figure 9 shows the L phase data using a 12 month interval size. This chart reveals a long term trend of declining scheduling accuracy which was not apparent in the previous graph. A dramatic reversal in this trend began at the time we made Project Progress Rate a key management tool in the effort to improve schedule accuracy in R&D.

SUMMARY AND CONCLUSIONS

This paper has outlined the derivation and application of two novel process metrics which can monitor overall scheduling accuracy in an R&D organization. The two metrics are quite similar: they each report average scheduling performance across many projects in a timely manner. The similarity has a deeper origin however. Both the Process EQF and Project Progress Rate measures are based on the average slip rate, d_k. Thus the choice between PEQF and Project Progress Rate is primarily a matter of style. PEQF is recommended for those organizations which have internalized EQF as a project estimation measure. Like EQF, PEQF tends to magnify small differences as scheduling perfection is reached. Values approaching infinity are theoretically possible. By contrast, Project Progress Rate has a more uniform scale with values approaching 1.0 as accuracy improves. Project Progress Rate has the additional advantage of forecasting the average percentage schedule overrun for the organization. Project Progress Rate is the primary schedule management metric at RND. The metric is quite scalable and could in principle be used at a division lab level, at a group level, or across an entire corporation.

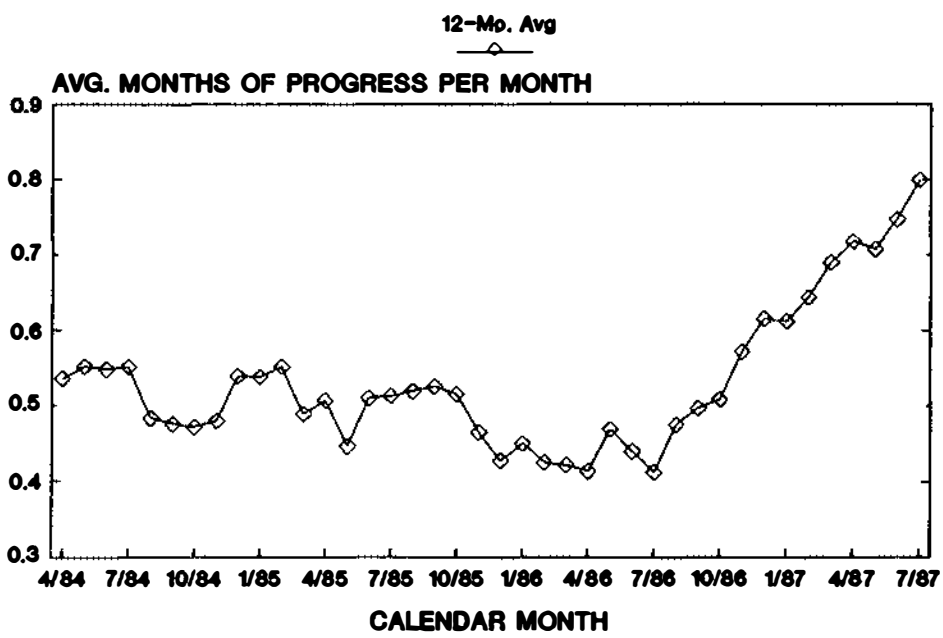


Figure 9

Though the topic of this paper is the metrics themselves, the discussion would be incomplete without a mention of the actions which have brought about substantial scheduling improvements at RND. The metrics have made scheduling accuracy much more visible to management. Management attention and emphasis on accuracy has caused project planners to devote more effort and care to scheduling with beneficial results. Project managers have received training in scheduling and estimation. A personal computer based project management program has been introduced to the project managers and is now in use in the RND lab. A task force of project managers has been formed to discuss common causes of schedule error and identify corrective measures.

While our overall schedule accuracy has improved, we feel that further gains are desirable. We have come to understand that the significant sources of delay include unforeseen tasks, changing project objectives, coordination with outside groups, and shortage of tools. Our next challenge is to reduce or eliminate these sources of delay and promote overall gains in productivity. The measures of scheduling accuracy described here will provide rapid feedback on the effectiveness of these improvements as they are made.

Acknowledgment

The author gratefully acknowledges the assistance of Gary Harmon and Rita Lasick in preparation of this paper.

References

[1] T. DeMarco, Controlling Software Projects, Yourdon Press, 1982.

MANAGING SOFTWARE DEVELOPMENT THROUGH A METRICS-DRIVEN LIFECYCLE

Cathryn Stanford, Software Engineer
Ronald C. Benton, Process Manager
Lake Stevens Instrument Division
Hewlett-Packard Company

ABSTRACT

An effective metrics program can remove much speculation involved in managing software projects throughout the development process. At the Lake Stevens Instrument Division (LSID) of Hewlett-Packard Company, a *metrics-driven lifecycle* has been developed to standardize the software development process and accommodate development in both object oriented and non object-oriented languages. Software metrics provide insight into scheduling and resource planning, and help detect process problems. The management team is able to track the progress of a project through its lifecycle and more definitively decide when the project can move into the next development phase.

BIOGRAPHICAL SKETCHES

Cathryn attended the University of Oregon. She received a BS in Computer Science and Psychology in 1983, and MS in Computer Science in 1985. Upon graduation she joined Hewlett-Packard Company's Lake Stevens Instrument Division as a software development engineer. She has worked on a variety of tools as well as on the Lab's metrics program and software product lifecycle. She is currently working on enhancements to product QA plans.

Ron graduated from Pacific Lutheran University in 1978 with a BA in Economics. In 1982 he received an MM degree from Northwestern University's Kellogg Graduate School of Management. Prior to joining Hewlett-Packard in Colorado in 1982, he worked as an analyst for a municipal government and performed market research for a major bank. Moving to Lake Stevens in 1983, he managed the Lab's administrative activities. He now manages the Lab's process group, supervising efforts in lifecycle/metrics development, software reliability, user interface design, management tools and process control.

INTRODUCTION

Historically, software development has not emphasized the use of metrics and historical data to manage projects during their development lifecycle, although tools and recommended practices have been available. Without common development processes and metrics, there is no mechanism to share metric data or learn from other project's experiences, even within the same design group. LSID's metrics program was adopted to address several problems inherent in software/firmware development:

- inaccurate schedule predictions,
- inadequate knowledge as to the current state of development, and
- feature additions after product definition.

Schedule estimates, whether by project managers, team members, or others, can be notoriously incorrect when there is no foundation of past data. This creates false expectations among those waiting for the designers to complete their tasks and may result in added pressure on the design team. The lack of knowledge about the extent of product design or construction at any point in time can have a similar effect, as do uncontrolled and unscheduled changes and additions to the product's specifications. Each of these affects not only the timely introduction of products but also the quality of decisions made and the effectiveness and morale of everyone concerned with the product.

In recognition of the potential and realized effects of these problems, LSID Lab management determined that a metrics-driven lifecycle for an object-oriented environment should be developed. "Metrics-driven" means that the measured status of a project influences decisions in scheduling, staffing, and defining its scope. A project team consisting of a manager, software designer, and statistician was created to carry out this charter. The group has grown to include a user-interface specialist and a project administrator as well as personnel working on other projects.

OBJECTIVES

The specific goals of the group were twofold. One was to change the lifecycle to fit an object-oriented environment. The other was to design and implement the new metrics program. The goals for the metrics program were to:

1. Show the progress of a project throughout different phases in its lifecycle
2. Indicate when a project could pass from one phase of the lifecycle to the next
3. Monitor any changes to the new product's definition, internal design, and code
4. Improve resource planning
5. Improve schedule planning
6. Flag any process problems that a project might have.

The metrics were intended to measure projects, products, and processes, not the individuals working on them. They were to be presented with one standard reporting mechanism each month in order to enable managers to leverage from each others' data. The collection process was to be automated and not impose on engineers.

THE METRICS-DRIVEN LIFECYCLE

An Object-Oriented Environment

Much development at LSID is done with an object-oriented language. The metrics taken are therefore discussed using some terminology applicable to that environment but all of the concepts can be applied to conventional languages as well. A *class* is the description of what is shared by a group of similar objects. An *object* is a set of organized data and the set of operations that can manipulate that data. A *method* describes how an object will perform one of its operations. For the purpose of discussing metrics, it is valid to equate a class with a standard programming language module and a method with a function.

The Lifecycle

Starting a metrics program and modifying the lifecycle had to occur simultaneously in order to get the measures that the lab needed. An outline of the major checkpoints of the revised lifecycle is shown in Figure 1. Key metrics and their status at major checkpoints are depicted.

The first lifecycle checkpoint is the Project Proposal. This is followed by the External Definition in which all features and the preliminary user interface are defined. The features of the product are then decomposed into a class list by the Module Decomposition Checkpoint. The Internal Design Checkpoint marks the completion of the design of the classes to the method level. Next, coding begins.

As each class is integrated and tested it goes through a formal release. When all classes have been released, coding is complete and the project has its Alpha Release Checkpoint. This begins the formal system level test phase (QA). During this phase resources are applied to find and fix defects. The project has its formal Production Release when release criteria are met.

Progress Metrics

Besides being good programming practice, the top-down decomposition of the product provides measurable units (first the number of classes and then methods) to serve as goals against which progress is measured. These measures also provide a means for making reasonable schedule estimates early in a product's development. At the Module Decomposition checkpoint, the project team has a reasonable estimate of the number of classes that the product will contain. Given a historical rate of classes completed per engineer month, the first educated estimates of the project schedule can be derived.

The progress metrics that are presented in the monthly metrics packet are at both the class and method level: *classes designed*, *classes tested*, and *classes released* versus *classes planned*; and *methods tested* and *methods released* versus *methods planned*. Figures 2a and b show an example of what graphs of these measures might look like for a project in its coding phase.

Progress throughout the design phase can be tracked by comparing the number of classes designed to the number of classes planned. When enough data is available to derive a rate of classes designed per engineer month for a project, one can predict the duration of the design phase. Historical rates from similar projects can also be used to make this estimate, given the number of classes planned for a project. When the number of classes designed equals the number of classes planned a project is ready to exit the design phase and enter the coding phase of its lifecycle.

During the coding phase, the classes released measure can be used to track progress in the same way that classes designed was used in the previous phase. This data does not provide timely feedback, however. While the majority of the product is coded, only a few of the classes may be released. Consequently, method level metrics are used to provide more accurate information on coding progress. The rate of methods coded per engineer month can be used to predict coding phase duration. At the end of the coding phase, classes released equals classes planned and the project enters the QA phase. A separate mechanism is used to track progress through the QA phase of a project as will be explained later.

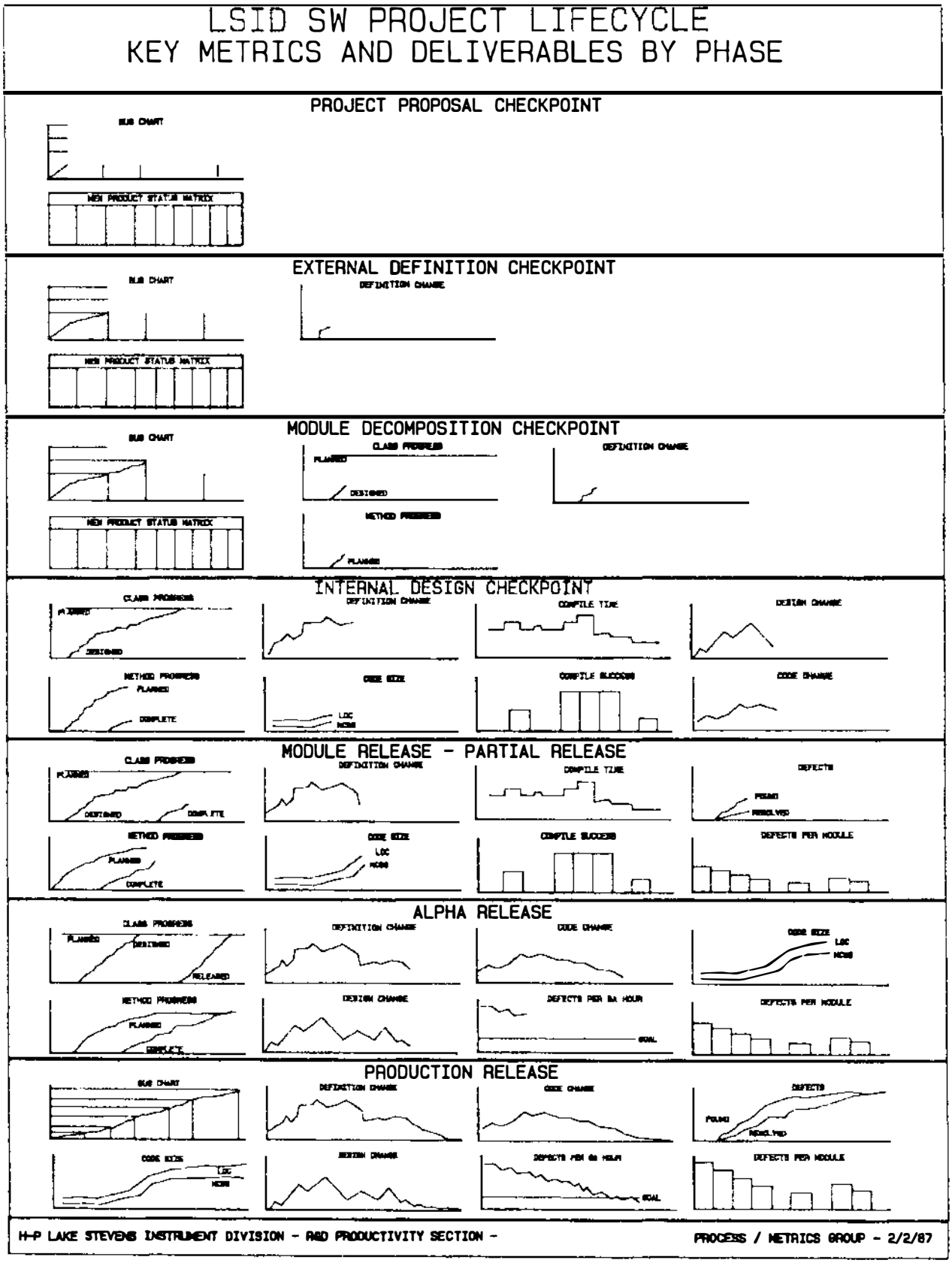
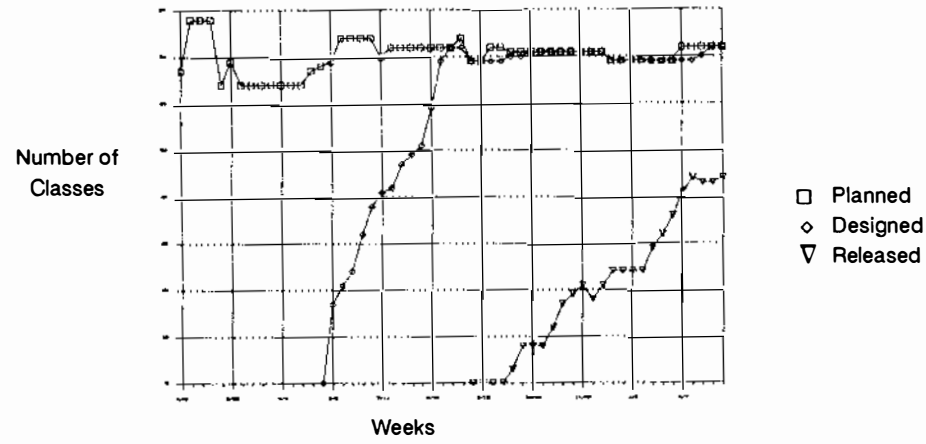
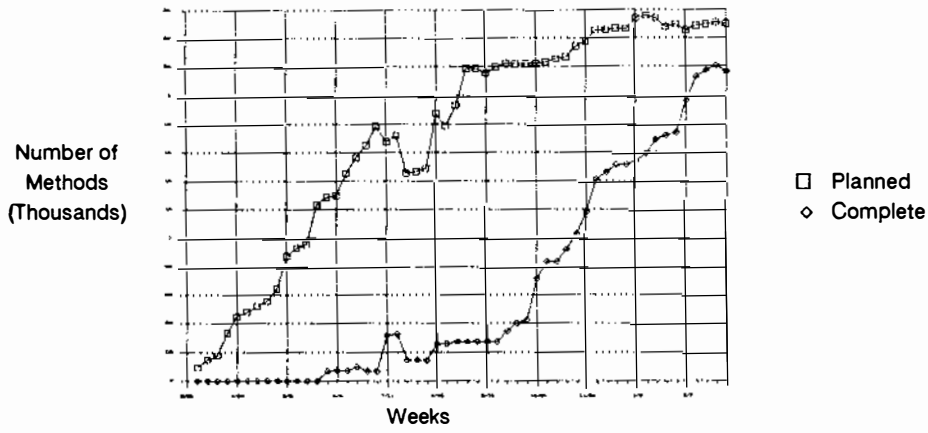


Figure 1. LSID SW Lifecycle Summary



a. Class Progress



b. Method Progress

Figure 2. Progress Metrics

A new progress metric, which may provide further insight into the coding phase, is currently being investigated. It was created by a project manager to support a schedule estimate. For this project, the number of methods coded did not look like it would converge with the number of methods planned on schedule, so he produced a graph showing that the number of methods coded would ramp up as more engineers shifted from designing or administrative tasks to coding. To avoid the "mythical man month" effect, the manager has experimented with various assumptions about the effect of incremental resource additions. The results of this investigation will be used to support future project plans. This is shown in Figure 3.

A normalized graph of the actual number of methods tested and methods coded through week 20, is plotted followed by projected schedules. The center line is the projected method tested schedule. The number of methods tested reaches 100% on the currently scheduled Alpha Release date. The slope of the line reflects a constant rate of testing completion required to reach the goal. The other two projections take into account engineer months invested in coding, changes to the number of methods planned, and the current coding rate (derived from methods coded or methods tested divided by total engineer months to date). These graphs can be used to show the impact of adding or deleting engineers from a project. They can also show the potential impact of adding or deleting features from a product given the number of methods affected.

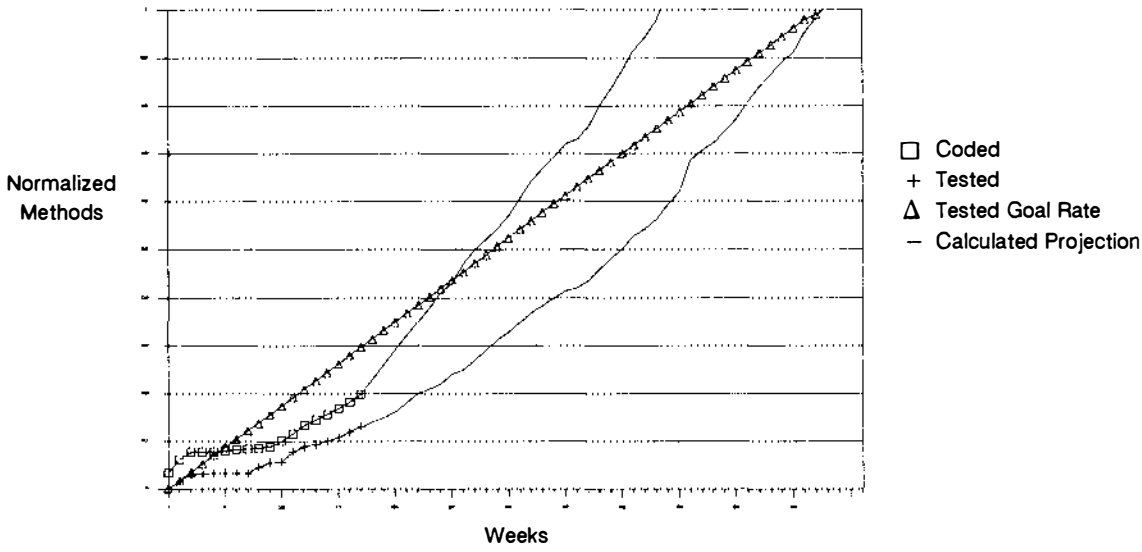


Figure 3. Schedule Projections

Project schedule, cost, and other data are published throughout all phases of the lifecycle. Some is presented in concise tabular form. Another method used illustrates the same information in graphical form and shows the history of the project. An example of the technique is shown in Figure 4 (tool and technique originated by Hewlett-Packard Company). This graph plots the dates predicted for each checkpoint and the expected release date on the X axis, and projected development expense on the Y axis. Each month the project's progress is tracked and estimates reconsidered if necessary. In the end, one can see when cost/schedule estimates were changed, when each checkpoint occurred, and the final development cost and duration.

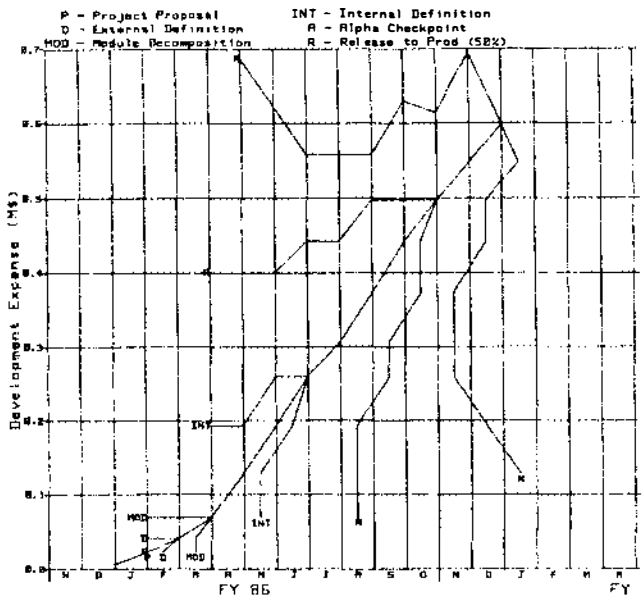


Figure 4. Project Progress Chart

For a comparison of the original versus the current schedule, a variation of the Estimating Quality Factor (EQF) metric will be used [1]. This looks at 1/EQF on an ongoing basis. The reciprocal is used to bound the range over which values appear.

Change Metrics

When planning total classes and methods early in the project, it is not intended that the specification be frozen or that necessary changes be restricted. It is important, however, to monitor changes and understand how each of them will impact the project's schedule. Experimental metrics are used to monitor changes in definition, design and code.

Definition change is the number of changes to the initial specification of the product. This is useful in determining how well the customer's needs and requests were assessed when the product was defined. Design change is the number of changes to the initial design of the code. This is intended to show how effective reviews and inspections were at finding errors in the design. The focus is on monitoring design changes that may affect other engineers. Therefore, the number of method heading changes and parameter changes per week is measured. Code change metrics yield an indication of the stability of the code. This is most useful at the end of a project's development lifecycle when one is concerned that the code should stop changing. During the QA phase it is also interesting to see what magnitude of code is affected by fixing defects. The long-term utility of change metrics is still being evaluated.

Standard Metrics

LSID continues to measure total lines of code as well as non-commented source lines. This gives an indication of the size of a product for comparison with other products and illustrates the growth of the code from week to week. The ratio of comments to code is also visible. Used with code volatility metrics, the lines of code metric indicates if the bulk of change is occurring in non-commented source lines or documentation (our current code volatility metrics do not distinguish comments from code).

Cycle time metrics are kept on all projects at LSID. We measure total time required to preprocess, compile and link. This measure indicates if the process exceeds reasonable limits. The success of the compile and link is also measured. If the compile success rate is not at an acceptable level, an investigation is triggered to identify and solve the problem.

A recent addition to the metrics set are measures of code reuse. Individual classes and methods may be flagged as new, leveraged or reused depending upon the degree to which the code under development has been extracted from existing code.

Quality Metrics

During the QA phase of a project, several defect metrics are tracked. The total number of defects fixed and unresolved per week is recorded to give an overall picture of the status of the code. Defects unresolved should reach zero as the QA phase comes to an end. The number of defects per module gives an indication of which modules have the most defects. Decisions can be made based on this information regarding where to apply defect finding resources or whether a module should be redesigned.

Application of the Goel-Okumoto model [2] has been found effective in projecting declining defect finding rates and predicting the remaining system QA duration of a project. The model uses the defect-finding rate of a project (defects found per QA hour in a given test interval) as a major factor in its calculations.

COLLECTION PROCESS

Projects at LSID are developed on networked HP 9000 Series 300 workstations with Series 300, 500, and 800 computers as system servers. Engineers design and construct code on their own systems and then check their code into a source code control system for nightly system compiles and weekly metrics collection. The aim of the metrics collection process is to be as nonintrusive as possible, but the engineers are asked to set flags in their classes and methods on an ongoing basis. The class flags are designed, coded, tested, and released. The method flags are coded, tested, and released.

Metrics are collected with a series of scripts and C programs. Each project team is asked to maintain a list of all the current files that compose their product. The scripts check these files out of the source control system. All the flags are counted as well as total lines and noncommented source statements (NCSS). Method headings are extracted with a lexical analyzer and are compared to the previous week's headings to get a count of differences. A function call is used to extract the number of lines added and deleted from a file. These are subtracted from the previous week's totals to get the change in lines for that week. A line that is modified counts as both an add and a delete.

During the QA phase of a project, an internal HP tool is used to log the status of up to 16 parameters for each defect. An automated reporting system is used to extract the weekly reports of number of defects fixed and unresolved, defects per module and defects found per QA hour. The Process group participates in the process to ensure data accuracy and provide support in applying QA models. This system is also used to monitor changes to the project's specification. These changes are entered as defects and become resolved when the project manager approves them.

The metrics collection scripts are automatically started over the weekend. The resulting metrics are electronically mailed to project managers in an overall product summary and with a breakdown by class of the total number of lines, number of source lines, number of changed lines and number of methods flagged in each status category. The product summary metrics are graphed monthly by the metrics project administrator. This is done by updating pre-created graphs stored in a spreadsheet with the new month's data points. These graphs are then plotted, compiled into the metrics packet organized by project and sent to all levels of management.

METRICS USE

Management Acceptance

One major success of the metrics driven lifecycle has been management's use of the metrics. What initially received mixed reviews (a few managers viewed metrics with indifference or as a possible infringement) has become an expected part of the management decision making process. Managers are even proposing new metrics to provide further insight into their projects. The program will not be successful if management is not willing to study the metrics, base decisions upon them, and provide feedback on their usefulness. The metrics effort at LSID has been particularly fortunate in receiving direction from the lab manager. His expectations for the use of metrics has caused project managers to be able to understand their projects in terms of their metrics.

Managers within the lab have also been very receptive to accepting changes to the lifecycle that were necessary in order to get useful metrics. The Process group is responsible for bringing all proposed lifecycle changes to lab management for discussion and approval or disapproval. The group compiles its own suggestions with those made by others in the lab and presents them up to several times a year. This ensures that the lifecycle document stays up to date without becoming so volatile that managers are unwilling to use it. The consensus made at the change proposal meetings also ensures that the lifecycle is acceptable to all sections of the lab.

Engineer Acceptance

Without the support of the engineers to keep accurate flags in their code, the metrics program could not succeed. One success of the program has been to receive this support from R&D engineers. The program was originally received with comments that the overhead on their part was not worth the effort. Several engineers expressed concern that they would be judged unfairly by metrics that looked only at quantity and not quality. These comments have subsided, however. The metrics group has had a strong position on not using metrics as a measure of individual achievement but to report only against the project as a whole. It is understood that the many dynamics that comprise a good engineer cannot all be measured and there is no motivation to do so. Managers have not abused the metrics they receive for evaluative purposes. Engineers have now requested that the metrics be posted for their general viewing. When implemented effectively, a metrics program can actually lead to improvements in the engineering environment by providing information on project progress and useful feedback on development processes.

Metrics In The Management Decision Process

LSID has monthly project review meetings between the project manager and lab/division management. The metrics packet has been timed to come out two days before these meetings so the data can be reviewed and used when appropriate. The review serves to inform upper management of the progress of a project; it also is a time to identify and resolve resource problems and to formalize any decisions that may impact the division as a whole. At this time, potential deviations to program objectives and costs/schedule are addressed.

In the past, when project teams did not have or keep effective metrics, there was no evidence to support resource, process or schedule change proposals. Using metrics, managers are now able to better justify such an action. For example, if a project manager believes the project is not meeting a schedule plan, he can look at the project's actual progress as opposed to the current schedule and demonstrate that there is a discrepancy. Actual designing and coding rates can be compared to original estimates. Potential schedule slips can be seen early in the project and can be resolved while there is still time to make an impact. Metrics can be used to compare alternate solutions to schedule slips by studying the impact of deleting features, improving tools and processes or adding resources. After the monthly management meetings, any agreed-upon changes in schedule or cost are published by the metrics project administrator.

Metrics are commonly used at major checkpoints. They are one of several deliverables discussed to determine if the project is ready to pass to the next phase of the lifecycle. At Release to Production, when maintenance of the project is passed to a new group, metrics are passed as well to show the exit status of the code. Metrics have also been used to define the scope of a project early in its lifecycle.

In one case, when market demand dictated that a new product be delivered by a specific date, the project manager was able to estimate which of the desired product features his team could complete by that date. Using historical data he determined how many classes could be completed per engineer month. From that he calculated the number of classes that his team could complete in the time available and finally he estimated the number of classes each product feature would require. This gave him an idea of what features his team could finish. By having a realistic picture of how much can be done early in the project, the project team saved time by only designing features that would be implemented and marketing was given time to determine how well the feature set meets customers' requirements.

EVOLUTION

The current monthly metrics packet is not what was originally proposed. The packet had to evolve. Measures that management did not find useful were thrown out or modified. For example, methods being coded was a measure intended to show the rate at which method coding began and the percent of methods being coded. It provided this information but was of no use to managers so it was eliminated. New metrics, such as the schedule projections metric shown in Figure 3, have been added to provide further clarification of existing metrics. Some metrics may require data from several projects before they can be useful. Without historical data, for example, it is difficult to know whether the level of design stability that a project has is acceptable. Ideally the metrics packet should consist of a stable set of metrics that are easily interpreted and understood by all managers plus an experimental set of new metrics to be tested on an ongoing basis.

When starting a new metrics program considerable time must be spent up front removing any ambiguities from the definitions of what is being measured. A measure such as methods tested or classes designed can be interpreted differently by two separate projects. One project may design classes by pseudo-coding and another by giving a two line description of what it will do. This invalidates comparisons between the project's class designing rates. At LSID it was necessary to publish a dictionary of definitions of all measures. It took several passes to reduce all ambiguities from the definitions and get a consensus from managers.

One problem encountered was in trying to apply the new metrics program to a project that was in the later stages of its coding phase. The original specification and design of the project had changed and it was too late in the project's lifecycle to require changes in its development process in order to establish realistic goal measures. The engineers on this project were correct in their assessment that flagging their code would not be beneficial. It would have been better to start metrics collection at the QA phase of their project. These later measures proved to be quite useful.

FUTURE DIRECTIONS

Future directions of the process group include the collection of productivity metrics at the project level. Engineer months per method is the currently proposed metric for the internal design and coding phases. For comparative purposes complexity measures should also be considered.

A centralized resource isolated to do schedule estimation is under consideration. The process group has been sharing available metric data with project managers to aid in estimation. They are now formally specifying the process by which schedule and resource estimates will be prepared in the future.

A long term goal for the process group is to set up a comprehensive metrics database for the lab. This would contain all the data on progress, change, cycle time, size, and defect metrics. The metrics collected each week would be automatically stored in the database so that managers could access up-to-date information at any time, as well as historical data. The transition from data collection to graph production would be automated as well.

CONCLUSION

Metrics should be used as an integrated part of a software development lifecycle. Major decisions regarding schedule, project scope and where to apply resources should be based on a substantiated knowledge of the project's status. Metrics can provide this knowledge. The metrics that are taken at Lake Stevens Instrument Division have focused on progress and change. They show how closely a project is tracking its schedule and indicate when a project can pass from one phase of the development lifecycle to the next.

The metrics program has been accepted and supported by both management and engineers. Metrics are used in monthly management meetings to adjust schedules, redefine project scopes to meet existing schedules and plan the allocation of resources. The paybacks are substantial in having a comprehensive metrics program that measures all phases of the software development lifecycle. This ultimately enables us to put needed products into the hands of customers as early as possible.

REFERENCES

Much of the work presented here is based upon a paper delivered by C. Stanford at the 1987 Hewlett-Packard Company SW Productivity Conference

[1] T. DeMarco, Controlling Software Projects, Yourdon Press, 1982.

[2] A. L. Goel, IEEE Transactions on Software Engineering, December, 1985

Session 4

CASE STUDIES

"The Copyright 'Look and Feel' Issue"
Nancy Willard, Attorney at Law

"Defining and Installing a CASE Environment"
Chuck Martiny and Steve Shellans, Tektronix, Inc.

"How to Make The System Work: A Review of a Successful Software Project"
Johnny M. Brown and Doran D. Sprecher, Schlumberger Well Services, Austin Systems Center

THE COPYRIGHT 'LOOK AND FEEL' ISSUE

NANCY E. WILLARD
Attorney at Law
296 E. 5th Avenue, Suite 302
Eugene, Oregon 97401

ABSTRACT

Recent cases pertaining to the extent of copyright protection for software programs exemplify two competing interests: the encouragement of creative effort through financial reward versus society's need to make use of the results of that creative effort. This paper reviews recent legal decisions and discusses the rationale for the decisions. The cases reviewed involve issues of infringement of screen designs and infringement of the structure, sequence and organization of software programs.

BIOGRAPHICAL INFORMATION

Nancy E. Willard is a member of the Oregon State Bar. She has a private practice in Eugene, Oregon, concentrating on computer law, technology transfer, trademarks and copyright. Ms. Willard is currently serving as President of the Eugene Software Council, an industry economic development/trade association. She has spoken frequently before software industry groups on legal issues. Ms. Willard is active in state and local economic development and is currently serving on the Business Council of the Eugene Chamber of Commerce.

THE COPYRIGHT 'LOOK AND FEEL' ISSUE

Nancy E. Willard
Attorney at Law

Two competing social interests are at the heart of any Copyright issue: the encouragement of creative effort through financial reward versus society's need to make use of the results of that creative effort. Recent cases pertaining to the extent of copyright protection for software programs and industry's response to these cases exemplify these two competing interests. On one hand, the industry recognizes the significant creative effort and development expenses which are involved in the development of software programs. On the other hand, there is a recognition that being able to freely build on prior art is essential to the development of the software industry.

While several cases have recently been decided in this area, the legal standards are just emerging. The challenge for the courts, and for the software industry, will be to distinguish between copying to achieve product compatibility or to serve market needs, versus copying in order to exploit the effort and investments of others.

This paper reviews recent decisions regarding copyright protection of software programs and discusses the rationale for the decisions. Generally, the term, 'look and feel' has been applied to recent software cases involving screen designs. Other recent cases have involved issues of infringement of the structure, sequence and organization of a program. The issues are interrelated, and this paper discusses both.

STATUTORY BASIS FOR PROTECTION OF SOFTWARE

Congress takes its power in the area of copyright from Article 1, Section 8, Clause 8 of the Constitution which provides that Congress shall have the power "(t)o promote the progress of science and the useful arts, by securing for limited times to authors and inventors the exclusive right to their respective writings and discoveries."

Patent protection is available for whoever "invents or discovers any new and useful process, machine, manufacture, or composition of matter".[1] As compared to copyright protection, patent protection is more difficult to obtain and the period of protection is more limited. However, the owner of a patent can prevent anyone else from using the patented process. Patent protection of software will not be discussed in this paper.

Copyright protection is available for literary works, musical works, dramatic works, choreographic works, pictorial, graphic and sculptural works, motion pictures and other audiovisual works, and sound recordings. Copyright law protects the rights to reproduce, distribute, modify, perform and display the works. These rights vest in the owner of the copyright.

Copyright protection is relatively simple to obtain and exists for an extensive period of time. However, copyright protection does not extend to "any idea, procedure, process, system, method of operation, concept, principle or discovery, regardless of the form in which it is described, explained, illustrated, or embodied in such work."[2] Copyright protects the expression of an idea, not the idea itself.

To establish the infringement of a copyright, the plaintiff must establish ownership of a valid copyright and copying by the defendant. Because direct evidence of copying is often unavailable, copying may be inferred where the defendant had access to the copyrighted work and the accused work is substantially similar to the copyrighted work.

Many of the early computer program cases involved video games and focused on the extent of protection of the audiovisual aspects of a program. These cases established that the audiovisual displays created by a program are protectable works under the copyright law.

The recent computer program cases have broadened the scope of the inquiry to consider the extent of the protection accorded to the structure, sequence and organization of a program and the screen designs of application programs. Two recent cases, *Whelan v. Jaslow* and *Plains v. Goodpasture*, focused on extent of copyright protection of the structure, sequence and organization of a program. Two other recent cases, *Broderbund v. Unison* and *DCA v. Softklone*, focused on the the protection screen displays. The factual settings and the court's decisions for these case will provide the basis for analysis and discussion.

RECENT CASE LAW

Structure, Sequence and Organization Cases

Whelan Associates, Inc. v. Jaslow Dental Laboratory, Inc.[3]

In *Whelan*, the plaintiff developed a dental laboratory management system in Event Driven Language (EDL) for an IBM Series 1 computer. The program had been written for defendant's use, but plaintiff retained full ownership rights. The defendant obtained a copy of the EDL source code and used it to produce a similar program in BASIC to run on IBM PCs. Although there was no direct copying of the code, the trial court found infringement of the manner "in which the program operates, controls and regulates the computer in receiving, assembling, calculating, retaining, correlating, and producing useful information either on a screen, printout or by audio communication."[4]

The appellate court upheld the trial court's decision. In doing so, the court determined that "copyright principles derived from other areas are applicable to the field of computer programs".[5] This decision enabled the court to borrow the concept of 'comprehensive literal similarity' from copyright cases involving literary works. Comprehensive literal similarity means a "similarity not just as to a particular line or paragraph or other minor segment but (that) the fundamental essence or structure of a work is duplicated in another."[6] In other words, a party can infringe the copyright of a story by copying its plot and characters, without copying the exact text. In applying this concept to computer cases, the court held that "copyright protection of computer programs may extend beyond the programs' literal code to their structure, sequence and organization."[7]

The court did not consider whether copyright extends to the computer screens because this issue was not raised by the plaintiff. Courts cannot rule on matters that have not been raised before them. The court did hold that evidence of the similarities of the computer screens was admissable in determining whether there was an infringment of the underlying program.

Plains Cooperative Association v. Goodpasture Computer Service, Inc.[8]

Plains is an agriculture cooperative that assists its members in the growing and marketing of cotton. Plains developed a computer software system for a mainframe computer which provided its members with information and accounting services. The programmers who developed the system ultimately were employed by the defendant for whom they developed a personal computer version of the system. Plains filed for a preliminary injunction, which was denied. In affirming the denial of the preliminary injunction, the appellate court noted that the defendants

had presented evidence that many of the similarities between the two programs were dictated by the needs of the market.

The court in *Whelan* had concluded that the unprotectable idea of the program was the idea of an efficient dental laboratory system and that the specific manner in which this was accomplished was the protectable expression. The *Plains* court determined that "market factors play a significant role in determining the sequence and organization" of the program and therefore, the sequence and organization were unprotectable ideas.[9]

User Interface Cases

Broderbund Software, Inc. v. Unison World, Inc.[10]

The case of *Broderbund v. Unison* involved two menu-driven programs that enable users to create customized greeting cards, signs, banners, and posters. Broderbund engaged Unison to convert its popular program, Print Shop, for use with IBM computers, but after a short time the relationship between the parties broke down. Unison continued the development of the program and released its own version, Printmaster.

The court first examined the question of whether audiovisual displays generated by an application software program were the proper subject of copyright protection. The court concluded that the "overall structure, sequence, and arrangement of the screens, text, and artwork (i.e. the audio visual displays in general) are protected under the copyright law."[11]

As to the infringment the court found that Unison had access to the copyrighted work and that the two works were substantially similar. Access to the commercially available program was sufficient because access to the code was not necessary to copy the program. In determining the substantial similarity, the court concluded that an ordinary person "could hardly avoid being struck by the eerie resemblence between the screens of the two programs. In general, the sequence of the screens and the choices presented, the layout of the screens, and the method of feedback to the user are all substantially similar."[12]

In assessing the infringement of the screen displays, the court looked to the *Whelan* decision for guidance and concluded that *Whelan* "stands for the proposition that copyright protection is not limited to the literal aspects of a computer program, but rather that it extends to the overall structure of the program, including its audiovisual displays."[13] In this regard, the *Broderbund* court overstated the holding of the *Whelan* court. This does not, in my opinion, detract from the legitimacy of the court's conclusion that screen displays or an application software program are protectable under copyright law.

Digital Communications Associates, Inc. v. Softklone Distribution Corp.[14]

DCA is owner of the rights to the communication program CROSSTALK. Softklone developed a work-alike, look-like communication program sold under the name MIRROR. The *DCA* case extended copyright protection to the compilation and arrangement of command names on the status screen, or main menu, of an application program. A copy of the printouts of the CROSSTALK and MIRROR menu screens are attached to this paper in Appendix A.

The court in *DCA* found that the software program and the screen displays are separate works and held that copying a screen display did not infringe the program. This differs from the conclusions of *Broderbund* from the practice of the Copyright Office, which views a program to be a work of authorship that includes the screen displays, the program code and the audio

component, if any. For example, when Lotus attempted to register the screen displays for its 1-2-3 program, the Copyright Office refused registration stating that the screen displays embodied within one computer program that generates them are covered by the registration for the program, without need or justification for separate registration of the display.

Although the court did not find that the program had been infringed, it did find infringement of the copyrights in the screen display. As distinguished from *Broderbund*, protection was granted in the *DCA* case to a single display screen with little or no graphic content. The *DCA* court concluded that the screen was protectable as a compilation and arrangement of the parameter and command terms of the program.[15]

LEGAL BASIS FOR DECISIONS

Copyright Law

The issues raised by these cases and the analysis made by the courts differ in many respects. As noted, the *Plains* court specifically rejected *Whelan*, the *Broderbund* court misread *Whelan* and the *DCA* court rejected *Broderbund*. What the courts are struggling with is the application of the Idea/expression dichotomy of copyright law to computer software. As noted above, copyright law does not protect ideas, it protects the expression of ideas. The distinction between ideas and expressions of ideas is not always clear. When there is apparent similarity, the question then becomes whether such similarity is due to a similarity of ideas or a similarity of expression of said ideas. The former is permitted, the latter is an infringement.

As the *Whelan* court noted, copyright principles from other areas are applicable to software cases. To better understand the parameters of the idea/expression dichotomy, the following examples are presented. It is assumed in each of these examples that the second party had access to the first party's work.

The first set of examples is in the area of visual arts. David, the jeweler, makes a pin which is in the shape of butterfly. Susie also has made a butterfly pin. Has Susie infringed on David's copyright in the butterfly pin?

If both David and Susie made biologically exact replicas of real live butterflies, there would be no infringment. This is because the expression, that is, a butterfly pin, is just the same as the idea, an actual butterfly.

If David's pin is not an exact replica because he has modified the shape and color and has added jewels and if Susie has made an substantially identical copy of David's pin, there would be infringment. But if Susie's pin is just another modified version of a real live butterfly, there is no infringment even though David may have been the first one who had the idea of making a butterfly pin.

If David has made an impressionistic butterfly pin (i.e. merely the hint of a butterfly) and if Susie has made a substantially similar pin, there would be infringment.

The second set of examples focuses on literary works.

Veronica develops and publishes a recipe for a chocolate cake. Barbara includes the recipe in her published collection of dessert recipes. There is no infringment because a recipe is an idea and the expression of that idea is the same as the idea. In other words, there is only one reasonable way to express "Add 2 cups of sugar and stir".

On the other hand, Barbara's collection of recipes would be protectable as a compilation. If Cory copied Barbara's collection and arrangement of the recipes, he would be infringing.

Nate develops and writes about a new accounting system and includes the blank forms necessary to use the system in his book. Aaron reproduces and publishes the forms. Since the forms do not themselves communicate information, then Aaron has not infringed Nate's copyright.

Steve is writing a report about the IranContra hearings. Bill is also writing a book about the hearings. The plot and characters are the same. Bill would infringe on Steve's copyright only if he made a substantially identical copy of Steve's actual text.

Lori has written a science fantasy romance set on the planet Orga in the year 2012. Tom also writes a science fantasy romance and although the names and locations are somewhat modified, Tom's book has is substantially similar to Lori's book. This would constitute an infringment.

As is evident from these examples, the extent of protection accorded a particular work is determined by the amount of expression in that work. A work that is entirely factual - a replica of a butterfly or recipe - is accorded no protection. A work that is purely an idea, such as a blank form is also not protected. A work that is in large part based on factual elements is infringed only by another work that is substantially identical. A work that is not factual has broader protection and infringment occurs if the second work is substantially similar to the first. It is necessary for the courts to determine where, on this continuum, certain cases fall.

Unfair Competition Law

In addition to understanding the idea/expression dichotomy, it is important to recognize that there are other factors which influence legal decisions. Although courts strive to base their rulings solely on the legal precedent and theory, certain issues, particularly those of faimess, often govern the outcome of a case as much as any legal precedent or theory. In the recent software cases, other claims were or could have been raised based on the common law (nonstatutory) of unfair competition. Two common law doctrines which appear to be of some unstated influence in these cases are the doctrines of 'palming off' and misappropriation. Understanding these doctrines will assist the software developer in gaining a fuller understanding of the court's decisions.

'Palming off' is an attempt by one company to make a purchaser believe that its product is that of its better known competitor. Certain physical attributes of one company's product can become the basis upon which customers or potential customers identify that product. It is generally permissable for a competitor to emulate certain functional attributes of another company's product. But if a competitor adopts the non-functional attributes of another's product, that competitor is in effect trading off of the good will of the first company. This is considered to be unfair competition.

Misappropriation is unfairly profiting from the work and investment of another. Misappropriation involves three elements: 1) The creation of a valuable 'thing' by the plaintiff through extensive time, labor, skills and/or money; 2) The appropriation by the defendant of that 'thing' at no or little cost, thereby gaining a special competitive advantage; and 3) Commercial injury to plaintiff. If the appropriation by the defendant occurred in circumstances where information was disclosed to the defendant in a confidential relationship, the claim is generally identified as misappropriation of a trade secret. Misappropriation is based on concepts of commercial ethics and morality.

DISCUSSION AND ANAYLSIS

With this background, the recent software cases can be more intelligently analyzed.

In *Whelan*, the situation was similar to the above example of the IranContra hearings. All software systems for use by dental labs will contain certain functions that are dictated by the market. The potential for creative expression is very limited. However, it appears in this case that the defendant did not merely develop a program that would meet the functional needs of the market, he "attempted to adopt in almost exact duplication all of the functions, the format of the screens, the language and abbreviations, methods of collating, the file insurance and work flow that had been designed into the (Whelan program)."[16] In addition, the defendant was closely involved with the development of Whelan's product, had improperly obtained a copy of the source code (misappropriation) and had advertised his product as "the new version of the Dentlab Computer System" (palming off).[17]

What is also significant about the *Whelan* decision is that is was decided before the decision in *Plains* which noted the importance of market factors. It does not appear from a reading of the decision that the defendant presented information or argued the importance of the market factors. The *Whelan* court did note in a footnote that "structural similarities can also arise in completely legitimate ways- e.g. where the authors of the two programs have included subroutines from common, unprotected subroutine libraries, or where the authors of both programs have consulted common, public domain, reference books."[18]

The *Plains* decision was an appeal from a denial of a motion for a preliminary injunction. To obtain a preliminary injunction, the plaintiff has a higher burden of proof than in a trial in that it must prove a substantial likelihood of success on the merits and substantial threat of irreparable injury if the injunction is not issued. The hearing for a preliminary injunction is also not as encompassing as a trial. Therefore, while the decision provides helpful insight, it was not based on a full record and does not provide a complete analysis of the situation.

The *Plains* situation resembles the *Whelan* situation, but the court's decision reveals an analysis more like the above example pertaining to the recipe. That is, if there is only one way to structure a program and that way is dictated by market factors, the first developer cannot prevent the second developer from utilizing that structure.

In *Broderbund*, the situation is similar to the modified butterfly pin example. It was necessary to convey certain ideas, instructions for use of the program, but the developer specifically focused on expressing these ideas in a artistic, aesthetic manner and the defendant copied not only the ideas, but also the aesthetic expression. Broderbund introduced into evidence another program which performed the same functions as their program, Stickybear Printer, but which had very different screen designs and user interfaces. There are also elements of palming off in *Broderbund* case. That is, a deliberate attempt to make Printmaster look and work like the Print Shop.

When comparing the *Broderbund* case to the three other cases, it is important to note that the features of this program are not as clearly driven by market factors. It is probable that in future cases, programs such as Print Shop, which have a wider latitude for development, will be given a wider range of protection than programs designed to fulfill certain market needs.

The court in *DCA* held that the menu screen was a compilation/arrangement of parameters and terms and that the defendant had copied this compilation/arrangement. This is similar to the above example of the collection of recipes. The *DCA* court took an extremely narrow view of the extent of protection of a computer program and considered that the copyright protection was limited to the program code. Having taken this position, they developed what, in the opinion of

this writer, is a rather novel approach to the protection of screen displays. It is questionable whether the court's analysis in this case will hold up.

In *DCA*, it is instructive to note that the name of the defendant, Softklone, and the name of the program, Mirror, give indications that that the defendants were involved in the direct emulation of an existing popular program (palming off).

SUGGESTIONS AND CONCLUSION

As is evident from the preceeding discussion, the legal system is facing the challenge of balancing competing interests within the software industry and determining what is fair and lawful competition. In future cases, the plaintiff's burden will be to prove that the organization, file structures, work flow, user interfaces, screen designs, etc., arose from genuine creativity and originality. The defendant's position will be that those features which the plaintiff claims are protected by copyright, are actually dictated by the requirements of the market, the hardware, the operating system, or by good programming practice.

On a practical level, it is highly recommended that software developers keep detailed development records which include daily notes about the source of particular program structures or routines and the rationale for certain development decisions. The development decisions should be based on objective market and hardware needs, not on an intent to capitalize of the work of another developer. If fitting for a particular application, incorporating design elements which are aesthetic and non-functional in the screen displays could lay the basis for wider protection of the displays.

It is always prudent to have written development contracts which provide for ownership of any work that is created. In programming situations where one party is specifying the structure and organization of a program it would be advisable to consider certain indemnity provisions. All employees and independant contractors should be bound by non-disclosure, non-use and non-compete agreements.

For those developers who are creating new user interface techniques, it would be prudent to consult with a patent attorney to determine whether the patent laws would protect the new techniques.

It probably would not be prudent to name your new company KloneWorks.

It is extremely important to recognize that this is an area of the law that is unfolding rapidly. The analysis presented in this paper could change in certain respects as the result of future decisions. It is important, at this time, to discuss all development plans with a competent computer law attorney and to request that the attorney keep you apprised of any subsequent case decisions which could affect your plans.

This is not the first time in our history that the development of new technology has forced a critical evaluation of the scope of protection granted under copyright law. The uncertainty is unfortunate, but because of the uncertainty, the software industry is being forced to determine what is fair and pr competition. The determinations made will provide the basis for establishing a strong industry for the future.

NOTES

1. 35 USC 101.
2. 17 USC 102.
3. 797 F.2d 1222 (3rd Cir. 1986), cert. den., 107 S.Ct 877 (1987).
4. Whelan, 609 F.Supp 1307, 1320, (E.D. Pa. 1985).
5. 797 F.2d at 1238.
6. Nimmer of Copyrights, at 13-20.1.
7. 797 F.2d at 1248.
8. 807 F.2d 1256 (5th Cir. 1987).
9. Id. at 1262.
10. 648 F.Supp 1127 (N.D. Cal. 1986).
11. Id. at 1135.
12. Id. at 1137.
13. Id. at 1133.
14. 659 F. Supp 449 (N.D. Ga. 1987).
15. Id. at 465.
16. 609 F.Supp at 1314.
17. 797 F.2d at 1227.
18. Id. at 1248, fn. 47.

```
                    CROSSTALK—XVI Status Screen                    Off line
NAme  Crosstalk defaults                          LOaded C:STD.XTK
NUmber                                            CApture Off

Communications parameters                         Filter settings
SPeed  1200   PArity None   DUplex  Full         DEbug  Off    LFauto  Off
DAta   8      STop  1       EMulate None         TAbex  Off    BLankex Off
POrt   1                    MOde    Call         INfilter On   OUtfiltr On

Key settings                                      SEnd control settings
ATten  Esc                  COmmand ETX (^C)     CWait  None
SWitch Home                 BReak   End          LWait  None

List of Crosstalk commands

  NAme     NUmber   ANswback  APrefix  ATten    BReak    DEbug
  DPrefix  DRive    DSuffix   EDit     EMulate  EPath    FIlter
  POrt     PWord    RDials    RQest    SCreen   SNapshot SWitch
  TImer    TUrnarnd Video     ACcept   CWait    DNames   FKeys
  GO       INfilter LFauto    LOad     LWait    MOde     QUit
  RUn      SAve     SEnd      XDos     BKsize   BLankex  BYe
  CApture  CDir     COmmand   CStatus  DAta     DIr      DO

More to come . . .   Press ENTER: _

  DUplex   ERase    FLow      GKermit  HAndshak HElp     KErmit
  List     NO       OUtfiltr  PArity   Picture  PMode    PRinter
  RCve     RKermit  RXmodem   SPeed    STop     TAbex    TYpe
  UConly   WRite    XKermit   XMit     XXmodem

For more information on a command, type "help xx" where "xx" is the command name (for example,
"help LO" for information on the LOad command).
If you need more general help, type "help general" or "help call".

Command?
```

```
                    MIRROR Status Screen                           Off-line
NAme  MIRROR Default Settings                     LOaded STD.XTK
NUmber                                            CApture Off

Communications parameters                         Filter Settings
SPeed  1200   PArity None   DUplex  Full         DEbug  Off    LFauto  Off
DAta   8      STop  1       EMulate None         TAbex  Off    BLankex Off
POrt   1                    MOde    Call         INfilter On   OUtfiltr On

Key settings                                      SEnd control settings
ATten  Esc                  COmmand ETX (^C)     CWait  None
SWitch Home                 BReak   End          LWait  None

List of available MIRROR commands

  ABort    ACcept   ALarm    ANswback APrefix  ASk      ATten
  BAckgrnd BKsize   BLankex  BReak    BYe      CApture  CDir
  CLear    COmmand  CRc      CStatus  CWait    DAta_bits DEbug
  DIr      DNames   DO       DPrefix  DRive    DSuffix  DUplex
  EDit     EMulate  EPath    ERase    FIlter   FKeys    FLow
  GKermit  GO       HAndshake HElp    IF       INfilter JUmp
  KErmit   LAbel    LFeed    List     LOad     LWait    MEssage

More to come . . . press RETURN:

  RDial    REply    RHayes   RKermit  RQuest   RUn      RWind
  RXmodem  SAve     SBreak   SCreen   SEnd     SKip     SNapshot
  SPeed    STop_bits SWitch  TAbexpand TImer   TUrnarnd TYpe
  UC_only  WAit     WHen     WRite    XBatch   XDos     XHayes
  XKermit  XMit     XXmodem

For HElp on a particular command, enter HE followed by the command name (e.g., HE LW for HElp
on the LWait command).

Command?
```

DEFINING AND INSTALLING A CASE ENVIRONMENT

Chuck Martiny
Steve Shellans

Software Center
Computer Science Center
Tektronix, Inc.

ABSTRACT

When a large high-risk project was terminated before completion, a post-project review was conducted by the authors to determine the causes. Based on the lessons learned, an all-new CASE environment was installed that addressed both 'people' issues and 'process' issues. The specific goals were:

- Improve Software Engineering Productivity
- Improve Product Quality
- Reduce time-to-market
- Improve project estimates & schedules
- Foster professional development and personal satisfaction

The tangible outcomes were:

- A software policy manual
- A well-defined software development process
- A productive software development environment
- A 'deliverables' library
- A metrics collection system

The intangible outcomes consisted of a cohesive team environment, a winning vision, and a mental commitment to quality software.

BIOGRAPHIES

Chuck Martiny

Mr. Martiny has a B.S. degree in Electrical Engineering from the University of Arizona. In addition to five years as an electrical engineer and six years as a software engineer, he has managed software development activities at Computer Sciences Corporation and Tektronix for 17 years. Mr. Martiny is currently manager of the Software Center at Tektronix.

Together with Ken Oar of Hewlett Packard, he was a co-founder of the Pacific Northwest Quality Conference and the Chair of the second conference.

Steve Shellans

Mr. Shellans has a B.S. degree from MIT in Engineering Administration, and an M.S. in Computer Science from Stevens Institute of Technology.

Most of his career has been spent in the management of software activities. Currently, he is an internal consultant for Tektronix, assisting product groups to achieve quality software in a productive manner. Prior to that, he was Software Engineering Manager in one of the divisions of Tektronix.

Mr. Shellans is this year's Conference Chair for the Pacific Northwest Software Quality Conference, and has played an active role with this organization since its inception.

Both authors can be reached at:

Tektronix M/S 19-245
Beaverton, OR 97077
(503) 627-6834

INTRODUCTION

In order to succeed in the fast-moving world of software development, we must often take risks. Sometimes we are not as successful as we had initially expected. Good managers, however, always learn from their experiences, either good or bad, and constantly strive to repeat their successes while correcting their mistakes. This paper describes how the cancellation of a high-risk project provided the impetus for an analysis of the problems that existed, and the subsequent correction of those problems through the installation of a productive CASE environment.

The project referred to was cancelled by management when the existence of serious problems was recognized. It was their judgement that remedial measures would not be cost effective. Immediately thereafter, the management requested a Post Project Review, to be conducted by the authors who serve Tektronix as internal software management consultants.

The purpose of the Post Project Review was to analyze the conduct of the project from inception to termination. Specific issues included what the project team had planned to do, what they actually did, what went well, what did not go well, the causes of their successes and failures, and the consequences of their decisions and actions.

The information was documented and the team members, including management, learned a great deal from it. The positive outcome of this analysis was that management made the decision to make the necessary investment in a software engineering environment that would take advantage of the information gained from the Post Project Review. At the completion of this process, they were positioned to be very successful.

This environment was designed to address two major issues. The issues with probably the greatest impact on the project, and certainly the most difficult

to address, were the ones related to the people. These include such things as attitude, skills, turnover, and matching people with the right assignment. Other issues related to the process. They included such things as the methods, tools, standards and policies that must be in place.

Collectively, these two sets of issues were the basis for what is referred to in this paper as "The Environment". In most cases, if the environment is 'right', it not only makes the team more productive and improves the quality of the product, it also becomes easier to attract and retain the desired staff. Software engineers, like everyone else, take a great deal of pride in getting a high quality product to market on schedule and within budget.

The first half of this paper describes the process of defining the software engineering environment from the "lessons learned" during the post-project review. The second half describes the components of the environment.

THE POST-PROJECT REVIEW

This process goes by several names. It is sometimes called a post-mortem, a post-partum or a retrospective analysis. What it is called is not important--what ***is*** important is that it is a process for developing an understanding of the dynamics of a project.

The authors initiated the concept of post-project reviews in 1981. Since then the concept has become widely accepted throughout Tektronix. The process of reviewing and documenting a project takes from a few hours to a few days, depending on the size and nature of the project. Some reviews are done informally and coordinated by the project leader; others are more structured and are facilitated by a third party from outside the project team. Due to the sensitivity and size of this particular project, the review was quite structured and required several days.

The significant issues that were raised in the post-project review are implicit in the following objectives that were established.

OBJECTIVES

Software Engineering Productivity

Although the productivity of the engineers had not been measured, it was agreed that improvements could be made, and metrics established so the improvements could be measured. Several tradeoffs had to be considered. They included the amount of time available before the next project was scheduled to begin, the cost of development, the amount of change that could be accommodated at one time, and the general process of causing the desired change to come about.

Participation

The intent was to have the people who would be affected by the environment participate in its definition and implementation. This would instill a feeling of ownership and could be expected to result in a better environment since one of the key factors in productivity is the attitude of the team members. It would also reduce the time required for implementation by spreading the effort over several people.

Quality

Both the quality of the process and the quality of the product were to be improved. Tektronix has always been known for the quality of its products but as the product mix moves from hardware to more software, sustaining engineering is becoming more of a concern. One of the objectives was to design-in high quality and spend less effort in rework. The quality would be measured by the cost of sustaining engineering.

Time to Market

Getting quality products to market faster is a major concern of almost all companies in the United States. One of the goals of our company is to reduce the time to market by 50% in three years, and one of the objectives of this environment program was to help meet that goal.

Project Estimates and Schedules

Some of the divisions in our company can consistently maintain their schedules very closely. The objective of the division under study was to consistently meet their schedules within 15%. This required better estimating techniques, and better project tracking and control.

Professional Development and Personal Satisfaction

All engineers want the satisfaction of doing challenging work, and producing quality products on schedule. They also have certain career goals that they want to achieve. The intent was to develop an environment that would support both objectives.

PROCESS

There are several ways to bring about a desired change. It is important that the method selected be consistent with the corporate and division culture as well as with the style of the implementors. In this case, it was important that everyone affected by the environment feel ownership for it. Therefore, several people in the division were brought into the development process and almost everyone in the division was brought into the review process. This style of definition phase takes longer than a "Top Down" approach, but in most cases the implementation takes less time and the results are far superior.

Nine people were selected by the division management to help develop and champion the new environment. They were involved in establishing the overall objectives as well as actually writing some of the documents to be used. In those cases where they were not involved in writing the documents, they reviewed the original drafts and made suggestions for changes to fit their specific needs.

After the core committee members were satisfied with a component of the environment, it was taken to a larger group for review. In some cases there were several reviews by different groups, such as hardware engineering, marketing, manufacturing, service support, manuals and evaluation. This required a substantial number of iterations for tuning and modification, but when the environment was complete, everyone felt ownership for it and they were ready to embrace it.

COMPONENTS OF THE ENVIRONMENT

Policy Manual

This manual defines those policies that the software engineers had determined to be desirable for their division. The manual contains 22 sections covering such things as project management, intra-project communication, training, the concept of mentors, the role of evaluation, and post project reviews.

Because of the process used to develop this manual, everyone affected by it agreed that the contents made sense; it would help meet the objectives of the environment; it would make them more productive; and their job would be easier.

Process Manual

Developing this manual was by far the most time-consuming of all of the components of the environment. The Process Manual describes and gives an example of every 'deliverable' to be produced during the software development process. (A 'deliverable' is something concrete and unambiguous that provides objective evidence that a certain phase/milestone/task has been successfully accomplished.)

The manual is organized chronologically by project phase. It stresses clean milestones which helps assure that potential problems are eliminated early in the project. The manual contained GUIDELINES, not policies. The original draft was developed by combining the best ideas and guidelines from several of the other divisions. It was then tuned and modified with input from key people in the division for which it was developed. In all, there were seven major drafts with hundreds of changes. When the manual was complete, no unresolved conflicts remained.

In order to meet the objectives of the environment, the manual stressed the need for the hardware and software developers to communicate and coordinate with each other. It had individual deliverables and milestones for each, as well as for the overall project.

Deliverables Library

The deliverables library contains about 75 documents. Most are examples of the deliverables called for in the Process Manual. These are such things as Product Proposals, Project Management Plans and other documents that are considered to be essential for project success. This library of documents is located in the site library and maintained by division personnel.

Methods and Tools

A recommended set of methods and tools was identified. The toolset included both project management tools and development tools. The development tools were C/UNIX based, and both VAX and Gould hardware were used. The project management tools included project estimating, planning, and tracking. Each development phase was analyzed and the appropriate tools for that phase were identified. These included tools for checking C code for style and hidden problems not caught by the compiler; coding standards; tools for source code control; tools for bug tracking; documentation tools; debugging tools; and tools for intra-project communication.

Training

Training needs were identified by circulating a questionnaire and consulting with managers. Whenever possible, the training was provided by experts among the division personnel. In such cases, a brief overview was given, a manual provided and an assignment given. The expert was then available to the trainees for ad hoc consulting, and this was a recognized component of his/her job. This process was used when the tool or method used was well documented and had a good user interface. In other cases, however, outside sources of training were required.

Metrics

As mentioned before, no productivity or quality measurements had been collected before this study, so productivity gains could not be immediately measured. However, metrics-collection procedures were put in place so that productivity and quality improvements can be measured in the future. These were simple and whenever possible, were transparent to the project team. In general, they consisted of maintaining records of how much effort was used in each phase, how many bugs were found, when the bugs were introduced, and how much code was produced. Effort numbers were collected by defining appropriate job codes (against which time could be charged) in the standard time-reporting system. Quality information was collected by writing scripts to monitor the revision control system.

Archiving

Policies were established for archiving the product and the tools required to reproduce it, should it ever become lost or destroyed.

RESULTS

The first project to use the new environment is not far enough along to measure the success of the environment. In fact, the total success cannot be measured until the second project is completed and improvements can be measured. It is expected that small gains will be realized on the first project because of the learning process that must take place. Significant gains will come during the second project. The greatest gains on the first project will be realized in the sustaining engineering activity, where a 50% reduction in effort is expected due to the increased quality of the product.

The most significant results so far are not directly measurable -- they are the changes that have occurred in the people. Morale is much higher. Everyone is eager to use the new environment they helped to build, and through that, to succeed.

SUMMARY

A canceled project provided the impetus for defining and installing a modern CASE environment. An enlightened management initiated a post-project review to understand the problems and correct them. The definition of an appropriate environment was a participatory effort. This engendered a sense of ownership and buy-in among division personnel.

The major components of the new software development environment are a policy manual, a process manual, a deliverables library, a set of methods, management tools and development tools, appropriate training, and a metrics collection system.

The environment is now in place, morale is high, and significant productivity and quality gains are expected.

How to Make the System Work – A Review of a Successful Software Project

Austin Systems Center
Johnny M. Brown and Doran D. Sprecher

Abstract

A high level of productivity can be achieved when good project management is combined with a relevant development model for a well-defined software project. But getting all these factors together, particularly the latter, is usually the problem for any group of software developers. We will show how this was achieved for a signal processing package (SPP) distributed to our worldwide field units in June, 1986.

The SPP consisted of seven modules containing 30,000 lines of new FORTRAN code along with over 50,000 lines of internal and external documentation. It was integrated into a software system containing over 2,000,000 lines of code that was already in place at each field unit. Average productivity for the SPP software designers was over 35 *new* lines of code per day, with less than 0.5 defects per 1,000 lines of this code reported by users during the year following the introduction of the SPP. Producing software with few defects was possible at this rate only by requiring agreement on specifications before 'final form' code was written and by fostering continuous communications between members of the software development team throughout the life of the project.

Biographical Sketches

Johnny is a senior project engineer with Schlumberger Well Services in Austin, Texas. He is currently adding enhancements to prototype applications software that is in field test within Schlumberger. He is also acquiring and maintaining the computer hardware used within his department to develop AI applications software. Prior to his transfer from the company's engineering center in Houston, Johnny was the manager for a section consisting of a real-time acquisition software development group and an applications software development group. He holds an AB degree in physics from the University of California at Berkeley, MA and PhD degrees in theoretical nuclear physics from Rice University, and a MBA degree from Houston Baptist University.

Doran is also a senior project engineer with Schlumberger Well Services in Austin, Texas. He is currently writing AI applications software related to automatic programming. Prior to his transfer from the company's engineering center in Houston, Doran was a software project leader and developed signal processing applications software. He holds a BS degree in physics and mathematics from Pan American University as well as a MS degree in systems engineering from the University of Houston.

Johnny M. Brown
P.O. Box 200015
Austin, Texas 78720-0015
(512) 331-3279
jbrown%asc%ascway%spar@SPAR-20.ARPA

Doran D. Sprecher
P.O. Box 200015
Austin, Texas 78720-0015
(512) 331-3283
sprecher%asc%ascway%spar@SPAR-20.ARPA

Introduction

This paper will describe a software project that successfully met the requirements of the project designers, the software engineers, the project management and, most importantly, the users of the final product. Although the technical nature of various modules is in itself of interest, the focus of the discussion will, instead, be to describe the development model used for the project and to present information generated by reviewing the project from the viewpoint of this model.

The body of the paper will begin with a short discussion of the development model followed by a brief presentation of the product, the signal processing package (SPP), along with some information about the environment in which the project was conducted. Various phases of the project for which data were gathered are also presented along with some discussion of these data. The importance of the specification phase is discussed, as is the need for good communications between the software engineers and others throughout the project. The testing phase of the project is covered in some detail.

Various standard metrics are used to present some of the project data in a form more amenable to discussion. These metrics, especially when used in combination, demonstrate the high productivity of the software engineers and indicate why a low rate of defects was predicted for the various modules. That this prediction agreed with subsequent observations by users demonstrates the usefulness of the development model and of the combined metrics.

The Software Development Model

A software development model was created by the authors in 1985 to assist in the effective production at our Houston engineering center of increasingly complex and interrelated software packages. The model itself is discussed in Appendix A. Of the four phases in developing software (product definition, software specifications, implementation and user testing) addressed by the model, this paper will concentrate primarily on the second and third phases.

The development of each module in the SSP and the structure of the package as a whole was managed very effectively with the use of this model. The model itself is based on current industry recommendations for software development models, on documentation and coding standards already in place at the various Schlumberger engineering centers, and on the authors' 25 years of experience in producing commercial software.

Finalizing the software specifications (external specifications plus internal designs) before generating code is an essential requirement if a software project is going to be a success. The external specifications must meet the requirements of both the product designers and the users of the final product. The internal design of each module must be approved by all the software engineers involved in the project. The involvement by whomever manages the overall commercial software environment is also highly recommended.

Preliminary schedules for software implementation, testing and release can be estimated during the specification phase, but management must be willing to accept schedule revisions as requirements are better defined. Go or no-go decisions regarding the project can be made with the use of these schedules if resources or time for completion are critical items (which seems to be the norm for commercial software projects).

Once specified, the implementation of the various modules begins. Schedules for each module should be detailed by now to at least the weekly activity level. Any changes to the schedules must be reviewed with all parties involved in the project before they are permitted. The addition of requirements that the users or project designers seemingly cannot live without are especially dangerous. At this time, if new requirements are really necessary, all parties must formally agree to the resulting changes in the schedule, with specific reasons for the changes well advertised. This agreement is particularly important for modules that are on the critical path. Of course, removal (or the delayed implementation) of existing requirements must also be agreed on by all parties before changing the specifications.

After the code and its supporting documentation are complete enough to test out higher level code, the software engineer can begin his testing. If all the required functionality can be demonstrated to work on at least one set of data, then integrated testing begins. Representatives of the user community may be involved at this stage (and were during the SSP development); however, they will probably not be involved until after integrated testing of all the modules in the package is either complete or almost complete. User testing then begins, with testing continuing either for a minimum period of time (say six months) or until the users are satisfied with the performance of the package. Assuming the specification and implementation phases were properly performed, the software engineer will become increasingly less involved as the testing phase continues.

General Application For The SPP

Schlumberger Well Services is in the business of well logging. That is, measuring (logging) characteristics of geological formations and the fluids they may contain from deep holes in the ground (wells) which have been drilled in the hope of producing hydrocarbons (oil and gas). These measurements are provided at regular depth increments (usually every two or six inches) while pulling a cylindrical, metal-encased electronic instrument package through the borehole of a well. Real-time transmission of the measurements to surface computers for storage, inspection or manipulation is done with the use of the armored electrical cable to which the instrument package is attached.

Schlumberger uses a variety of these cylindrical instrument packages, known simply as "tools" in the industry. These tools are about 4 inches in diameter and 8 to 15 feet in length. One such tool used by Schlumberger transmits and receives sonic signals. The current sonic tool in commercial use transmits and receives 8 signals (waveforms) at either two or six inch depth increments for the depth interval of interest. Each waveform contains 512 values. Figure 1 shows a sample set of waveforms recorded with eight transmitter/receiver pairs over a short depth interval in a well.

Once the waveforms have been recorded, certain acoustic arrival times must be ascertained from them. In particular, the compressional, shear and Stoneley acoustic arrivals are determined. The compressional arrival is used to compute the porosity in the formation (the percent of space filled with fluids – water, oil or gas). The compressional arrival, combined with the shear arrival, is used to compute the strength of the formation (*e.g.*, the ability to hold up under the extreme downhole pressures). The shear arrival is also used in determining the location of fractures in the formations. The Stoneley arrival has great potential for predicting how easily fluids will flow through the formation and thus be produced.

These pieces of information, when combined with information obtained from other tools, are

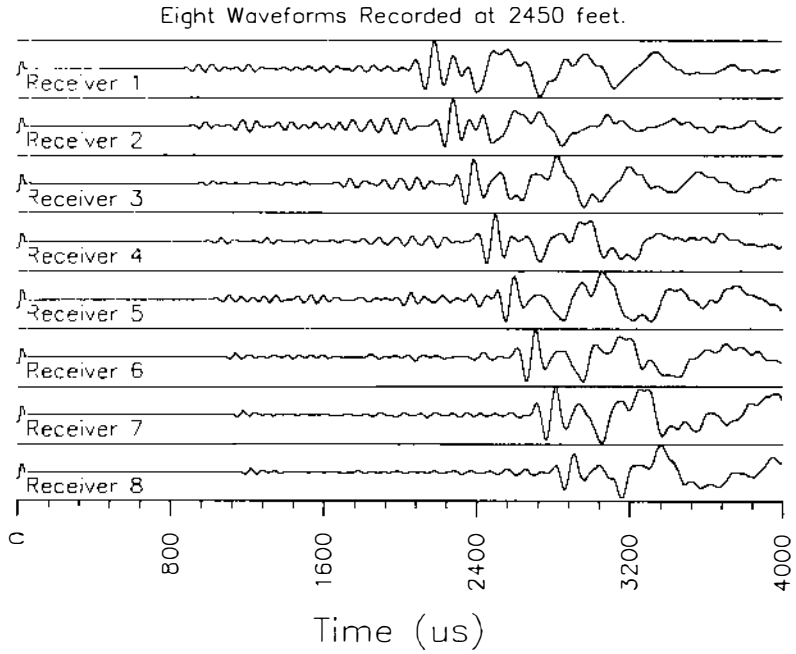


Figure 1: **A set of waveform data prior to processing by the SPP.**

critical in determining whether it's worth spending considerably more money than has already been spent to prepare a well for commercial production.

The Purpose Of The SPP

The primary goal of the SPP software discussed in this paper is to compute the compressional, shear and Stoneley arrivals from the sonic waveforms recorded with the use of the tool discussed above. The sonic data acquired at the well site are transferred to one of Schlumberger's field locations (called a FLIC) for processing by this software.

At the core of the SPP are two programs – one to detect the acoustic arrivals, the other to classify them by type. Other modules were developed to satisfy the requirements needed to make the process a commercial one. These included programs to normalize the waveforms, compute waveform energies, apply general signal processing calculations to the waveforms, and display the results. Figure 2 shows one example of the results produced by the SPP.

The software was developed and used commercially on Digital Equipment Company's VAX [1] series of computers, with much of the actual signal processing performed on an attached array processor. Most programming was done using FORTRAN for both the VAX host computer and the array processor.

A normal job processed at a FLIC contains data collected from about 4,000 feet in the well and takes 2 to 3 hours of elapsed time to compute. The large amount of processing time used is due to the number of data elements computed (3.3×10^7 elements input for 4,000 feet), and the substantial signal processing needed to produce the arrivals.

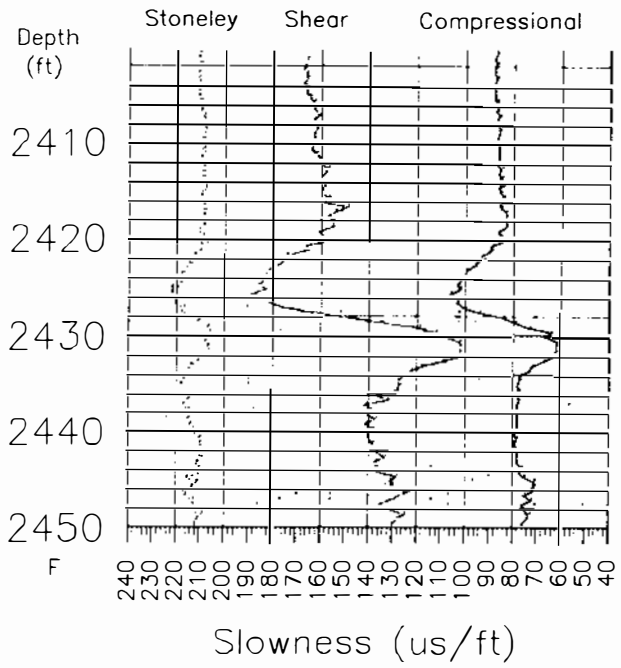


Figure 2: **Results of processing waveform data by the SPP.**

The Software Environment For The SPP

The software system in which the modules discussed in this paper were to be integrated consisted of over 2,000,000 lines of FORTRAN code alone by mid-1985. The roughly 500 modules in this system, along with their documentation and control files, are distributed to the FLICs as commercial baselines. All modules in a commercial baseline store identifiers for parameters and data in a common data base. Access to data and parameter values via these identifiers is performed by the modules with the use of a standard set of data base querying subroutines.

Commercial baselines are updated and released to the engineering centers and to the FLICs by a group at the Austin Systems Center of Schlumberger Well Services. Only software modules that are judged to be commercial are allowed into the baseline (this judgement made by the users). To permit the addition of new software packages into commercial use, 'incremental' releases of baselines are made every four to six months for incorporation into 'major' baselines. Major baseline releases are made whenever sufficient changes have been made to the underlying operating system, the data base, or the full set of modules. Regression testing is performed by the group responsible for the baselines on all software before a baseline is distributed. Rejection of software modules is possible at this stage by either the users or the group performing the regression tests.

To permit testing of new software, experimental baselines can be produced at any engineering center for distribution to the other engineering centers as well as to any or all of the FLICs. The baseline administrators at the receiving sites can place the experimental baseline on their systems at their discretion. With a simple one line command, a user at a FLIC can then logically add an experimental baseline onto the commercial baseline to access the modules contained in both for his or her data processing needs. The SPP was in an experimental baseline from June, 1986 though June, 1987, with its incorporation into a commercial baseline occurring during the summer of 1987.

Statistics Analyzed For The SPP Development Effort

The statistics on which we will base our analysis of the SPP development effort are summarized in Table 1. The contents of this table are described as follows:

	SM1	SM2	SM3	SM4	SM5	SM6	SM7	SUM
Ext Specs	2.6	10.8	2.8	0.2	1.4	11.6	3.0	32.4
+ overhead	3.6	13.1	3.5	0.3	2.8	13.9	3.9	41.1
Int Design	1.8	4.2	1.0	0.1	5.0	2.2	1.6	15.9
+ overhead	2.2	5.2	1.3	nc	5.6	3.2	2.0	19.6
Code	3.4	11.8	3.3	0.5	5.1	9.7	3.7	37.5
+ overhead	3.7	12.5	3.5	nc	5.5	10.4	4.0	40.0
ϕ_1 Test	1.4	3.5	2.0	0.1	3.3	4.0	1.3	15.6
+ overhead	no measurable overhead for this activity							nc
Integration	0.6	0.2	0.2	0.0	0.6	0.2	0.2	2.0
+ overhead	no measurable overhead for this activity							nc
ϕ_2 Test	2.4	0.9	0.6	0.1	2.8	3.5	2.0	12.3
+ overhead	3.9	4.6	1.8	0.3	4.9	7.2	3.4	26.0
Audit/Release	1.4	1.8	0.8	0.5	1.1	2.0	1.1	8.7
+ overhead	1.7	2.5	1.0	nc	1.5	2.7	1.4	11.2
Total Weeks	13.6	33.2	10.7	1.5	19.3	33.2	12.9	124.4
+ Overhead	17.0	41.5	13.4	1.9	24.1	41.5	16.1	155.5
Lines of Code	1,612	10,987	1,236	372	2,673	10,666	1,399	28,945
Lines Int Doc	1,820	3,367	2,885	439	1,906	20,729	829	31,975
Lines Ext Doc	3,200	2,600	2,240	560	1,520	7,800	1,440	19,360
Total Lines	6,632	16,954	6,361	1,371	6,099	39,195	3,668	80,280
Units	44	107	33	13	47	225	21	490

Table 1: SSP Project Data By Module And Summed

- The first seven rows of data in Table 1 give the time in weeks of effort (not calendar weeks) for each of the activities indicated. The entries are described as follows:

  Ext Specs – The gathering of information from product designers, potential users and other software engineers about the requirements for each module. The time used in reviewing these requirements and in developing the first drafts of the external documentation based on these requirements is included here.

  Int Design – The design of the internal structure of each module and the review of these structures to ensure they meet the requirements imposed by our design standards.

  Code – The writing of the code for each module. Only the time spent in writing FORTRAN code is given here for each module, whether it is for use on the VAX or on the array processor.

  ϕ_1 Test – Phase I testing of each module by the software engineer is done to verify that the internal structure works as expected. This includes testing of the module with

various combinations of requested inputs and outputs to ensure most of the internal logic paths are exercised.

Integration – All the software modules are put into one directory and a test job is run through them to ensure that their interfaces are working properly. All the parameter and data identifiers are placed into a single data base at this time as are all diagnostic messages which may be output to the user during a job.

ϕ_2 **Test** – During Phase II testing product designers, possible users and others outside the software engineering group try out the package. Any problems found with the code or documentation are corrected within 24 hours by the responsible software engineer.

Audit/Release – The procedure used to generate each module and the module itself are audited by a software engineer uninvolved in creating the module. The MAT [2] procedure must be run by at least this stage in testing to detect such problems in source code as can be found with static checks. Problems encountered are corrected prior to putting the package on a single experimental baseline release tape. A test job is run through the package and is included as part of the release.

- The total weeks required for the development of each software module and for each activity are given in the indicated row (**Total Weeks**) and column (SUM). **Overhead** accounts for the time spent by product designers and management on the SPP during its development. This time was estimated to be 25% of the total time spent by the software engineers on the project. The breakdown of this time by activity is given in Table 2. Notice that 'nc' is used in Table 1 to show where overhead resulted in no change to the data within the precision indicated. Also note that the fraction of overhead allocated to an activity is determined as being the percentage of overhead assigned to that activity times the total overhead for the software module.

Overhead for the SPP	Management	Product Designer
External Specifications	3	4
Internal Design	2	1
Code	1	1
Phase I Test	0	0
Integration	0	0
Phase II Test	2	9
Audit/Release	2	0
% Of Total Software Engineer's Time	10	15

Table 2: Management And Product Designer Related Overhead

- The lines of code are restricted to FORTRAN code only. A line of code is defined to be any line in a subroutine or main routine that is not preceded by a 'C' and is not a blank line. INCLUDE files are included in the lines of code count for each module, but are counted only once. Code written external to the modules to test them out is not included. Code which was added or modified by the software engineers within vendor or Schlumberger proprietary libraries is not included. Since all modules were written essentially from scratch, reused

code is not included in the count. As is evident, a rather conservative approach has been taken in defining what a line of code amounts to for this project.

A line of internal documentation is any line within a subroutine or main routine that begins with a 'C' and contains additional characters on that line. That is, a non-blank comment line. The lines of external documentation were estimated by setting a page of external documentation equivalent to 40 lines of internal documentation.

- **Total Lines** are the simple sums of the preceding three rows of line data in Table 1.
- Units are the main routines, subroutines and function routines written in FORTRAN for each of the software modules. These units do not include any routines which were added or modified by software engineers within libraries that are referenced by the modules.

Specification Phase Information For The SPP

The activities given in the first seven rows of Table 1 come directly from the development model. Preliminary schedules for each activity were entered into the TELLAPLAN [3] based project management system. Planned start times of each activity were used as checkpoints to review progress made on the project and to fine-tune estimated times needed for remaining activities.

With the project management system and the development model used, the development team was able to control the project rather than letting it control them. Changes in schedules did occur, but were made in a carefully controlled manner. All engineers were aware of their module's status with respect to that of other modules and the package as a whole.

Figure 3 displays activity information derived from Table 1. The time spent in establishing

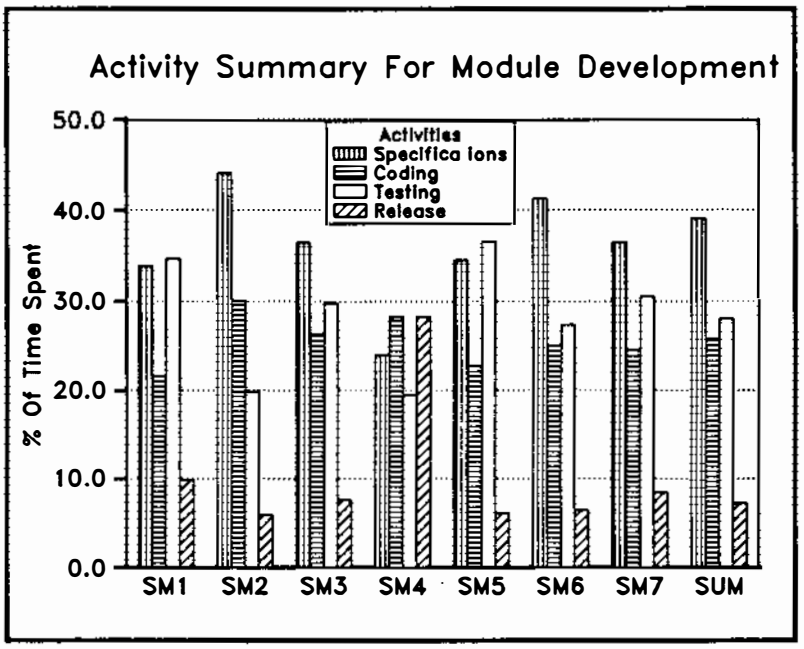


Figure 3: **Comparison of time spent on various module development activities.**

specifications (external specifications plus internal design) is clearly greater than the time used in

any of the other activities. For the largest modules (SM2 and SM6), the percentage of specification time is greatest; by contrast, for the smallest module (SM4), the time spent on specifications is relatively small.

The time spent in defining the requirements prior to coding was crucial to the high productivity shown in developing this package. Rewriting of code was minimized, retesting of the modules due to logic flaws or missing requirements was reduced, revisions to documentation were minor and auditing revealed few problems. Much of this success was due to using a well defined process, namely the development model, during the SPP project. The time spent up front in determining requirements more than paid for itself with savings later in the development period and with the quality of the product sent to the field.

For the successful development of software packages, the need to spend time defining specifications before coding is clear. When scheduling software for commercial use, it is important that plenty of time be allocated for this activity. As is evident from Figure 3, about 40% of the time was spent on the SPP in determining specifications. Note that *this assumes the requirements are already known by the product designers*, so the software engineers need only to determine how to translate them into software.

The time spent by a software engineer in interacting with others on a project does not stop after the external specifications are determined. As Figure 4 shows, the time an individual spends

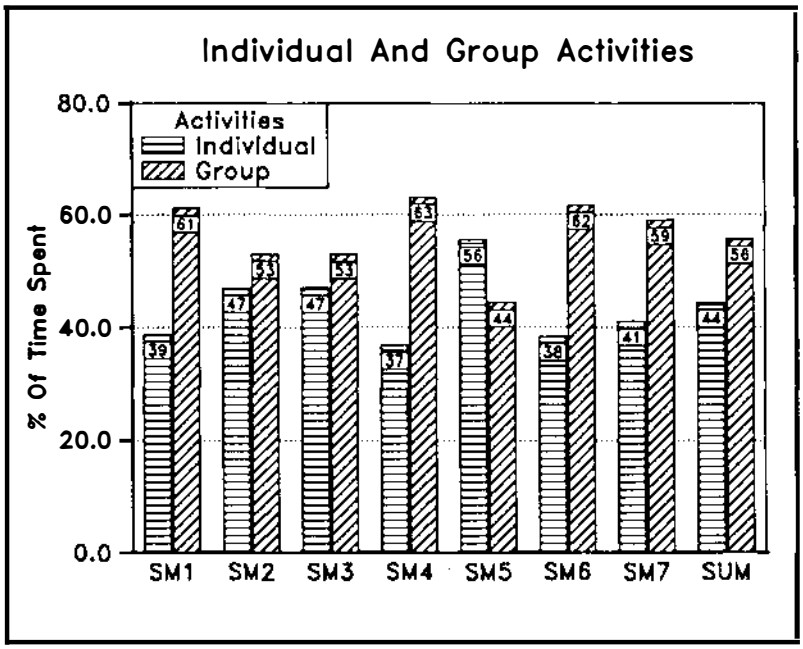


Figure 4: **Per module comparison of individual and group activities.**

on his or her own part of the project (determined as internal design, coding and Phase I testing without overhead) is less than that spent in dealing with other members of the development group.

The ability to communicate and the need for constant communications are necessary for a productive software designer. Since much of this time is spent in talking with people outside of the software design group, as in the external specifications and Phase II testing activities, software engineers must also be effective communicators for a project to be successful. As the high

productivity associated with the SPP indicates, interactivity was quite high for everyone involved.

Test Phase Information For The SPP

As is discussed in the appendix, the testing phase consists of three stages – Phase I, Phase II and user testing. The criteria used for the three stages of testing the SPP are outlined here, followed by a discussion of how the testing effort was managed and the impact of testing on scheduling.

Test Criteria

The following criteria were set for Phase I testing:

- Test the software on one job.
- Test all functions described in the external specifications.
- Test all parameters.
- Check the validity of the graphical results.
- Update the internal and external documentation.

The following criteria were set for Phase II testing:

- Test the software on several jobs (at least 3) which exercise different functionalities of the software.
- Test the communication links between the various software modules.
- Test all the functions described in the specifications.
- Test multiple combinations of parameter settings.
- Test boundary values of parameter settings.
- Verify the validity of all error and help messages.
- Test all graphics outputs.
- Report updates needed to be made to the external documentation.

The successful execution of one commercial job to be run in-house by an eventual user was also included as part of the criteria.

The criteria for user testing mirrored the criteria for Phase II testing. However, the tests were performed independently by users at selected FLICs. This testing was completed in June, 1987. Approximately ten defects were reported and corrected during user testing. These results indicate that Phase I and Phase II testing were performed properly.

Resource Management

Central to the success of the in-house testing (Phases I and II) was the strategy adopted to manage the storage and distribution of the software and the test jobs. This strategy is illustrated in Figure 5. The left side of the figure (left of the bar) shows the software engineer's code management. The right side of the figure shows the tester's 'released' code and test job management.

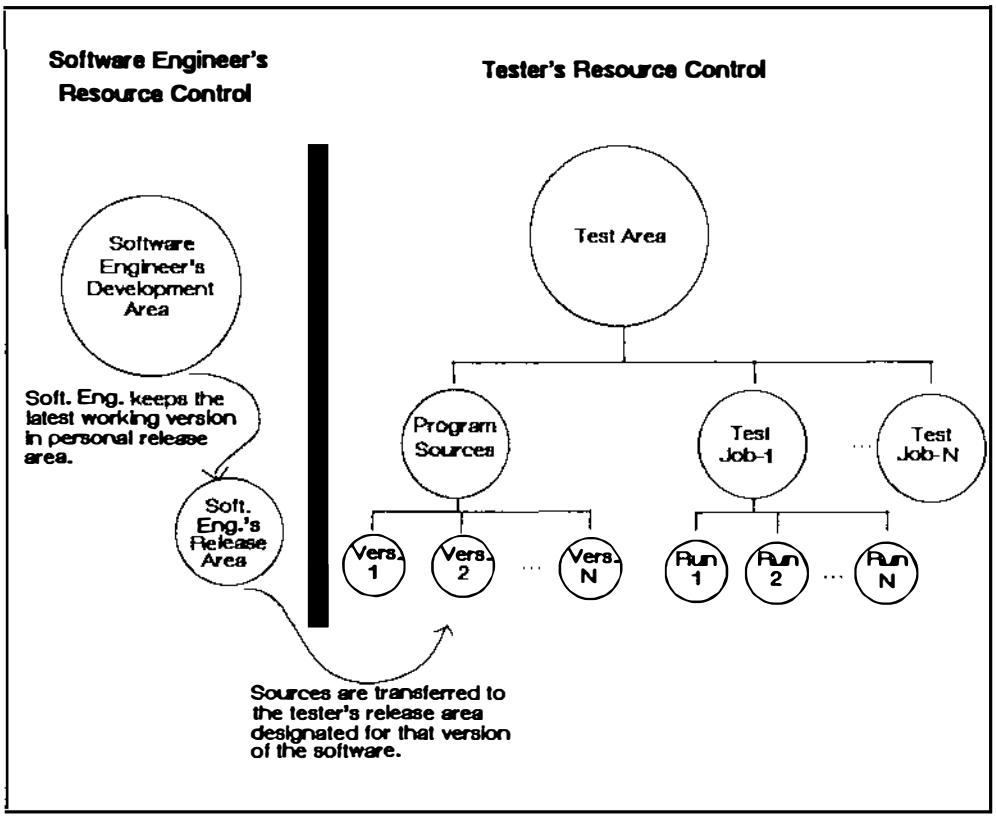


Figure 5: **Code and test job resource management.**

Each engineer developed the code in one personal area and kept the latest working version in a separate personal area. The project leaders (one for software engineering and one for the project design) had the responsibility of periodically gathering the latest working versions of each module and transferring them to a reserved sub-area of the main test area. Each such gathering was treated as a release with its own version number. Command procedures were developed to ease the building of the releases and to minimize human error in the building process.

After a new release was completed, the testers would apply these latest released versions to the desired test jobs. Each execution of a test job using a new version of the applications software was done in a new sub-area of the main test area.

The organization exercised in the management of the software proved essential in monitoring the testing and subsequent code revision processes. Without such management of the configurations, utter chaos would likely have been the result!

The Impact Of The Test Phase On Meeting Schedules

The preliminary schedules for the various modules were made in November, 1985 with the first revisions made in late January, 1986. The SPP was released to the FLICs in the first week of June, although the planned release date had originally been the end of March. The January estimates indicated that all activities through coding would not be completed until the end of March, unlike the preliminary schedule for completion of these activities by the end of February.

Much of the increase in the elapsed time came in the testing activities for a variety of reasons (increased number of tests desired at both the module and package levels, underestimating the time needed to set up and run the test jobs, other activities that conflicted with the Phase II testers' availability, etc.), however the users were happy to have problem detection migrate upstream in exchange for the SPP being slightly behind schedule. Some more time was needed due to the addition of two more modules (SM3 and SM4) to the SPP soon after the January schedules were defined. Finally, more effort was involved in one of the two largest modules (SM2) than had been anticipated.

Comparing SPP Statistics To Industry Standards

A book written by Jones [4] is the primary reference used here for comparing SPP project data to that of the software development industry. For these comparisons, the metrics used will be what he refers to as project productivity metrics. To be consistent, lines of code for this project have been defined to be quite similar to his line-counting rules. Although his examples and case studies are primarily COBOL based, Jones considers the generation of FORTRAN software to be equivalent in scope to COBOL. By comparing (red) apples to (yellow) apples here, the paradoxes Jones described that exist with many productivity indicators used by the industry should be avoided.

As Jones indicates in his book, tangible and intangible factors can dramatically effect software development. To be fair, the project data is compared only to his case studies and to examples that assume the software engineers are highly motivated, experienced professionals who are familiar with the product's functionality. Jones assumes the average software engineer works about 50 hours per week, although this time can increase considerably when a project gets 'hot'. In addition, he assumes the latest tools are available for developing third generation language software.

By confining comparisons of the SPP to Jones' best case examples of software product development, a very interesting conclusion can be reached for the hallowed and often misused metric: the lines of code produced per day (LOC/day) per software engineer. His best case examples for a software package of medium size would indicate an average productivity of around 9 LOC/day in the industry for a software engineer. This value may be as low as 6 and as high as 12 LOC/day.

If the four lowest productivity software modules in the SPP are used for determining LOC/day, a value very close to 20 LOC/day is determined. For a simple average of all the modules, a value of 37 LOC/day[1] is determined. The productivity of the software engineers on the SPP project is thus up to four times greater than the norm of the best case examples provided by Jones in similar situations.

[1]This is easily determined from Table 1 to be 28,945 lines of code divided by 155.5 weeks, with 5 days per week. The result is 28,945/777.5 or 37.2 LOC/day.

While this comparison is extremely gratifying, there are more data available in Table 1 that relate to software productivity metrics. These data are discussed in the next sections with numerous references made to the various figures derived from these data.

Lines of Code Metrics And Their Implications

Figure 6 indicates the number of lines of code generated per module. As shown, it is only

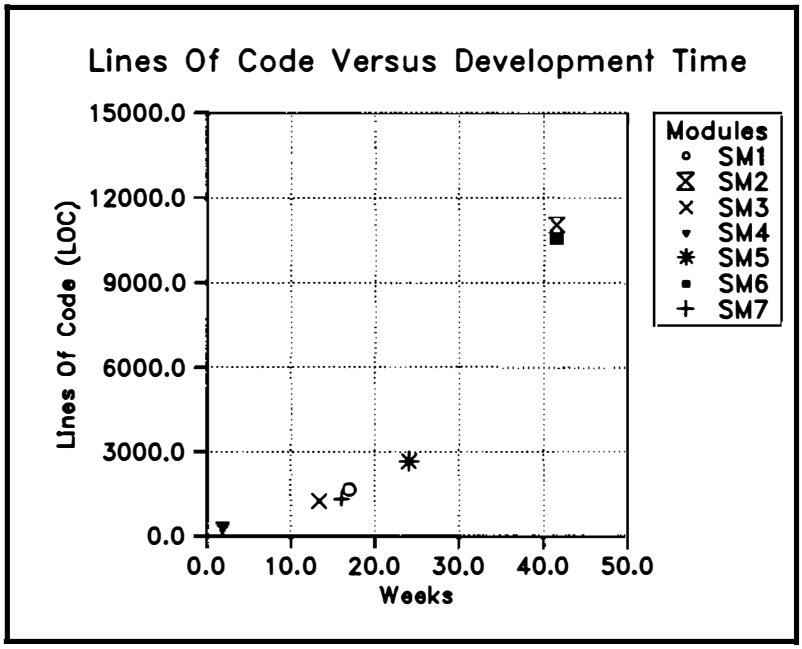


Figure 6: **The time used per module to produce the lines of code.**

marginally useful as a metric. With no other information available, it would appear that productivity increases as the size and complexity of the module increases (contrary to intuition and experience). This topic will be addressed in more detail when using combined metrics.

The figure does, however, give some hints as to the productivity of the software engineers. If just the modules that took between 10 and 30 weeks to develop are examined, it appears that 2,000 lines of code can be generated in 20 weeks, a rate of 20 LOC/day. A regression line drawn through these points gives the same results. These results are more easily determined by looking at the cluster of points around the value of 20 LOC/day on Figure 7.

Figure 7 indicates rather clearly that, when compared to the development of the other modules, something very different is going on with the development of the smallest module (SM4) and the largest two modules (SM2 and SM6). At first glance, it would seem that whatever is affecting these three modules is desirable since the LOC/day for them is much higher. If an average of the LOC/day values of all modules on Figure 7 is taken, a figure of over 37 LOC/day is determined. The dotted line in the figure indicates what a reasonable rate of LOC/day productivity would be while the dashed line indicates what a rather high rate would be.

A regression line drawn with the use of all the points in Figure 6 gives a rate of nearly 35 LOC/day

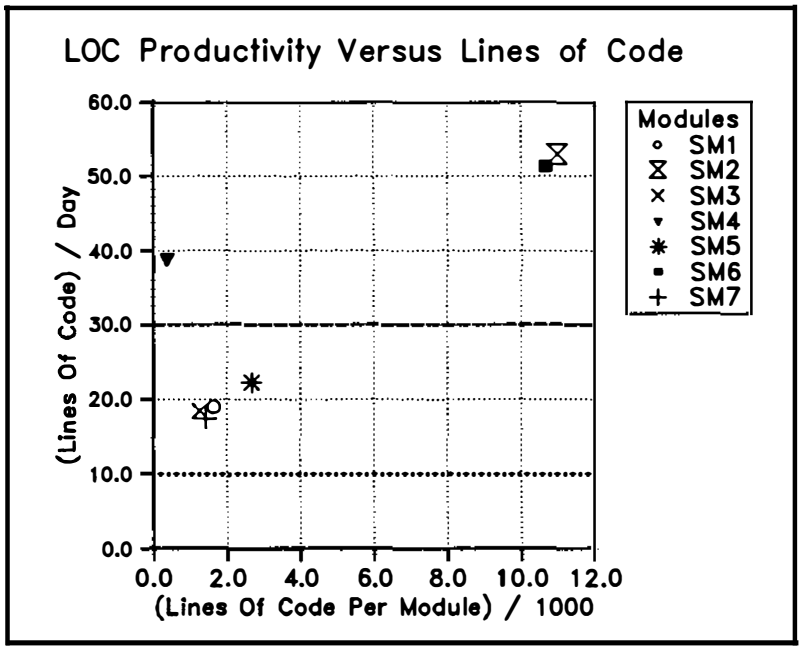


Figure 7: **Productivity in terms of lines of code.**

for a module with 2,000 lines of code. Why these results should be taken with a grain of salt is suggested in the next section.

Total Line Metrics And Their Implications

If the production of documentation along with generating code is included as a metric, then the results can be displayed as is shown in Figure 8. Note that the points for SM2 and SM6 are at quite different locations, although the LOC points plotted for them in Figure 6 are very close. Total productivity for SM6 is much higher than that of SM2 if this metric is used. A regression line drawn with the use of all modules except SM6 provides a productivity rate of 72 total lines per day for a 20–week development. SM6 itself was produced at over twice this rate (188 total lines per day).

Figure 9 indicates SM4 and SM6 were produced much more effectively than the other modules. Due to the nature of these two modules, both of the software engineers were able to make extensive use of templates and copying of the units comprising their modules. The dotted line in the figure indicates a relatively low rate of total line productivity, at least for this project, while the dashed line indicates a relatively high rate of productivity. Clearly, the ability to duplicate code and documentation formats within a module have positive effects on productivity.

Combinations Of Metrics And Their Implications

So far the production by the software engineers has been reviewed at a line by line level. However, like any other experts, software engineers think of their production in chunks of the simplest

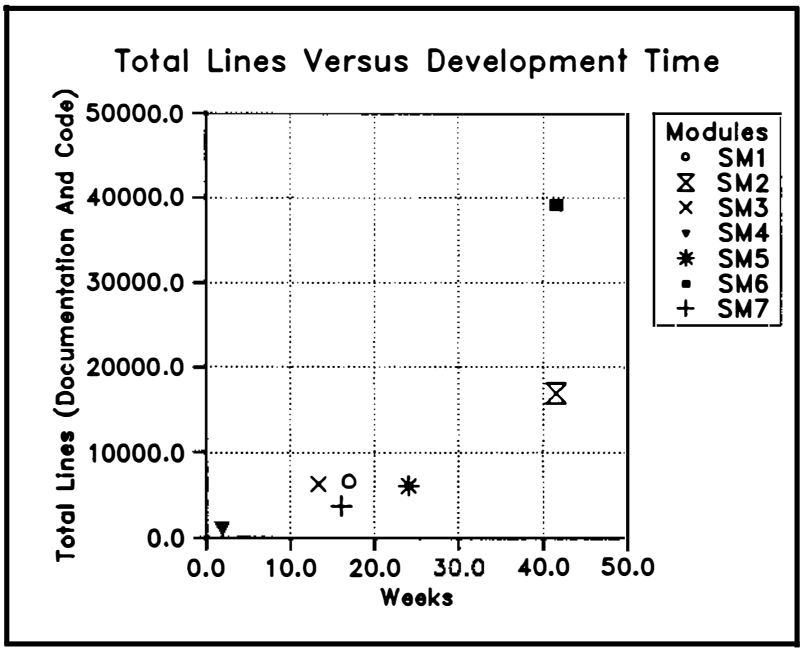


Figure 8: **The time used per module to produce code and documentation.**

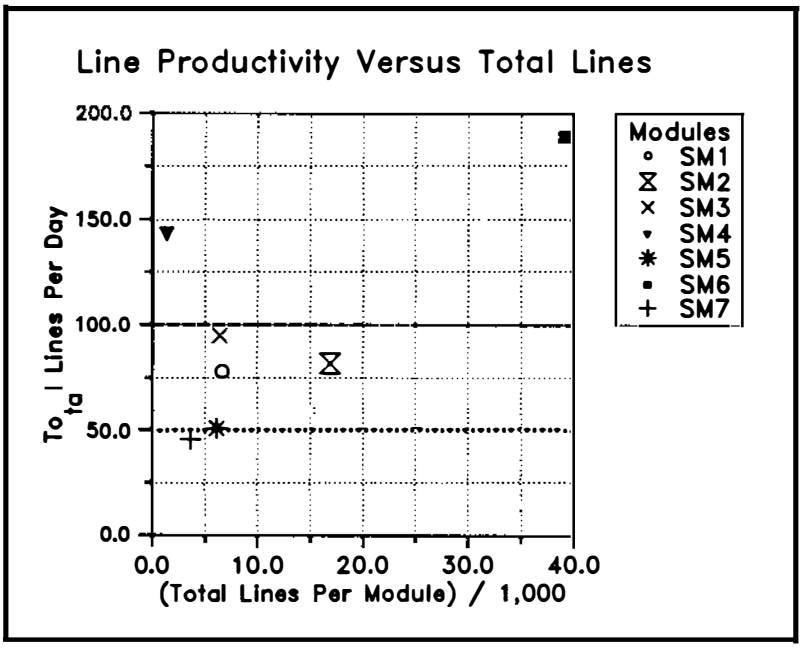


Figure 9: **Productivity in terms of total lines.**

components. For the SPP, the most reasonable chunk to address is a software unit – either a main routine or subroutine. Figure 10 shows the rate at which software units were developed for

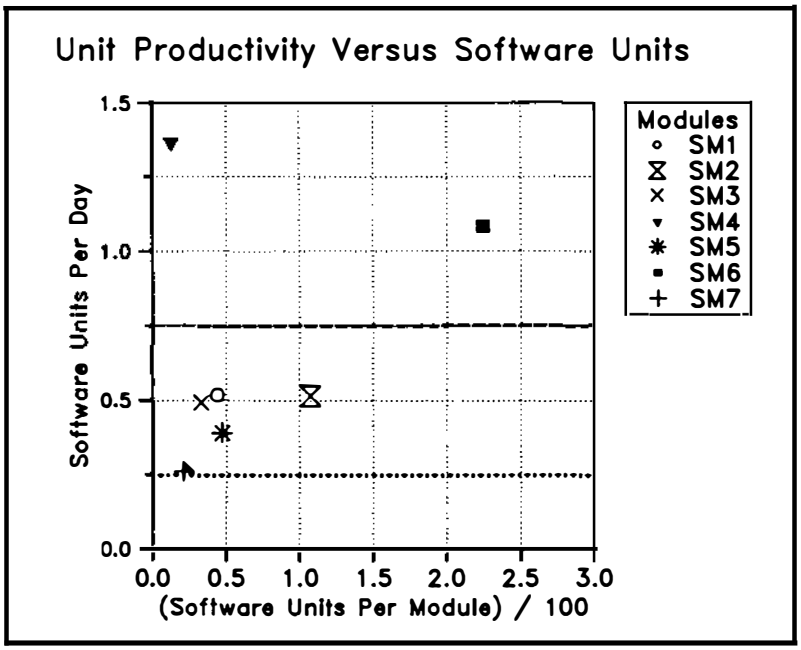


Figure 10: **Productivity in terms of software units.**

the modules in the package.

As has been noted before, the production rate of SM4 and SM6 is much greater than that of the other modules. Figure 10 indicates this is true at the unit level also. The lowest rate of productivity was for SM7. It may be noted that the software engineer responsible for SM7 had to split time between developing this module and supporting another on a completely different hardware system. Also, in violation of our software development model, several of the requirements for this module were not defined until the middle of the coding activity. Thus the engineer's ability to manage the development of the module and to understand its use before writing code were both reduced in comparison to the other modules.

In Figure 11, we combine the LOC per software unit metric with the software units per day metric. This display isolates modules with high LOC/day rates but lower units per day rates on the left side of the dashed line. A module that is produced in this fashion may have maintenance problems in the long term. Large units produced at lower rates are usually less carefully structured and reviewed by the author than smaller units. This conclusion is based on the authors' experience and on observations made by Jones.

Modules that appear on the right side of the dashed line in Figure 11 contain small, usually well designed units produced at a rapid rate. It should not be surprising that two of the modules falling into this region for the SPP are SM4 and SM6. Characteristics of the remaining five modules could also be determined from this figure. However, Figure 12 provides a better method.

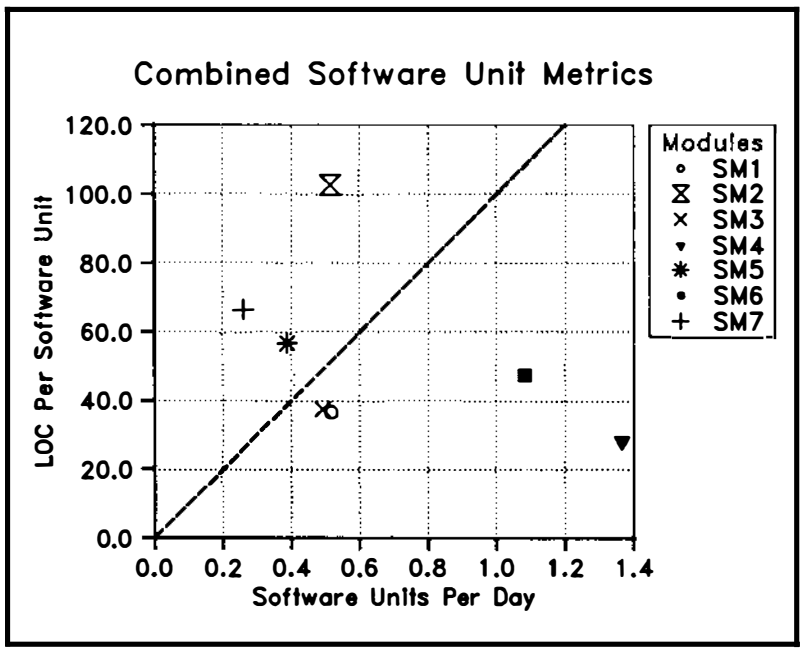


Figure 11: **Software unit metrics for flagging maintenance problems.**

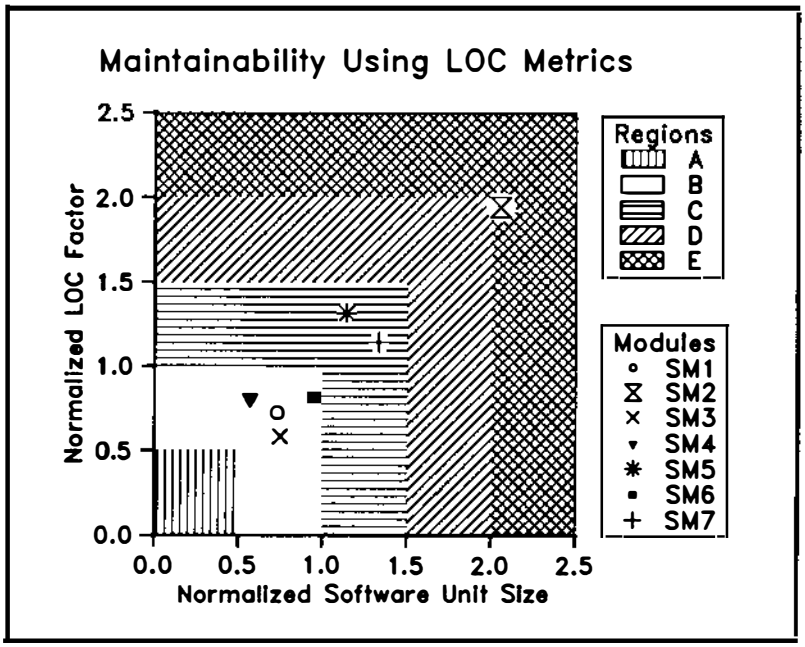


Figure 12: **Normalized LOC metrics for estimating maintainability.**

Figure 12 uses a combination of the metrics we have looked at so far, but is normalized to provide similar values on the two axes. For the horizontal axis, the LOC per software unit have been divided by 50 to give a normalized software unit size. On the vertical axis, the LOC per total lines produced for each module has been multiplied by three (As a reminder, total lines are the

sum of the LOC and the lines of documentation for a module). These normalization factors are based on the experience of the authors and on the software development standards used at the Schlumberger engineering centers. With these normalization factors, well-designed software modules will usually have values of around one on both axes.

This figure provides quite an interesting set of possibilities. In or beyond region E, modules that are difficult to maintain and use would be expected to appear. Modules in the left side of region E would be less documented, causing both maintenance and user problems.[2] Modules in the right side of region E would have software units that are too complex and unstructured to permit reasonable maintenance.

Region D is a diluted version of Region E. While modules in Region E are very likely to be unstable, the chances are less that those in Region D will have some major long term problems. Modules in Region C will probably not have major problems, but the possibility of minor nagging ones exists. Modules in Region B should be very stable, with the possibility of deficiences relative to the agreed upon requirements being rather remote. Region A identifies over-engineered modules with insufficient functionality per unit (as Jones indicates) or so much documentation that the reader gets lost in the noise.

The user testing done on the modules for the SPP plus the observations of the Phase II testers bear out the above assumptions. SM2 had the most problems in test, although the amount is small compared to industry standards.[3] SM2 was found to fall in Jones' category of 3 defects per 1,000 LOC per year (indicating the base code is stabilizing). SM5 and SM7 had a couple minor problems, with both of them close to the stable category of 0.5 defects per 1,000 LOC per year.

The remaining modules, all in Region B, fall well within the stable category. Most of the problems reported for these modules were caused by the underlying data base system subroutines rather than by the modules themselves. The few remaining problems were primarily due to overlooked variants of the initial requirements. No defects due to gross logic errors were reported for any of the modules.

Figure 12 provides a handy way in a third generation software language development environment to detect whether the development process for new software was properly followed. With appropriate renormalizations to account for more compact, higher level languages or differing environments, Figure 12 should be useful for a variety of software development projects. Not only can it be used to flag new modules as possible troublemakers, but it can be used to identify existing modules that may continue to have maintenance problems.

Conclusion

With good communications between management, product designers, software designers and the user community, software quality and productivity can easily exceed current industry standards. The use of a well-defined methodology such as the software development model applied during this

[2] The testing done on these modules would probably also be inadequate. Insufficient documentation generally occurs when the module is rushed, because sufficient time time cannot be scheduled to properly test the module.

[3] It should be noted that considerable time was spent by the software engineer for SM2 on correcting problems with a vendor library to which the module was strongly coupled. In addition, it wasn't possible to template the units in this module as was done for SM6. Given this information, the tremendous amount of code that was generated with as few problems as have been encountered in testing was remarkable.

project is necessary to properly manage the project and to ensure all agreed upon requirements are met. In addition, having a critical mass of good people who know what is needed up front and who are provided with the tools needed to perform their tasks is key to producing quality software.

In summary, the high quality of the SPP is due to the expertise of the software developers involved, the sound development model used, the manner in which requirements were set and followed, and the considerable time spent testing the SPP before releasing it to the user community. The long term maintenance effort for the SPP is expected to be minimal as regards defect repairs. Both the metrics used to review the project and a year of user testing indicate this conclusion is well warranted.

Acknowledgements

The authors' wish to express their appreciation for the efforts made by the SPP development team. Without their hard work and willingness to follow the development model discussed in this paper, it is unlikely that this project would have been a success. The team was made up from the following caste: Olive Liu and Steve Palmer as the product designers; Doran Sprecher, Debby Willis, Homer Brown, Alan Barrilleaux and Ali Toufaily as the software engineers; Sam Ninan and Dewey Guthrie as the software technicians.

Several other people, although not on the team, played an important part in making the project a success. These include Frank Morris, Dave Scheibner, Peter Wu, Ed Pierce, Charles Flaum, Dave Smith and Darrel Cannon. A general thank you is given to various people in the engineering centers and at the FLICs who contributed to the project.

References

[1] VAX is a registered trademark of the Digital Equipment Company (DEC).

[2] MAT (Maintainability Analysis Tool) is a copyrighted product distributed by the Science Applications International Corporation (SAIC).

[3] TELLAPLAN is a registered trademark of the Integrated Software Systems Company (ISSCO).

[4] Jones, Capers: *Programming Productivity*, McGraw-Hill Book Company, 1986.

Appendix A. A Commercial Software Development Model

Introduction

A software development model is needed to insure that:

- the software being developed is necessary,
- the software being developed is on target,
- the development engineer knows where he/she stands during the development period, and
- the software is developed in a timely fashion relative to other related software efforts.

The model described here is primarily intended for usage by those software engineers involved in producing applications software. It is similar in scope and in content to a number of models that have been discussed in the literature, but has been tempered by the experience of the authors.

The Model

This model assumes that there are four distinct phases in the life of the software development cycle:

1. product definition,
2. software specifications,
3. implementation, and
4. user test.

Of these four phases, this model primarily addresses the two phases related to the development of the software itself – phases two and three. Less detailed descriptions will be given of the other two phases.

The key players in this model are the product designers, the software engineers, the testers, and the user test coordinator. Section heads and group leaders are responsible for seeing that the model is followed on schedule, participating in reviews, overseeing schedule changes, and gathering statistics for project analysis.

Product Definition

The purpose of the product definition phase is to conceive and define the initial product specifications. This phase is the responsibility of the product designer. The product designer may be the software engineer (as is usually the case with data handling utilities), or may be a separate person (as is usually the case with data analysis packages). If the product designer (or designers)

is distinct from the software engineer (or engineers), it is helpful if the software engineer(s) can allocate roughly five percent of available time to interacting with the product designer(s) during this phase.

The steps of this phase are not described here. The important thing to note about this phase is that there should be a definite break between the end of this phase and the beginning of the commercial software specifications phase.

There must be a formal 'sign-off' by the product designer when the product specifications are delivered to the software engineer. This sign-off should include the delivery of product specifications, a presentation by the product designer and acceptance by the software group.

If the product designer is unable to deliver reasonable product specifications, then

(a) the product is not ready for commercialization, and specifications must be revised until they are properly defined, or

(b) the product and software development groups must agree on continuing jointly to refine the product definition until it is ready for the next phase.

If (b) from above is the resultant case, then some prototyping by the product developer and software engineer may need to be done. Sufficient time should be allotted for the prototyping.

There is a simple rule of thumb that can be used to indicate the completion of the product definition phase. If the external specifications can be generated primarily from the product specifications and the resulting document can be sent to the users as a comprehensive preview of the upcoming product, then the product definition phase is finished.

Software Specifications

The purpose of the software specifications phase is to develop the specifications needed to produce the initial release of the software, and to complete the scheduling. This phase is the first phase directly related to commercialization and is performed by the software engineer. The major outputs of this phase are:

1. external specifications (ES),
2. internal design (ID), and
3. implementation schedule.

This phase can be broken down into the following steps:

1. determine the critical paths,
2. make a preliminary schedule for this phase,
3. gather facts for the ES,

4. identify user contacts (who is to review upcoming documents),
5. write the ES,
6. formally review the ES (peer review),
7. base a letter on product specifications and deliver the ES to assigned users for review,
8. <optional> perform prototyping prior to the ID,
9. develop an initial ID,
10. review user comments regarding the ES, preferably with users present,
11. revise the ES,
12. revise the ID,
13. develop an implementation phase schedule, and
14. formally review the ID and the implementation schedule.

The ES document should in essence have the contents of the eventual documentation provided to the user when the software is released for user testing (*e.g.*, purpose, theory, inputs, outputs, and examples). The generation of a nicely illustrated external specifications document is encouraged.

The letter which is sent to the users involved in the review process should be a description of the product with illustrations, based on the ES document(s). The ES document should be made available upon request.

There are no set methodologies for internal design which are proposed by this model. Whatever design is used must, however, include the program logic flow, a break down of the software units that comprise each module (at least the top-level units), data structures, and performance estimates (if possible). The design must be composed in a form which will be easy to communicate to reviewers. A well constructed internal design document will serve as the basis for the internal documentation (title pages and inline comments) written for the module.

Implementation

The purpose of the implementation phase is to develop the software and make an experimental (perhaps more correctly referred to as pre-commercial) release of the software. The software engineer, along with an in-house test engineer, is responsible for this phase. This phase consists of developing the code, testing it and releasing it. The steps in this phase are as follows:

1. developing the code,
2. debugging the code,
3. Phase I test,
4. integration,

5. Phase II test,

6. code and documentation submitted for audit,

7. letter written describing the Phase II test results, and

8. release a pre-commercial version of the software.

Steps 1, 2 and 3 will be iterative and are the responsibility of the software engineer. The software engineer is also responsible for submitting the code and documentation for audit as well as for the release of the code. The standards for release of the code as well as the control over the release procedure should be the responsibility of an independent group.

Integration involves placing the module in the software chain of which it is a part and working out problems which may result from improper communications between modules.

Phase II testing and reporting is the responsibility of a designated test engineer. It is very important that this be someone other than the software engineer (software engineers have a tendency to keep enhancing the code rather than fully testing the code that exists). There must be a clear distinction made during Phase II testing as to what constitutes a request for defect corrections, and what constitutes a request for an enhancement. Time should be scheduled for defect corrections. Some time should also be scheduled as well for enhancements, though these should be minimal if this model has been followed properly. Allow plenty of time for Phase II testing (15% to 20% of the total development time).

Additional capabilities can be added during the Phase II testing step only if all parties formally agree to do so and the schedule clearly indicates the reason for the delay. If these requested capabilities are significant, then either the development process should be restarted from the beginning or their implementation should be delayed until another major upgrade of the software is scheduled.

User Test

The purpose of the user test phase is to initially introduce the software to selected locations for rigorous testing by members of the user community. Persons responsible for this phase are the software engineer, the Phase II tester, and the user test coordinator. This phase should primarily involve the latter two. However, the software engineer should be available as needed.

The steps in user testing are:

1. develop the user test package,

2. <optional> host training schools to kick off the user test,

3. distribute the software to the users' locations,

4. send out reminders periodically requesting test results,

5. write a user test report (with input from user testers),

6. make revisions to the code as a result of the user testing, and

7. make the first commercial release of the software.

After this last step, the development team can proceed to their next assignment.

The user test package should include a sample job and details about what needs to be tested. This should be supplied by the Phase II tester with assistance from the software engineer. The user test coordinator needs to determine who will perform the tests and the time needed for testing.

Summary

This model captures the essence of the development of commercial software. Details, such as how to go about signing off when each step in a phase is completed and how to best handle changes to specifications or schedules, have been overlooked. As this model is applied within the software engineer's environment, these details should become clear.

Session 7

SOFTWARE QUALITY MANAGEMENT

"Documentation of Software Test Experience with an Expert System"
John P. Walter, California State University

"A Case Study of a Basic Quality Measurement"
Millard J. McQuaid and Walter M. Wycherley, AT&T Information Systems

"How to Decide When to Stop Testing"
Elaine Weyuker, New York University

DOCUMENTATION OF SOFTWARE TEST EXPERIENCE WITH AN EXPERT SYSTEM

John P. Walter, D.Sc.
Professor, Computer Information Systems
California State University, Dominguez Hills

ABSTRACT

The paper describes an expert system for management of software test documentation. The role of this system is to facilitate the documentation of test results, assure the reusability of test transactions, automate the interface between and among development and testing personnel, and link the testing activities of development and test personnel working at different sites. The system builds a reusable test library from user-supplied test transactions created during the initial user requirements analysis, test activities conducted by software development personnel, and work of the testing organization. Bug reporting and filing of test plans are interactive processes. During execution of these processes, the data base receives test transactions and information that describes the test environment.

The data base is analyzed periodically and statistical reports are generated. Reports include analysis of variance, control charts, correlation/regression, and trend tests. A pattern recognition technique is applied for detection of common features of reported software problems (to reduce Type 1 testing errors), and outliers among reported software problems (to reduce Type 2 errors).

BIOGRAPHICAL SKETCH

Dr. Walter's primary teaching and writing areas are systems analysis and design. Most of his consulting is in software quality, systems development, data communications, statistics, and simulation. John was a co-founder of a multinational computer services firm based in Europe, where he developed a system for quality assurance of more than 3000 software files for U.S. and European multinational enterprises. John holds a DSc in Applied Computer Science from the University of Paris Faculty of Sciences, an MS in Systems Engineering from West Coast University, and a BA in math. He holds the CISA, CDP, and CSP certifications, is a registered professional engineer, and has been elected a Fellow of the Institute for the Advancement of Engineering. He has served as an officer, organizer, and speaker for ACM, DPMA and IEEE, as well as a guest speaker for ASM, the EDP Auditors Association and the Structured Techniques Association. For further information about the material of this present paper, John may be reached at 213 / 516-3348.

I INTRODUCTION

The role of this system is to:

* Assist in management of software test documentation.
  When software reliability is crucial the accumulation of large amounts of test plans, test results, and so forth may become so difficult to maintain in an organized fashion that valuable time is taken from essential development and testing activities. Managing the test documentation should be computer work and not people work.

* Facilitate the documentation of test results.
  When documenting one's work becomes too cumbersome, a common practice of systems people is to delegate the honor to persons less capable of refusing it. This may take longer, may not be done very well, and maybe not at all.

* Assure the reusability of test transactions.
  If they are available, previously-used test transactions are likely to be used again. If they are not available, it may be difficult to reconstruct them exactly.

* Automate the interface between and among development and testing personnel.
  Physical separation and organizational constraints can make it almost impossible for developers to talk with test personnel. If they share a common data base, the need to talk and meet can be reduced.

* Link the testing activities of development and test personnel working at different sites.
  With data communications capabilities, personnel at several sites can be working with the same records.

II BUILDING A REUSABLE TEST LIBRARY

A library of test environment data is built on the basis of the test transactions originally supplied by users during the initial user requirements analysis. This startup version of the test library provides at least a beginning collection of test transaction models for subsequent phases of the project.

Test activities conducted by software development personnel are the production of test plans, test transactions, recordings of expected results, and so forth. These are added to the test library and kept in one place for review by development personnel and the testing organization. Work of the testing organization is later added.

This scenario is depicted in Figure 1.

By providing the library of test environment data, the shell helps enforce reporting and follow-up of software failures.

III BUG REPORTING

Bug reports are entered by means of an interactive program, such as FormTool*, that enables production of a corresponding hard copy form. This way, persons submitting failure data can complete paper forms manually. They are not necessarily constrained to online data entry. Data can then be entered later by somebody who has access to a communicating PC.

A sample of a filled-in bug report is provided in Figure 2. Figure 2 was obtained as follows: (1) A blank bug report form was designed interactively and an image of it was stored on floppy diskette . (2) A camera-ready hardcopy image was produced on a dot-matrix printer. (3) A printed copy of the form was used to manually record the test environment information. (4) Using the machine-readable image of the blank form as a template, information from the hardcopy bug report was entered interactively into a file on floppy diskette.

A separate file is used to receive only the data entries from bug reports such as Figure 2, excluding the template information. In bug collection (see Figure 1), the shell command BUGR provides for storage of such data entries on a mainframe disk for subsequent retrieval and analysis.

---

* FormTool (see References section of this paper) is an inexpensive program for IBM-compatible personal computers. It assists in the design of hardcopy paper forms. Once the form is designed, the same program can be used to display an image of the form on screen, for purposes of data entry.

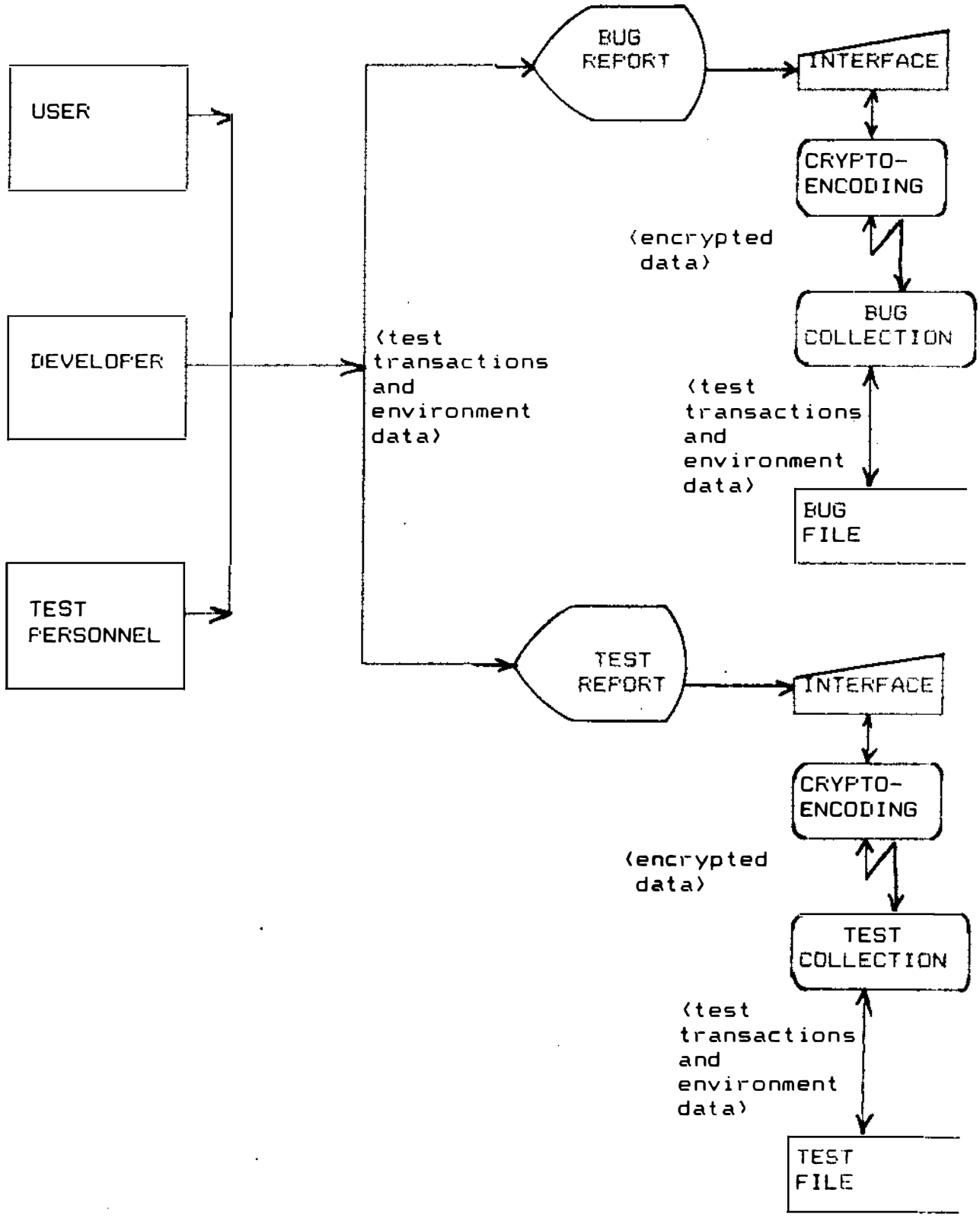

USER
DEVELOPER
TEST PERSONNEL
BUG REPORT
INTERFACE
CRYPTO-ENCODING
(encrypted data)
BUG COLLECTION
(test transactions and environment data)
(test transactions and environment data)
BUG FILE
TEST REPORT
INTERFACE
CRYPTO-ENCODING
(encrypted data)
TEST COLLECTION
(test transactions and environment data)
TEST FILE

Figure 1

BUG REPORT

USER

User Name:Mgr, Project 3348	Phone: 3348	Department: Micro Systems	Location: Home Office	Date: 2/13/87
System:LABOR PRODUCIVITY		Program/Procedure:Labor,Billing,Productivity Name:Calc & Report	Number: 711.11	
Vendor: M.I.S.	Service No.: 2324	Service Location: Home Office		Date: 2/7/87
Document Title Lab Prod User	Document No.: 707.2	Release Date: 3/15/86		

Support: [ ]Source Documents [X]Execution Report
[ ]Input Media [ ]User Report [ ]Other:_____

Impact: [X]Cost:_____? [ ]Delay:_____
[X]Project [X]Department [ ]Gov't Agency:_____

Problem Description:Reported productivity too far above average.

Action Expected:Correction & re-run.

INVESTIGATOR

Date Received: 2/13/87	Assigned to: J.S.	Date Assigned: 2/13/87	Received by: J.S.	Date Resolved: 2/20/87

Immediate Action Taken:Correction of file access module.

Description of Corrective Action: Update source, object, and load modules. Re-run.

ACCOUNTING

Refund: [X]Full [ ]Partial [ ]Refused [ ]Not Applicable

Why:Calculation error due to M.I.S. programming mistake.

Bill to Account: 605.86	Quantities Billed: 8 runs.	Costs Billed: $405.00	
Accounting Action by: D. Jones	Location: Home Office	Phone: 1055	Date: 2/20/87

Return to: Software Quality Assurance

Figure 2.

IV FILING TEST ENVIRONMENT INFORMATION

Using the shell command TESTR, software test procedures are handled in a manner similar to the way BUGR handles bug reports. A sample of a completed hardcopy form for a software test procedure is provided in Figure 3.

Data concerning the environment of the test is captured from bug reports and test procedures for recording on a common test environment record. The test environment record contains:

* Test document type (This may be a test procedure, bug report, or another.)
* Date of completion of the test document
* Program Name
* Program Number
* Data Set Type (This may be a small sample of transactions (reduced) or full-sized.)
* Source of test data
* Reports Produced
* Functions Exercised
* Specifications Documents
* Observations
* Findings
* Final Determination
* User Name
* User Department
* User Location
* System
* Vendor
* Service Location
* Document Title
* Document No.
* Support (This includes references to source program and data listings, diskettes, chips, and so forth.)
* Impact
* Problem Description
* Corrective action
* Costs billed

S O F T W A R E T E S T P R O C E D U R E

Program Name: LABOR EARLY WARNING SYSTEM	Program Number: 711.5
Reduced Data Set Type: Project labor hour estimate transactions, entered by conversational terminal.	Test Type: Acceptance [ ] Maintenance [X]
Source: Managers of Projects 3348, 3556 & 4961.	No. of Input Records: 3

Reports Produced: Project Labor Forecast Summary.

Functions Exercised: Update, Consolidation & Summary Reporting.

Specifications Documents: 711.5 User Requirements Analysis,
711.5 System Specification.

Preliminary Observations: Some head counts are artificially low if
1 month = 168 labor hours.

Preliminary Findings: [ ]Accept [X]Reject [ ]Hold pending:_____	By: J. Swanson	Date: 2/14/87

Full-sized Data Set Type:

Source:	No. of Input Records:

Reports Produced:

Functions Exercised:

Specifications Documents:

Maintenance Instructions:

Final Determinations: [ ]Accept [ ]Reject [ ]Hold pending:_____	By:	Date:
Routed to:	Location:	Date:

Page 3

V ANALYSIS OF THE DATA BASE

Figure 4 gives an overview of the process by which test environment information is organized, analyzed and acted upon. In Figure 5 the analysis part is shown in greater detail.

Consolidation is done through the PC, with test environment records categorized at the mainframe according to the features selected. This is achieved by having the shell command CONSOL drive the mainframe. During the execution of CONSOL the mainframe computes the Mean Time To Failure (MTTF)*, as well as the variance of failure times, associated with each category of test environment records selected. The program detects and displays the attributes of those categories demonstrating persistent failure, in addition to categories having low failure rates.

Analysis is also done through the PC. Driven by the PC, the mainframe performs traditional quality control reporting techniques, as applied to the selected categories of test records. Shell commands for analysis and reporting include:

* ANOVA (Analysis of Variance)
* CHART (Control charts)
* CORREG (Correlation / Regression)
* TREND (Trend Test)

Using the analysis commands of the shell, it is easy to focus on the extreme values (outliers) for failure rates in planning future tests. By quickly identifying programs with high failure rates, software quality personnel can minimize risk due to accepting problem-prone software. Risks due to unwarranted rejection of good programs are also reduced. Patterns of frequent bug reports based on few actual failures are also identified. This way such patterns are easier to substantiate than pencil and paper analysis would allow.

---

* Time-to-failure statistics (MTTF and failure time variance) are used in preference to time-between-failure measures because it is easier from a management point of view to compare MTTF with shelf-life statistics.

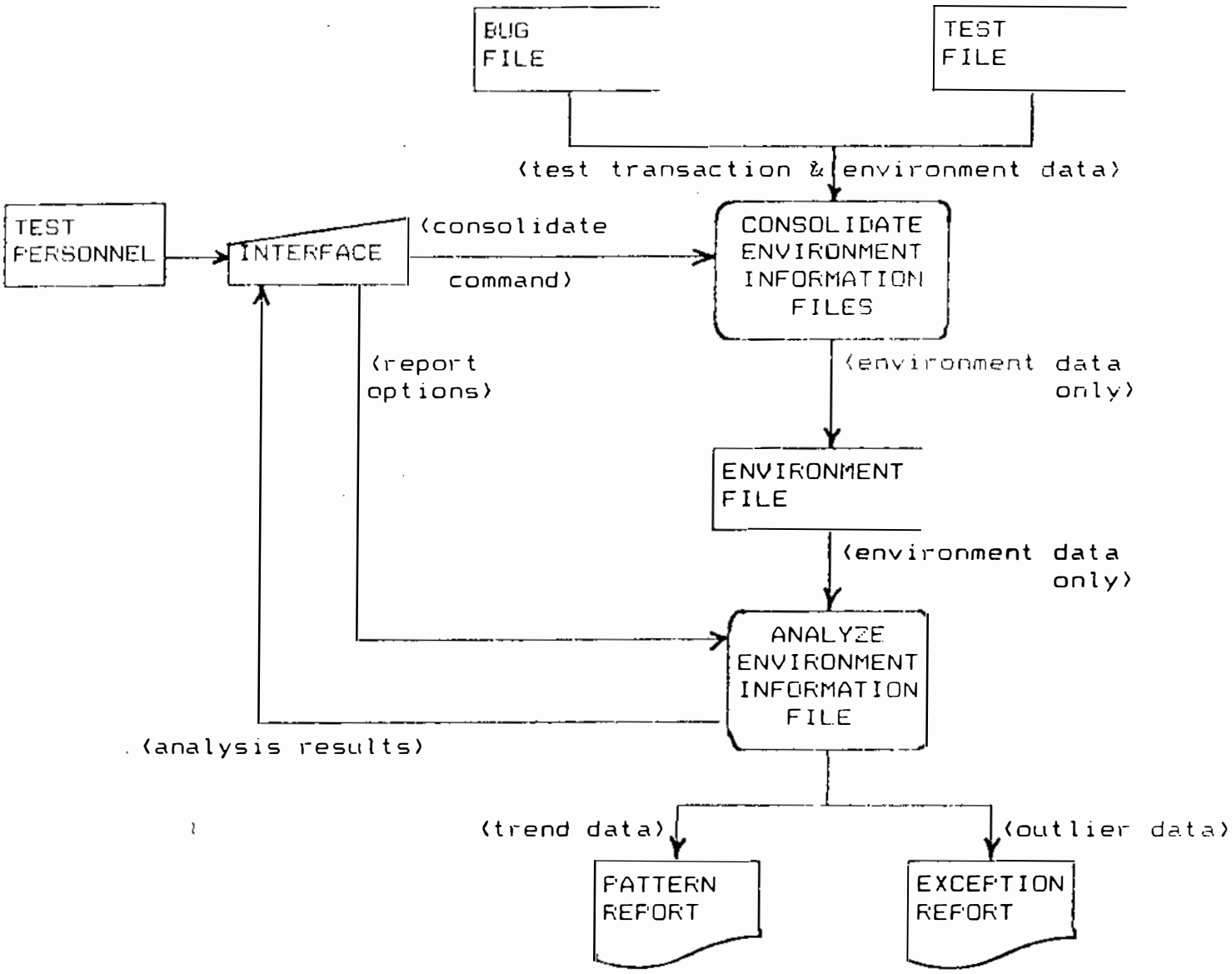
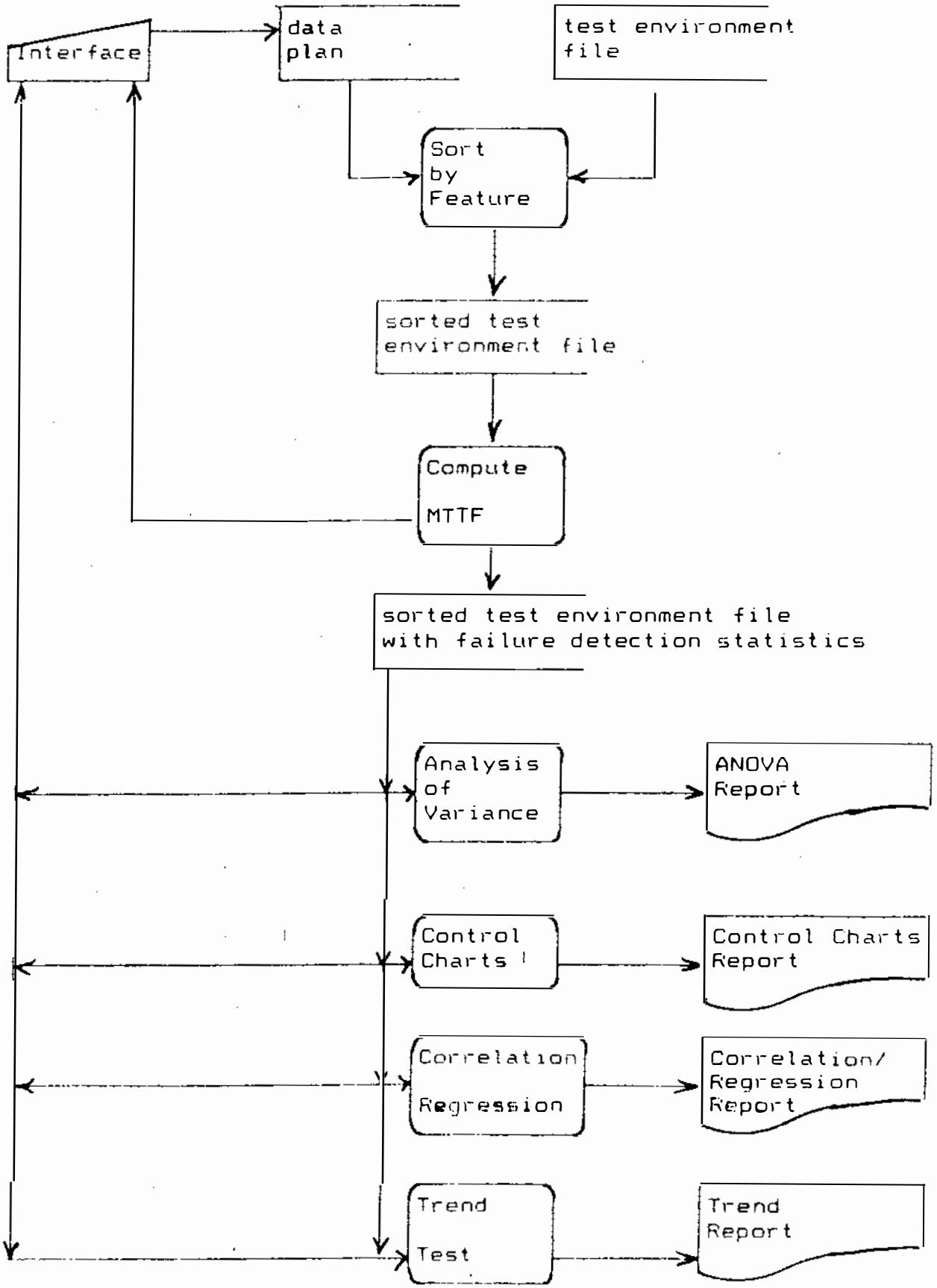

BUG FILE
TEST FILE
(test transaction & environment data)
TEST PERSONNEL
INTERFACE
(consolidate command)
CONSOLIDATE ENVIRONMENT INFORMATION FILES
(report options)
(environment data only)
ENVIRONMENT FILE
(environment data only)
ANALYZE ENVIRONMENT INFORMATION FILE
(analysis results)
(trend data)
(outlier data)
PATTERN REPORT
EXCEPTION REPORT

Figure 4


Interface
data plan
test environment file
Sort by Feature
sorted test environment file
Compute MTTF
sorted test environment file with failure detection statistics
Analysis of Variance
ANOVA Report
Control Charts
Control Charts Report
Correlation Regression
Correlation/ Regression Report
Trend Test
Trend Report

VI REFERENCES

(Anonymous)
FormTool Version 2.0 User Manual
Miami Beach, FL: BLOC Development Corporation, 1986

ANSI/IEEE
Std 829-1983: Standard for Software Test Documentation
New York: Institute of Electrical and Electronics Engineers, 1983

ANSI/IEEE
Std 1008-1987: Standard for Software Unit Testing
New York: Institute of Electrical and Electronics Engineers, 1987

Arthur, Jay
"Software Quality Measurement"
Datamation, 15 December 1984, pp 115-120

Bernstein, Amy
"Putting Software to the Test"
Business Computer Systems, December 1984, pp 48-51

Dixon, W.J. & Brown, M.B.
BMDP-83: Biomedical Computer Programs
Berkeley, CA: University of California Press, 1983

Gaydasch, Jr., Alexander
Principles of EDP Management
Reston, VA: Reston Publishing Co., 1982

Gress, Larry M.
"Eight Ways To Rate Good Documentation"
Small Systems World, November 1983

Leavitt, Don
"The Why's and How's of Software Testing"
Software News, June 1985, pp 65-67

Norusis, Marija J.
SPSS Introductory Guide
New York: McGraw-Hill, 1982

Scheiber, Stephen F.
"In Search of ... Software Reliability and Testing"
Test & Measurement World, November 1986, pp 18-23

Walsh, Robert J.
"The Testing Cycle"
DEC Professional, November 1985, pp 102-115

Walter, John P. & Dawson, Peter P.
Analysis and Design of Business Systems
Long Beach, CA: Kinko's, 1987

Find and Fix Rates
A Case Study of a Basic Quality Measurement.

AT&T Information Systems
Millard J. McQuaid
Walter M. Wycherley

ABSTRACT

The collecting and graphing of defect report data has been done for some time. The typical curve used shows a cumulative number of defects found over time. Most developments plot this cumulative data to see the rate at which new problems are being found. It is not only the rate of finding problems, it is the rate of fixing problems which is important. The effects of using find and fix rate measurements on a large real time software product are discussed.

BIOGRAPHICAL SKETCH

After receiving a B.S. in Mathematics from Westminster College (PA.), Millard worked for Computer Sciences Corporation, Systems Division. There he designed and programmed simulations for large scale communication systems. Millard attended graduate classes at both George Washington University and American University, receiving a M.S. Computer Science in 1978. He then moved to Denver, Colorado to work for Bell Laboratories and has been involved in system testing real-time communications software since that time. He is currently a supervisor of a system test group at AT&T Information Systems.

Walt has been in the computer business for over 26 years. His background has been primarily in realtime systems software. He worked for Univac on NASA's Apollo project and as a communications software consultant for both Northern Telcom and Siemens. Walt also was involved with the software development of General Telephone's stock brokerage system. He is one of the founders of ComSult, a small systems consulting firm. Since 1983, Walt has been working for AT&T Information Systems on real-time communications software testing.

Walter M. Wycherley
11900 N. Pecos Ave.
Denver, Colorado 80234
(303) 538-4346

Millard J. McQuaid
11900 N. Pecos Ave.
Denver, Colorado 80234
(303) 538-4346

Find and Fix Rates
A Case Study of a Basic Quality Measurement.

INTRODUCTION.

The collecting and graphing of defect report data has been done for some time. The typical curve used shows a cumulative number of defects found over time. Most developments with which we are familiar plot this cumulative data for the sole purpose of watching the slope of the curve. That is, to see the rate at which new problems are being found. While useful, this information only gives a partial picture. It is not only the rate of finding problems that gives an indication of how well the software development has worked, it is also the rate at which these problems can be fixed. The fix rate is manageable from the standpoint of scheduling fixes to occur at desired points.

Serious software defects must be fixed at least as fast as they are being discovered in order to reach the best possible quality level within a project's time constraints. Yet, most developments do not track find and fix rates as standard measures. This simple mechanism has been used with some success on several related software projects.

The software being discussed is a large real-time oriented system. Each software project is an incremental addition to the previous product. A new project typically contains in excess of 100K lines of new and changed code. The development methodology includes formal reviews of design and code reading. A formal system test group exists and all changes must be approved prior to release to a customer site.

Data has been collected on four projects. The first did not use find and fix rate tracking. The three subsequent projects evolved toward that measurement. These find and fix rates have been used to size intermediate deliveries to system test and to help decide the timing of deliveries to customers. An additional benefit has been a behavioral effect in both the development and testing organizations.

The find and fix rate is a useful project management tool to affect the quality of the product delivered to the customer. Effective management of these rates has shown that the testing interval can be reduced while positively impacting the quality.

BACKGROUND.

Product.

The product studied is a complex real-time system. The development effort consisted of four major projects, each

representing a substantial addition of functionality. Each major project is self sufficient, that is, capable of being sold to an identified customer population as a stand alone product. Subsequent projects provided a mix of existing product improvements and additional capabilities aimed at providing more sophisticated functionality. Improvements and fault fixes were provided to customers by periodic field updates.

Environment.

The production environment consisted of separate organizations for architecture, software development, hardware development, system testing, and field support. The four software projects we tracked were quite similar in nature. The management was basically the same across the projects, as was the staff. Each project was on the same processor with similar supporting hardware. The types of faults encountered were much the same as those mentioned in any good text on software testing [1] (although upon several occasions we experienced some truly innovative fault creation techniques). Problems detected in the system software were formally documented via a problem tracking and management system.

The product followed a classic transition through a standard life cycle[2] ,[3] that merged the final version at the System Testing organization. At this point, while functional and System level testing was in progress, corrections to faults were incorporated into an updated version of the software and were "recycled" back to system test. Some percentage of these modifications caused additional problems which then were corrected. In addition, the testers were required to repeat some number of regression tests.

A scheduled plan of field testing the product at a set of customer locations provided the opportunity for realistic testing. These field trials gave insight to the customer's perception of the product and identified configuration and usage dependent problems. These problems were resolved and the software changes tested.

Once the field testing process was completed, the product was officially offered to the general customer. Again, unique configurations and idiosyncrasies of customer usage produced problems that required software changes. Periodic updates were made available to customers to resolve these issues. The system test organization managed the fix schedule and the planning of upgrades to the customer. These upgrades also contained enhanced capabilities suggested by the customer, or identified internally as a result of marketing analysis.

While one project was being field tested, another was under development. (Figure 1) This timing is important because the same staff was responsible for the maintenance of previous projects

and development of the current product. Also, measurement techniques were being refined based upon experience. New projects benefited from these changes while existing projects could not.

PRODUCT CHRONOLOGY

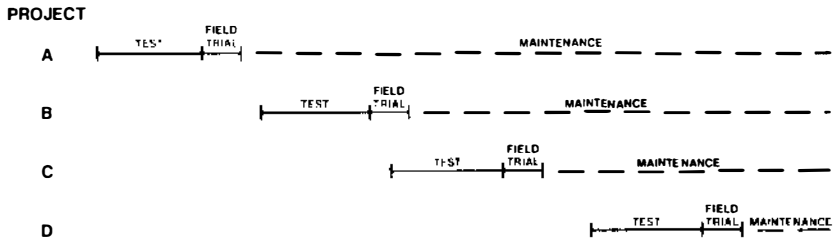


FIGURE 1

THE EVOLUTION OF A MEASUREMENT.

Project A - The first step

Project A was well into the test phase of development when the need for a more sophisticated quality measurement was identified. A number of different efforts were initiated, and it would be difficult to distinguish the impact of each. It is clear that the measurement evolution described here had a major role in the results reported.

The initial measurement used in Project A for tracking was a cumulative total of the number of problems found on a weekly basis. (Figure 2) This kind of data collection has long been recommended. Many developments track similar factors or use this data to calculate some trackable measure.[4] ,[5] As the total number of problems increased, the scale of the graphic became smaller, causing a loss in resolution. Any significant events that could be determined by the measurement became vague. The point on the curve which was supposed to indicate a reduction of new fault discoveries, the "knee" of the curve, was impossible to detect.

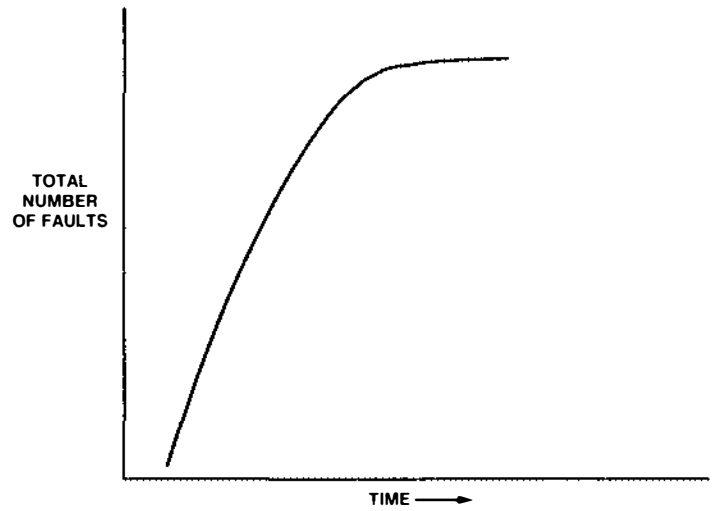


FIGURE 2

The typical development delivery in Project A was characterized by a large number of fixes in a single load. The system test organization would test the load, find newly introduced faults, identify failed fixes, and turn the load back to development. The results of this "Big Bang" approach of large, infrequent deliveries was obscured by the cumulative fault curve. Deliveries were separated by a significant interval, so that testing resources were being used excessively at some times and not at all on other occasions. In addition, due to the increased testing, a number of new problems were discovered after each load was delivered. The product was then re-fixed and system tested. This cycle may have been repeated several times, depending on the extent of changes required.

Project B - An Obvious Truth

Most of the people consulted used the cumulative curve primarily to determine the rate at which new faults were being discovered. Plotting the find rate directly, then, yielded a more useful picture. It became apparent, once this was done, that the find rate was an indication of the implementation processes and the testing efficiency. While interesting, the current implementation was complete. This tracking revealed process defects that could only be corrected in the next project. It could not improve the current project.

One factor that could influence the current project's quality was testing efficiency. Since fault find rate was a measure of this factor, Project A's fault find rate was plotted and this information was communicated to the test organization's supervision. (Figure 3.) Analysis of Project A's find rate showed that it peaked late in the test period and continued at a

substantial level throughout the project's life. This was because of the delivery procedures previously discussed and the number of faults discovered late in the test process.

PROJECT A & PROJECT B FIND RATES

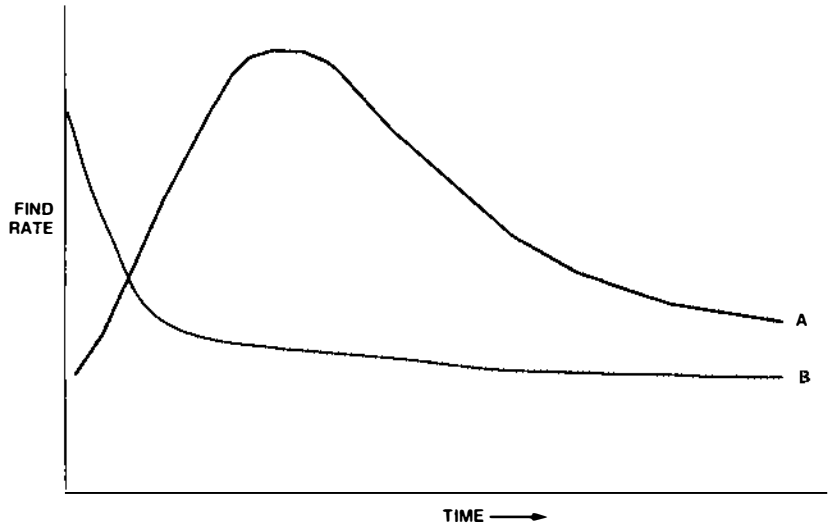


FIGURE 3

A decision was made to smooth the use of the test resources in Project B by offering metered incremental deliveries. Each delivery was planned so that it would contain complete fixes or enhancements. Each could have been delivered to the field as a sub-project. The result was that testers received a more consistent flow of work and that newly introduced problems were found and fixed earlier. The find rate measurement showed this as a more steady flow of new problems but at a lower rate than in Project A. Project B's find rate decreased very slowly after an initial drop.

The incremental deliveries succeeded in creating a number of new questions. The behavior observed was a more regular, consistent testing of the product. This benefited both the development and the test organizations. The measurement still depicted only development history and system test's current efforts. There was a need to depict development's involvement in the ongoing support of the project. In addition, the long tail of the curve was troublesome. Faults were not being discovered early enough in the test process.

It was logical that if the find rate indicated system test's work, the fix rate would indicate development's current efforts. Much consideration was given to the effect of this measurement. It was determined that the behavioral response to a fix rate measurement would be an increased awareness of the need to repair faults in a timely manner. Thus, starting late in Project B, both find and fix rates were tracked. (Figure 4.)

PROJECT B FIND/FIX RATE

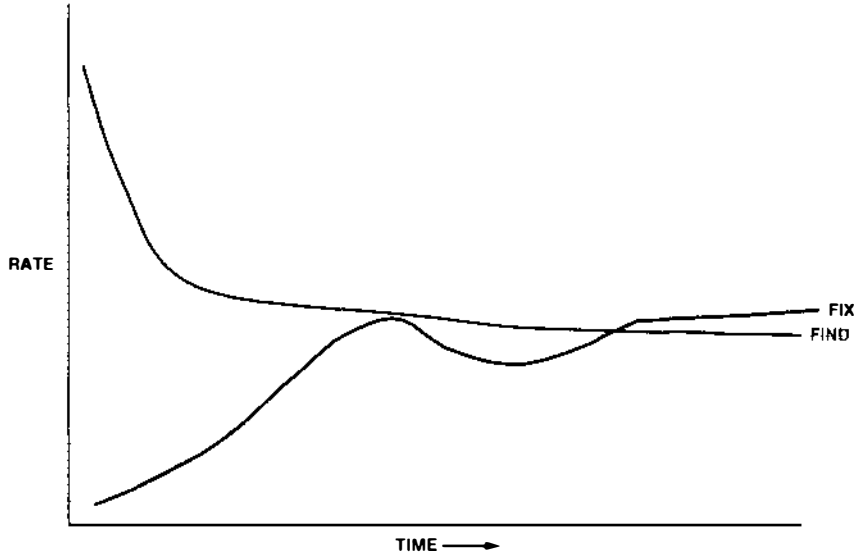


FIGURE 4

A natural but unrealized fact was discovered shortly after the fix rates were placed upon the same graph as the find rates. In order to really improve software quality quickly, the fix rate must exceed the find rate. It is obvious, when stated this way, but this form of tracking is seldom, if ever, used as a means of measuring progress toward the elusive goal of zero defect code. The fix rate, after all, may be the more critical of the two. Too many changes per unit of time cause the reintroduction of faults and impacts the software stability. Too few fixes, on the other hand, prevent the software from improving at a reasonable rate.

As a direct result of the find/fix measure, an effort was mounted to increase the number of fixes on the incremental deliveries to system test. The development organization started to fix defects faster than the test organization was finding them. This resulted in a type of "quality metric" usable by management. If the fix rate was ahead of the find rate over the test interval, it was an indication that the testing effort was not producing more problems than could be reasonably handled by the programming staff. This meant that new, serious problems could be fixed expeditiously. If, in addition, the find rate compared favorably with preceding projects, the expected field performance of the new project could be predicted from current field experience.

The median time to fix faults (from the time the fault was documented until it was on a project usable by the field) was reduced by 35% between Project A and B. It had yet to be determined whether this was the result of more attention to the process ("Hawthorne effect") or whether the push to fix faster than the find rate had caused some change.

Project C - Further Tuning

As the next project was being delivered to system test, there was an indication that better software was being produced. By maintaining the fix rate greater than the find rate, the backlog of Project B problems to be addressed was smaller than in previous releases. This allowed development staff more time to address Project C issues. Efforts were proceeding within development to prevent errors from getting into the design and code, and those problems that were discovered by the test organization were being handled before a customer received the product.

During the testing of Project C, it was established that the fix rate tracking was important and that managing it in this way was reasonable. Project B had been a success and reports of faults discovered in the field were significantly reduced from previous projects. Complaints from the test organization, however, were growing about the number of fixes flowing to them on each incremental delivery. It was felt that there must be an optimum size and frequency of delivery to keep the test organization productive and the fix rate sufficiently high. Some analysis of Project B lead to the conclusion that if the fix rate was maintained at a calculated percentage higher than the find rate, all the known software faults could be fixed prior to the first delivery. The specific percentage used was determined by the scheduled test interval. The implementation involved altering a delivery's size and frequency to maintain a higher fix rate and still provide a reasonable time for development and test to respond to problems.

A result of the increased fix rate was increased testing activity. Fixes required to allow system testers to complete test plans were available earlier. New faults introduced by these fixes were detected and corrected earlier. Thus, the find rate curve had a peak and a decline rather than the continuous tail as in Project B.

After some analysis of the data, it was decided that another key to the increased find rate rested in the theory that "bugs" have a tendency to "nest".[6] After uncovering a fault, and inserting an appropriate fix, it was highly probable that another fault would manifest itself in the same area of code. The sooner this subsequent fault could be detected, the better the end product would be.

Using this measurement technique on Project C, produced the best quality project to date, as measured by the find/fix criteria. (Figure 5.) This was supported by field experience. In addition, the median time to fix was reduced by 15% over Project B.

PROJECT C FIND/FIX RATE

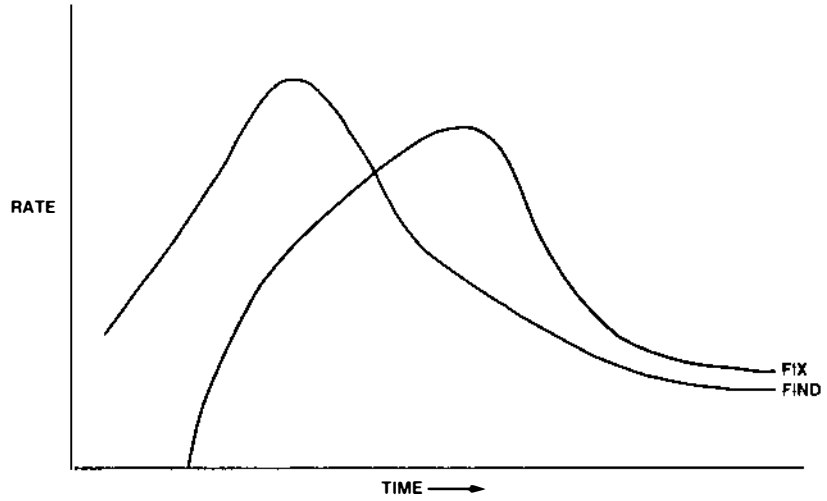


FIGURE 5

Project D - Validation of the Technique.

The measurements outlined above have had a significant impact on the projects discussed. In the final project, D, find/fix tracking and controlled incremental deliveries to system test were used to continue to reduce the median time to fix problems. Project D's median was 18% less than Project C. Some analysis of the Project C data showed the peak find, and corresponding peak fix, occurred fairly late in the development/test interval. If the find rate and corresponding fix rate peaks could be forced earlier in the testing, a reduced test interval could be achieved. Also, the development staff could be reduced to maintenance levels earlier than in previous projects. An effort was mounted to force the peaks to occur in the early system test period via tester education and early integration testing. This effort was successful, and the peak find rates were approximately 10 weeks earlier in Project D than in Project C. (Figure 6).

PROJECT B, C, D FIND RATES

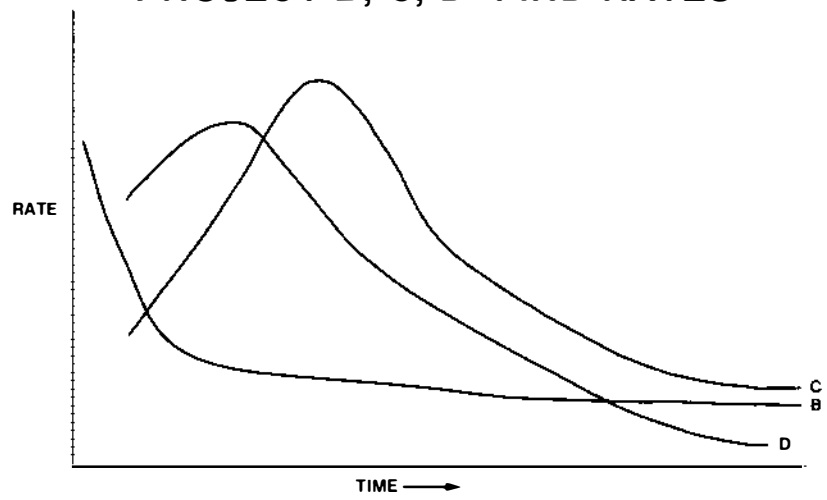


FIGURE 6

One result of this movement was negative, however. Previous find/fix peaks occurred late enough that the major test effort was completed prior to the major fix deliveries to the test organization. By moving the find/fix earlier, the major fix deliveries are made before testing is complete. Thus, the test organization is trying to use the same staff for two different tasks simultaneously. This problem has yet to be solved.

THE MEASUREMENT.

Developing the Metric.

Some form of "smoothing" of the find measurement was needed to accommodate the irregular data and still provide an indication of either improvement or degradation of the product quality. The decision was made to use a floating average to calculate the find/fix rate.

This measurement technique solved many difficulties in monitoring the software product. It provided a simple smoothing for the periods when literally no testing effort was employed, and for those short, sudden aberrations of increased activity. It allowed us to avoid the possible perturbations caused by using absolute numbers of problems found and/or fixed at any given point in time. The specific interval used became the **window** from which trends in overall quality could be derived. The measurement was also used to compare the current project with other projects being developed within the same organizational and cultural environment.

How it Works.

The Floating Average represents a **snapshot** of the most recent window. It can be perceived as an "n" position shift register with each position representing the number of faults found, or fixed, for a given sampling. In each interval, the totals for the most recent window are averaged to produce a new measurement. For each calculation, the key factor is the difference between the latest sample and the sample just excluded from the window. If the newly added data is larger than the "stale" data just ejected, the new rate will show an increase over the previous calculation.

N POSITION SMOOTHING OF FIND/FIX RATES

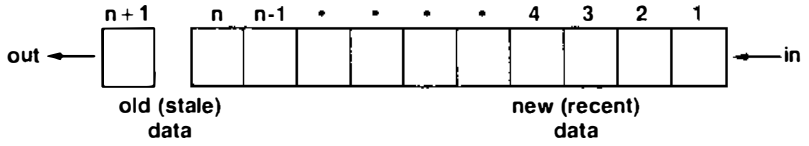


FIGURE 7

The weight of the data in the "n" positions reveals, in itself, a trend. If values are higher at the "stale" end, the general trend is decreasing.

Find Rate.

Tracking of find rate begins as soon as formal testing is initiated on the project. Finding problems that exist in the software product cannot be manipulated other than by changing the testing effort. Fault finding drives the overall process, and as long as there are a significant number of faults left in the software, the testing effort should discover them at a reasonable rate.

An improving quality is demonstrated by the continued ability to test and downward movement of the find rate. Unlike the cumulative fault tracking chart, the find rate plot reveals a substantial and recognizable peak. This occurs when the trend of problems found begins to level off and decrease. A leveling and sustained decreasing trend, coupled with the ability to continue to make testing progress, supports the improved quality notion.

Fix Rate.

Fix rate and find rate tracking begin at the same time but, unlike find rate, the fix rate can be manipulated to accommodate staffing, testing interval, and delivery requirements. When formal testing begins, faults are relatively easy to find and the backlog of unresolved faults increases at a rapid rate. The resolution of these faults requires management in order to prioritize serious problems and to allow testing to continue.

Effects on the Product

From a product view, the use of the find/fix measure, coupled with the other changes being made by the development organization, had a significant effect. Project D became stable, as indicated by the peak find rate, approximately 10 weeks earlier than the previous project. The median time to fix a problem was reduced by 55% from Project A to Project D.

The product's performance in the field has continued to improve throughout the interval we have discussed. This is measured by the number of field initiated changes as well as the perceptions of the users. As a result, the ongoing support requirements are smaller than they have been in previous projects. (Figure 8)

NUMBER OF KNOWN FAULTS OPEN AFTER SYSTEM TEST

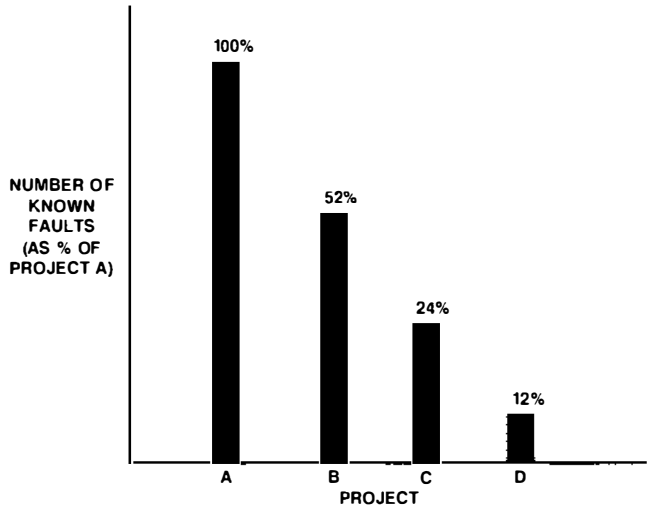


FIGURE 8

Effects on the staff have been observed also. As C reached a low change rate very early, programmers could concentrate on new design and development for Project D without consistently fighting "fires". This is a practical and cost efficient benefit of improving software quality. An important aspect of this simple measurement is the control each group has on it. If testers, being naturally pessimistic, do not feel the software is good, the burden of increasing the find rate rests on them. The developers can maintain the required level of fix rate no matter what the test organization does, simply by fixing more problems. This friendly competitiveness allows each group some control of the final measurement.

Another side effect is the renewed programmer interest in software maintenance. Since this measure is being used by management in the maintenance cycle, there is less motivation to get on with the "new design" to the detriment of the current maintenance.

CONCLUSION.

Find rate is a function of both the development process and the testing effort. The fault density must be managed during the design and code process and is already defined at the delivery to the testing organization. It cannot be managed during the test interval. Fix rate is manageable during this time and, indeed, must be controlled to assure quality software.

These measurements have proven to be useful over the series of projects described here. These measurements are easily produced and intuitively valuable. As with any new measurement that will have an impact, the programmers and testers involved have to

understand what is being measured. Once this occurs, they will adapt their activities to maximize the measure. The find/fix tracking is a fairly simple concept, but requires a change of culture to make it effective.

Once the critical problems have been addressed, attention can be directed toward the elimination of backlog. New severe faults can be fixed as they occur, and the less important ones dealt with as need and staffing permit. If progress is not being made, the fix rate trends will indicate it and additional staff or management direction may be required. The fix rate provides indications as to the effectiveness of the methodology, and provides information indicating when and where methodology could be improved.

Several project decisions were made based upon the measure. In some instances, programmers were pushed to provide more fixes, in others, testers were asked to complete testing faster. The result, however, was a long term gain as both organizations were able to release personnel to other projects sooner. Fewer staff were held hostage to software requiring large amounts of maintenance in the field.

As it became obvious that the measure was useful, tools were developed to obtain and report the data on a timely basis. This was done using several simple data base systems and evolved in parallel with the measure. Once the measure was commonly adopted, timely, accurate data became essential.

Future projects will continue to use the described methods of tracking find/fix rates as a measurement of the software development/test process. Several other smaller projects have adopted the measure with some success.

There are several other ideas to be investigated. It may be possible to predict, based on find/fix data and previous experience, when the software is of sufficient quality to stop testing. It may also be reasonable to plan development staff reductions based upon this prediction. In addition, some normalization of the historical data could yield a picture of the "typical" development. This may be useful in deciding test intervals during the planning of new projects. Characterizing new developments by their peak find/fix values may serve as some indication of the expected quality of the new software.

REFERENCES

1. Boris Beizer, *Software System Testing and Quality Assurance* (New York: Van Nostrand Reinhold Company, 1984), pp.22-32

2. William E Nowden, "Introduction to Software Validation," Tutorial: Software Testing and Validation Techniques (1981), p.2

3. Philip W. Metzger, *Managing a Programming Project*, (2nd ed.; New Jersey: Prentice-Hall, Inc., 1981) pp. 12-15

4. C. V.Ramamoorthy and Siu-Bin F Ho, "Testing Large Software with Automated Software Evaluation Systems," Tutorial: Software Testing and Validation Techniques (1981), pp.182 - 183

5. Glenford J. Myers, *The Art of Software Testing* (New York: John Wiley and Sons, 1979), p.126

6. Glenford J. Myers, *Software Reliability Principles and Practices*. (New York: John Wiley and Sons, 1976), p.193

HOW TO DECIDE WHEN TO STOP TESTING

Elaine J. Weyuker
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, New York 10012

ABSTRACT

General principles for software test data adequacy criteria (rules used to decide when to stop testing) are introduced. For each principle an example is given of a criterion which does not satisfy the principle and an evaluation of three proposed adequacy criterion is made based on the principles.

BIOGRAPHICAL SKETCH

Elaine Weyuker is currently an Associate Professor of Computer Science at the Courant Institute of Mathematical Sciences of New York University, and the Associate Director of Graduate Studies. She is also a consultant for several large corporations.

She began her career in computers as a programmer for Texaco, was a systems engineer for IBM, and was a Lecturer of Computer Science at the City University of New York, before joining the faculty at N.Y.U.

Her major research interests are in the area of software testing, and has authored many papers in the field. She is currently involved in the design and implementation of a testing tool, ASSET, which relies on data flow information as well as the more usual control flow information, and a second tool, SPECMAN, which is particularly useful for managing system level testing of systems with large evolving specification documents.

Professor Weyuker received a B.A. in mathematics from the State University of New York at Binghamton, an M.S.E. in computer and information science from the University of Pennsylvania, and a Ph.D. in computer science from Rutgers University.

HOW TO DECIDE WHEN TO STOP TESTING

Elaine J. Weyuker
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, New York 10012

INTRODUCTION

Software testing comsumes a great deal of resources. By some estimates it typically exceeds 50% of the total software budget. One would expect that this would be ample incentive for the development of good test data selection criteria and good test data adequacy criteria. And yet companies continue to use ad hoc techniques for testing, and frequently assign their weakest employees and inexperienced new hires to quality assurance and system test organizations. As for adequacy criteria, (i.e. rules used to determine whether or not it is reasonable to stop testing) probably the two most widely used criteria are, as Myers reports [My], time and money. We test until the delivery date or until the testing budget has been expended. But what does a calendar date or bank balance tell us about the quality of software? Clearly very little.

This paper, therefore, takes as given that good test data adequacy criteria are needed, and argues that we desperately need general principles we can use to assess the worth of a proposed adequacy criterion. When a new adequacy criterion is being considered, we must have some yardstick to measure it against, so that we can objectively determine whether it is a good, usable criterion. In addition, such a set of principles could be used as a basis for defining new criteria.

PRINCIPLES FOR TEST DATA ADEQUACY CRITERIA

In this section we introduce properties which we believe are fundamental in the assessment of proposed test data adequacy criteria. These criteria, in turn, are the basis for assessing whether it is reasonable to stop testing a program. The properties we propose should, for the most part, conform to an experienced tester's intuition. They are not based on any deep theory, but rather arise from empirical evidence and are justified by our experience.

For each principle, we shall first present a brief argument to convince the reader that it is a characteristic which "good" adequacy criteria should satisfy, and then state the principle. Following that, we present an example of a criterion which *does not* satisfy the principle.

This research was supported in part by the National Science Foundation under grant DCR-85-01614, and by the Office of Naval Research under contract N00014-85-K-0414.

We assume a structured programming language in which programs are single entry/single exit. We also assume that all input statements appear at the start of the program, and all output statements appear at the end. This is simply a technical convenience which will be useful when we discuss program composition later in this section.

Test data adequacy criteria can be categorized as being of one of two general types. The first type are known as *program-based* or *structural adequacy criteria* and are characterized by the fact that they rely largely on the program's structure. Two well-known examples of program-based adequacy criteria are *statement* and *branch coverage.* In the former case, a program is deemed adequately tested provided that every statement of the program has been executed at least once by some test case. *Branch coverage*, which is somewhat more demanding than statement coverage, requires that if a program is represented as a graph, then every edge of the graph be traversed at least once by some test case.

The other type of test data adequacy criterion, the *non-program-based* criterion, does not rely on the program's structure. These criteria may depend on the specification, in which case they are referred to as *specification-based* criteria, or may be independent of both the program structure and the specification.

In this paper, we shall present what we believe is a set of fundamental principles that should be satisfied by every good *program-based* test data adequacy criterion, and use these principles to assess the strengths and weaknesses of proposed criteria.

PRINCIPLE 1

Our first principle requires that for an adequacy criterion to be useful, it must be universally applicable. Generally, testers don't get to decide whether or not they care to test a particular program. We must therefore be prepared to test any program, and there must be some point at which we can declare that the program has been adequately tested. That is:

For every program, there must be some way to adequately test it as determined by the adequacy criterion.

What type of adequacy criterion would fail to satisfy such a fundamental principle, and in particular, for what type of program would the criterion fail? Consider the statement or branch criterion and the program fragment:

```
if x ≥ 0 then y ← 0
         else y ← 1
if x × y > 0 then z ← 0
             else z ← 1
```

Since the product of x and y will always be non-positive, the "true" exit from the second decision statement can never be taken for any input value of x. Therefore the statement "z ← 0" can never be executed. Thus, neither statement nor branch coverage can be satisfied for this simple case, regardless of how extensive testing is. Thus, both statement and branch testing fail to satisfy our first principle, because of the possibility of unexecutable code.

PRINCIPLE 2

Of course, one way to test a program is to test it *exhaustively*, that is, on every member of its domain. Occasionally, when the domain is very small, this is feasible or even necessary, but in general the whole point of testing is to select test cases which are representatives of the entire domain. If the test cases were selected wisely, we expect that we can infer from the proper behavior of the program on the test set that the program behaves correctly in general. Thus, an adequacy criterion which said that the *only way* to test programs thoroughly was to test them exhaustively, would be a useless criterion. Our second principle is therefore:

It must sometimes be possible to adequately test a program using less than an exhaustive test set.

Consider the adequacy criterion that requires sufficient test data to distinguish a program from *all* inequivalent programs. Since for every possible input t, there exists a program which differs from the original program only at t, such a criterion would fail to satisfy this principle.

PRINCIPLE 3

Exhaustive testing requires that a program be tested on all possible inputs, and we have just argued that it is not reasonable for an adequacy criterion to *always* require that. The other extreme is not testing a program at all. Since an adequacy criterion is intended to assess the *quality of testing*, if the program has not been tested at all, surely it is not reasonable for an adequacy criterion to deem it thoroughly tested. In particular, a program may be correct (i.e., produce the required output on every possible input) and still not be adequately tested. Correctness is a quality of a program, relative to a specification. Adequacy, in contrast, is a property of a program, specification, *and* test set. We therefore conclude:

An adequacy criterion should never assess an untested program as adequately tested.

Consider the following criterion: a program is adequately tested provided it has been tested on every input which produces an incorrect output. In one sense, this criterion is perfect. Whenever a program has been adequately tested according to this criterion, it is guaranteeed that all the faults have been

located. Of course, if we knew where all the faults were, we would have no need to test the program. This criterion fails to satisfy the above principle since a correct program requires no testing. The criterion confuses program quality with the quality of testing. Note also that there is no way of determining, short of exhaustive testing, whether this criterion has been satisfied.

PRINCIPLE 4

How is an adequacy criterion typically used? Generally, in practice, a tester derives test cases until he or she believes that the testing has been thorough. It is at this point that an adequacy criterion should be invoked. If the criterion reports that the program has not been adequately tested, testing should continue. But if a program is deemed adequately tested by a criterion, testing usually ceases. After all, the adequacy criterion has certified that enough testing has been done. But what if some additional test cases are run anyway? Should it be possible that the adequately tested program is now assessed to be inadequately tested? Recall that an adequacy criterion is used to assess the quality of testing, not the correctness of the program. Therefore, if additional test cases expose program faults, they should certainly be removed, just as they would be if the program had not been considered adequately tested. Although such an occurrence might cause us to reevaluate the usefulness of the adequacy criterion, it should not change the assessment of adequacy relative to the chosen criterion. The criterion is assessing the quality of testing, not the program's correctness. Therefore, our next principle is:

If a program has been adequately tested using a given criterion, then the addition of extra test cases, regardless of the program's outcome on them, cannot cause the program to be considered inadequately tested.

An example of a criterion which fails to satisfy this principle might require the outcome of every test case to be correct. Suppose the program has been tested using ninety nine carefully selected test cases, which satisfy all the requirements of the adequacy criterion, including the requirement that the correct output be produced for every test case. The program would then be considered adequately tested. Now suppose the hundredth test case produces the wrong output. Then, according to this criterion, the program is no longer adequately tested. The fact is that the purpose of testing is to uncover faults in the program, and a criterion which fails to satisfy this principle rewards the tester for *not* locating faults!

PRINCIPLE 5

The principles we have presented so far pertain to *any* test data adequacy criterion, not just program-based ones. We now consider principles more

specific to program-based criteria. The first thing to recognize is that a program-based criterion depends primarily on the *program*, not just the function it computes or the specification it is intended to satisfy. Therefore, it may well be the case that two programs which have the same input/output behavior, will nonetheless require different test cases in order to satisfy a given program-based adequacy criterion. That is:

Even though two programs may compute the same function, they may have to be tested differently.

An adequacy criterion which is essentially a form of random testing would fail to satisfy this principle. Let k be an integer greater than 0. We shall say that a program has been *k-adequately tested* provided it has been tested on at least k points. That is, every size k test set is k-adequate for every program. This is clearly the antithesis of a program-based adequacy criterion, and therefore it is not surprising that it fails to satisfy this principle.

PRINCIPLE 6

Suppose that a program has been divided into subprograms, and that each of these subprograms has been adequately tested according to some adequacy criterion. Does it necessarily follow that the entire program should be considered adequately tested according to the given criterion? A superficial consideration of this question might lead us to conclude that the answer is "yes". However deeper reflection convinces us that the correct answer is "no". The reason is that by only testing individual components of a program in isolation, the added interactions and interfaces which should be tested as a result of the composition, are ignored. Intuitively we want to say:

Let P be a program composed from components P_1 and P_2. It is not necessarily sufficient to test P_1 and P_2 in isolation.

It is necessary, however, to be precise about what we mean by "composition," and which test cases are intended when one program is composed with another. Let P_1 and P_2 be programs using the same set of identifiers. We shall write $P_1;P_2$ to mean the program formed by replacing P_1's unique exit and output statements by P_2 with P_2's input statements deleted.

Let T_1 be adequate to test P_1 and let T_2 be adequate to test P_2. The intuition we wish to capture is to consider whether $T_1 \cup T_2$ is adequate for $P_1;P_2$. But if the test set T_2 were input to $P_1;P_2$, the values that arrive at P_2 would not, in general, be T_2. Rather, we want to find a test set T_3 such that T_2 is the union of the vector of values produced by P_1 for each member of T_3. We denote this by $P_1(T_3) = T_2$. Alternately, we can choose a test set T such that T contains test cases which are adequate for P_1 plus test cases such that P_1(T) is adequate for P_2. Principle 6 formalizes the intuition that it is not necessarily sufficient to test the two components P_1 and P_2 in isolation.

There exist programs P_1 and P_2, and a test set T, such that T is adequate for P_1, and $P_1(T)$ is adequate for P_2, but T is not adequate for $P_1;P_2$.

Considering, again, the statement and branch coverage criteria, we see that they do not satisfy this principle. Whenever T causes every statement (branch) of P_1 to be exercised, and $P_1(T)$ causes every statement (branch) of P_2 to be exercised, then necessarily every statement (branch) of $P_1;P_2$ will have been exercised by T.

PRINCIPLE 7

Now let us consider the inverse situation. Suppose a program P has been adequately tested by a test set T, using a given criterion. Does it necessarily follow that each of the components that comprise P have been adequately tested? Again, a casual response might be "yes", but deeper consideration convinces us that the proper response is "no".

First, suppose P is a program which contains a sorting routine as one of its components, and suppose further that P's processing of inputs is such that regardless of what vector of values is input to P, when the data arrives at the start of the sort component, the values are always in reverse order. Therefore, regardless of what test cases are selected for P, the sort routine can only be tested *within the context of P* on data of one form. Now even though one might construct a test set which is adequate to test P, it would be difficult to imagine that a plausible adequacy criterion would deem the sort routine adequately tested outside of the context of P.

An even more extreme situation occurs when P contains some unexecutable code. It may either be the case that this was done deliberately, as when a feature has not yet been "hooked up", or it may be a sign of an error. Nonetheless, it is a situation which occurs frequently. Now suppose P has been adequately tested using some criterion. Clearly the unexecutable component has not been executed at all by any test case, even if P has been exhaustively tested. Is it reasonable to consider the component adequately tested? Would one feel confident using it in some other context with no testing? Clearly the answer to both of these questions in "no". Therefore, our next principle is that context counts!

Having adequately tested a program, it does not necessarily follow that each of its components have been adequately tested for other contexts.

Again, the statement and branch coverage criteria do not satisfy this principle. Suppose T causes every statement (branch) of P to be exercised, and Q is a component of P. Since every statement (branch) of Q is a statement (branch) of P, it follows that every statement (branch) of Q has been exercised, contradicting this principle.

PRINCIPLE 8

Suppose you had a print routine consisting of a few lines of code and a payroll routine consisting of several hundred lines. Would you expect them to be tested the same way? Would you expect the test cases to be the same? Which would you expect would require more testing? Virtually everyone would answer that the payroll program was more complex, and would therefore require more testing than the print routine. A program-based adequacy criterion should certainly reflect these facts. There are many different ways that one could measure program complexity, and there has been a great deal of discussion about what is the most appropriate way to do this. Without selecting such a measure, however, our intuition nonetheless tells us: complex programs should require a lot of testing; simple programs should require much less. Clearly, an adequacy criterion which says that every program should be tested the same way is not satisfactory. In fact, we can really make a stronger statement than that. For example, if someone proposed an adequacy criterion which categorized all programs as being in one of two classes: those requiring n test cases and those requiring m test cases, that "two tiered" adequacy criterion would be no more acceptable than a "single tiered" criterion which said that all programs require k test cases. Our next principle reflects this intuition:

For every program P, there are programs which require more test cases than P to be adequately tested.

The random testing criterion defined earlier, k-adequacy, does not satisfy this principle since every program is adequately testable by a test set of size k.

PRINCIPLE 9

Now all of the principles presented so far capture qualities that we want a "good" program-based test data adequacy criterion to possess. But our principles have *not* really described the essence of a program-based test data adequacy criterion. That is, what do we expect such a criterion to tell us? Is there a single property which rules out anything which is not really an appropriate program-based adequacy criterion? Is there some sort of a central principle?

If we look at different proposed program-based adequacy criteria, one basic pattern emerges. All of them, whether they seem good or poor, have a common philosophy: in order for a test set to adequately test a program, it must cause every part of the program to be exercised at least once. After all, if some part has never been executed, anything could be computed there and we would be completely ignorant of any faults.

Myers [My], in his early book on reliability states as one of his fundamental tenets of testing: "one basic criterion for a set of test cases is ensuring that they cause every instruction in the module to be executed at least once. The criterion is certainly necessary, but it is not the place to stop." As we shall

discuss in the next section, we agree with Myers, that statement coverage "is not the place to stop", but we also agree that in spite of its weaknesses, it is a necessary condition. Therefore we propose as our final, and most fundamental principle:

If a test set T is adequate for a program P, then T causes every executable statement of P to be executed.

Once again k-adequacy fails to satisfy this principle. This is not surprising since k-adequacy is clearly not a program-based adequacy criterion. It is, in essence, a "nothing-based" adequacy criterion. It is independent of the program's structure as well as the specification.

EVALUATING PROGRAM-BASED CRITERIA

Having set forth our principles for program-based test data adequacy criteria, we now summarize the evaluation of three adequacy criteria: statement and branch coverage and k-adequacy, to see how they stack up.

PRINCIPLE	STATEMENT/BRANCH COVERAGE	K-ADEQUACY
1	no	yes
2	yes	yes
3	yes	yes
4	yes	yes
5	yes	no
6	no	no
7	no	yes
8	yes	no
9	yes	no

We see that the statement and branch coverage criteria satisfy six of the nine principles we have proposed, a rather disappointing "score" for such popular criteria. This analysis should have made clear the weaknesses of these criteria. The first, and more important weakness is their failure to be "universally applicable". Whenever a program contains unexecutable code, these criteria will not be satisfiable. The problem is that when you are informed that your test set has caused only 60% of the statements (branches) to be exercised, there are two possible reasons. The first is that you have done only a mediocre job of testing the program, and substantial additional test cases must be selected. But it is also possible that the reason that some code was not exercised is that that code is not executable, and it is not always easy to discern that this is the situation. Thus the failure to satisfy this

"applicability" principle is fundamental. An adequacy criterion is supposed to provide a means for deciding whether or not it is time to stop testing. But, as demonstrated above, if the criterion does not satisfy this principle, a test set's failure to satisfy the criterion does not necessarily mean that more testing is necessary. Unfortunately the tester cannot in general tell that this is the case, surely an unacceptable situation.

The other weakness of the statement and branch coverage criteria which was exposed by this analysis was their inability to be sensitive to context. If a component is combined with other components, there is no provision for the possible interactions between the components, and no way to take into account a component's context within a program.

Now an interesting question is: what does this tell us? Does this imply that we should not use statement and branch coverage as adequacy criteria? Perhaps, but not necessarily. It should tell us what properties we want to address if we design new adequacy criteria based on these criteria. It warns us that these are criteria with real limitations. But what if there is a statement or branch coverage tool available in-house which is relatively cheap and easy to use? It may well be that this is the only such tool currently available, and that the alternative to using statement or branch coverage in that situation is to use no adequacy criterion at all. In that case, I would argue that in spite of their deficiencies, the use of these coverage criteria was acceptable (certainly better than nothing.)

k-adequacy satisfied only five of the nine principles. It was shown that it does not satisfy the fundamental principle of program-based criteria. What are the implications of this? Clearly random testing is not generally suitable as an adequacy criterion, particularly for unit testing. Effectively, without at least including some provision for varying the amount of testing needed, based on a program's complexity, the k-adequacy criterion is little more than the time and money criteria in a slightly different guise. In essence, it is not an adequacy criterion at all.

CONCLUSIONS

We have proposed a set of principles for when a criterion was appropriate for use as a program-based test data adequacy criterion. The set is not necessarily complete, nor are each of the principles of equal importance. They do provide us with a way of evaluating the strengths and weaknesses of propose adequacy criteria, and also with a basis for defining new and "better" criteria.

REFERENCES

[MY] G.J. Myers, *Software Reliability Principles and Practices*, John Wiley and Sons, New York, 1976.

Session 2

LIBRARIES: FINDING AND RETRIEVING

"Design Documentation Retrofitting: An Approach for Retrieving Reusable Code
Patrick H. Loy, Johns Hopkins University

"Pacific Bell's UNIX/C Library Effort: An Example of a Means of Taking Advantage of Reusability"
Richard I. Anderson and Rebecca Green, Pacific Bell

"Fulcrum: A Reusable Code Library Toolset"
Mary Hsia-Coron, Jennifer Marden, and Eeman Wong, Hewlett-Packard

DESIGN DOCUMENTATION RETROFITTING: AN APPROACH FOR RETRIEVING REUSABLE CODE

Patrick H. Loy
Whiting School of Engineering
The Johns Hopkins University

ABSTRACT

This paper presents a technique for generating architectural design documentation from existing software. This technique has been proven to be effective in assisting the software professional in making decisions about the reusability of the code.

The potential reusability of code is often severely constrained by the quality of existing design documentation. In many cases no graphical design documentation is available; in other cases the documentation is inadequate to assess the relationships between software program modules. To determine whether or not code is reusable, design documentation is needed that graphically depicts the program architecture showing the relationships between program modules in terms of hierarchy, calling sequences, and parameter passing.

An approach is presented that enables the software professional to start with existing code and retrofit design documentation consisting of structure charts and interface details. A case study is discussed in detail.

BIOGRAPHICAL SKETCH

Patrick H. Loy is the Director of Professional Development Programs for the Whiting School of Engineering, The Johns Hopkins University. He also serves part-time on the computer science faculty teaching graduate courses in software engineering. He has several years experience as a consultant and trainer in the areas of software design, testing and quality assurance, and is currently involved in training software acquisition managers at the National Security Agency. He conducts workshops and seminars in the aforementioned areas for government organizations and private companies, and has published papers on software design topics in various journals and magazines. He is a member of the IEEE, the American Society of Engineering Education, and Computer Professionals for Social Responsibility, and is a chapter officer of the ACM. Mr. Loy holds the M.S. degree in computer science from The Johns Hopkins University, and the B.S. degree in mathematics from the University of Oregon. He can be reached at:

Patrick H. Loy
The Johns Hopkins University
7301 Parkway Drive South
Hanover, Maryland 21076
(301) 796-8200

Introduction

Software maintenance claims an extremely large share of the software dollar, and is becoming the most expensive part of the software life cycle. In fact, in some environments maintenance costs are upwards of 75% of the total life cycle costs.[1] One of the biggest problems encountered by software maintainers is retrieving and modifying reusable code.[2]

The term "software reusability" is applied to many techniques, methods, and processes. These include portability, code-sharing in successive releases, common subsystems, common routines in application families, repeated exploitation of algorithms, and modular "black box" program components. The technique described in this paper is relevant to all of these notions, but our paradigm will be the salvaging of code from an existing system to use in a similar system that is being built. Therefore, when a module is referred to as being potentially reusable, it means it is likely it could be easily modified to fit into another program environment.

Identifying potentially reusable code is often severely constrained by the quality of the existing design documentation. In many cases no graphical design documentation was ever created; in other cases the documentation that exists is inadequate to give the software professional the needed "big picture" of the interrelationships between system components. To determine whether or not code is reusable, and to decide to what extent such reusable code must be modified, design documentation is needed that graphically depicts the program architecture. Such documentation must show the functionality of program modules and the relationships between them in terms of calling hierarchy and parameter passing.

This paper first will discuss a method for creating graphical design documentation from existing code. Then a description of an actual project where these techniques were employed will be given. The primary product of the technique is the design structure chart found in the structured design method made popular by Yourdon and Constantine,[3] Page-Jones,[4] Myers,[5] and others.

Creating design documentation from existing code

Figure 1 shows the pseudo-code, based on Pascal, for a program containing eight software modules. By convention, the parameter list associated with a module declaration references the interface between that module and the module that calls it. The input it receives from the calling module is to the left of the colon, and the output it returns to the calling module is to the right. Thus, for example, Process-word receives "word" from its calling module, and returns nothing. Another convention is that, on the structure chart, data items on the interface between modules are named from the point of view of the calling module. This is important since the calling and called modules may not reference a particular interface variable by the same name.

The system in figure 1 is an automated "junk mail processor" that reads characters alternately from a terminal and a card reader, changing input devices when it detects the two-character symbol "*S." When "*E" from the terminal is encountered, input is taken from the cardreader until the end of the file. The system thus allows personalizing a form letter by adding terminal input at specified places. The program creates words from the characters read and processes those words according to a pre-defined format for the letter. Although the above description is quite brief, it is sufficient for our purposes: the exact workings of the system are not important to grasping the use of the documentation retrofitting technique.

The procedure for creating the design documentation involves first determining the relationships between the modules in terms of calling hierarchy, and showing this relationship with a design structure chart. In addition, the data that is passed on the interfaces between modules is shown on the chart. To start this process, we begin scanning the pseudo-code in the order in which it appears, looking for calls to subordinate modules. We first note that the module Textscanner calls two subordinates within its code. Using structured design notation in figure 2, we put the name of the module, Textscanner, in a box and show its

subordinates in boxes underneath it, connecting the boxes with lines to indicate the calling hierarchy. In the call to module Process-word, we see that the interface contains the data item "word," so we show that on the diagram. In the call to Find-word, we note that there are three data items on the interface: "word," "eof," and "errorstop." At this point we do not know the direction of flow of the interface data. However, upon reading the declaration lines of the called modules (Find-word and Process-word) we can determine it and add directional arrows, as shown on figure 2.

Our investigation so far has given us the first two layers of the structure chart, as illustrated in figure 2. Notice that a second call to Process-word and to Find-word within the code of Textscanner does not affect the structure chart: that level of detail is not shown on the diagram as it would add little or nothing to the conceptual view of the program structure, and instead would increase clutter on the chart making it less readable. Also note that the calls to Process-word and Find-word are not mutually exclusive during a given execution of Textscanner. Where calls to subordinate modules <u>are</u> mutually exclusive, such as calls from Case statements or mutually exclusive If-Then-Else logic blocks, a diamond (decision symbol) is shown on the calling module, as in Get-char on figure 3.

By successively examining the pseudo-code for all of the modules in the program as we did with Textscanner we come up with the diagram in figure 3. Notice that we did not have to read the pseudo-code in any particular order to develop the structure chart using this method. By examining the code of all the modules for calls to subordinates we would have created the same structure chart regardless of the order of modules examined. The only difference might have been the relative left-to-right placement of subordinates, which has no bearing on the logic of the chart. Also note that for the purpose of generating the structure chart we did not need to look at the pseudo-code in any great detail.

Now that we have created the structrue chart we can begin examining it for certain attributes that will aid us in determining the reusability of the software, re-visiting the pseudo-code as needed.

```
MODULE Textscanner
   BEGIN
      eof = False
      cptr = 0
      source = term
      stop 1 = False
      errorstop = False
      CALL Find-word (word, eof, errorstop)
      WHILE NOT eof AND NOT errorstop DO
         BEGIN
            CALL Process-word (word)
            CALL Find-word (word, eof, errorstop)
         END
      CALL Process-word (word)
      IF errorstop THEN write ("abnormal termination")
   END


MODULE Find-word (:word, eof, errorstop)
   BEGIN
      eof = False
      errorstop = False
      word = blanks
      UNTIL char = space OR eof OR errorstop DO
         CALL Get-char (char, eof, errorstop)
         IF NOT eof AND NOT errorstop THEN
            BEGIN
               UNTIL char = space OR eof OR errorstop DO
                  BEGIN
                     CALL Build-word (char, word)
                     CALL GET-char (char, eof, errorstop)
                  END
            END
   END
```

Figure 1. Pseudo-code for automated "junk mail processor."

```
MODULE  Process-word (word:)
   BEGIN
   *Process words according to the specifications *
   END


MODULE Get-char (character, stop, errorstop)
   BEGIN
      errorstop = False
      stop = False
      havechar = False
      UNTIL havechar OR stop OR errorstop DO
         BEGIN
            IF source = term THEN
               BEGIN
                  CALL Read-term (character)
                  IF character = '*s' THEN source = cards
                  ELSE
                     IF character = '*e' THEN
                        BEGIN
                           source = cards
                           stop = true
                        END
                     ELSE havechar = true
               END
            ELSE
               BEGIN
                  CALL  Get-card-char (char, errorstop)
                  IF char = "*s' THEN
                     IF errorstop  THEN stop = true
                     ELSE source = term
                  ELSE havechar = true
               END
         END
   END
```

Figure 1 (continued).

```
MODULE  Build-word (character, word:word)
   BEGIN
      *Concatenate character to partial word *
   END

MODULE Read-term (:character)
   BEGIN
      *prompt terminal for a character.  Return the character, or
      '*e' if time-out occurs *

   END

MODULE Get-card-char (: character, card-eof )
   BEGIN
      card-eof = False
      IF cptr ≥ 2 and cptr ≤ 80 THEN
         BEGIN
            character = card (cptr)
            cptr = cptr + 1
         END
      ELSE
         BEGIN
            CALL Read-card (card, card-eof)
            IF card-eof THEN character = space
            ELSE
               BEGIN
                  character = card (1)
                  cptr = 2
               END
         END
   END

MODULE Read-card (:card, eof)
   BEGIN
      * read next card; if none left set eof to true*
   END
```

Figure 1 (continued).

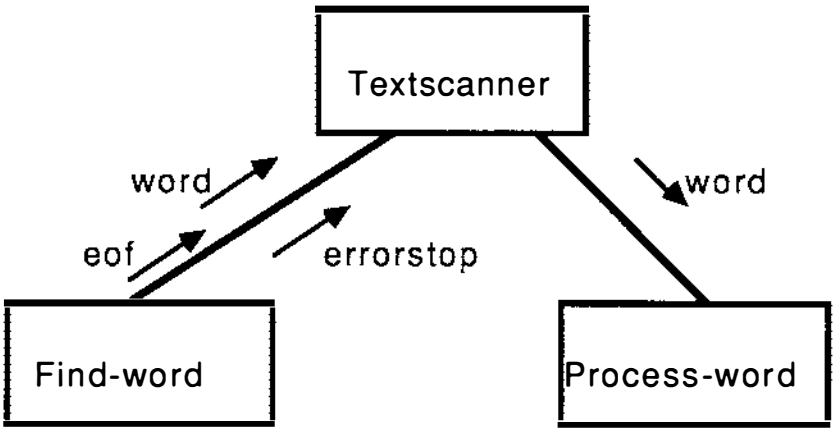


Figure 2. Partial structure chart.

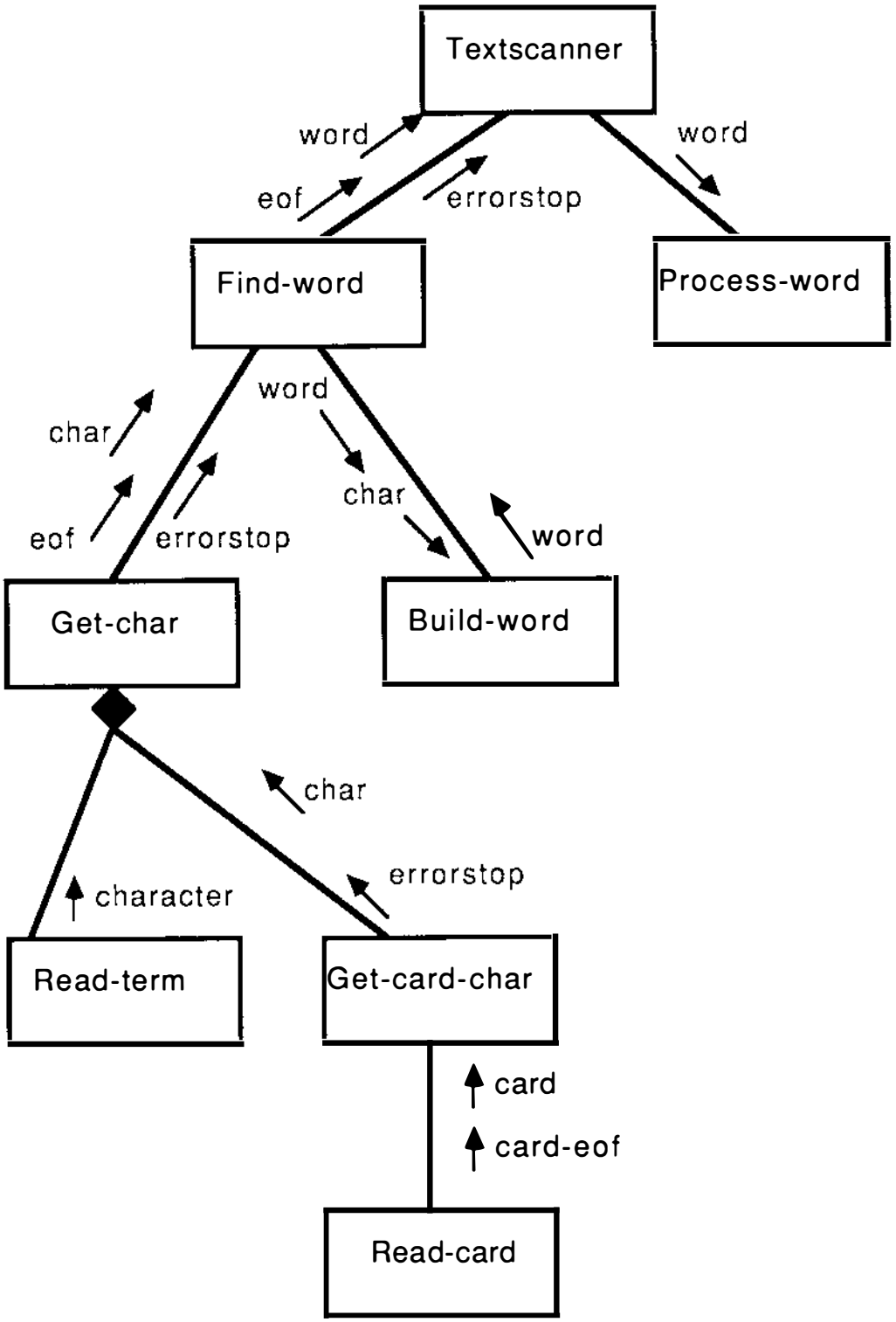


Figure 3. Completed structure chart.

Criteria that help determine reusability

Cohesion. One of the two most important criteria for determining the quality of a design is cohesion, which is a measure of how the activities within a module are related to each other.[6] Cohesion is also of great importance in evaluating potential reusability. There are seven levels of cohesion that are discussed in much of the literature on structured design and, in general, the higher the level of cohesion the more likely a module will be reusable. The cohesion categories on a best-to-worst scale are briefly described below.

A *functionally cohesive* module contains elements that all contribute to the execution of one and only one problem-related task. A module that performs two or more functions, each acting upon the same data in such a way that the ouput of one function is input to the next function is said to have *sequential cohesion.* A module that performs multiple functions, each of which acts on the same data but without the input/ouput relationship of a sequentially cohesive module is said to have *communicational cohesion.* A module whose elements are involved in different and possibly unrelated activities, in which control flows from each activity to the next is said to have *procedural cohesion.* A module that performs multiple functions which are related in time, such as an intitialization routine, is said to have *temporal cohesion.* A module that performs multiple functions that have some common attribute between them, in which some subset of these functions is explicitily selected at each invocation of the module, is said to have *logical cohesion.* Finally, a module that performs multiple functions that have no meaningful relationship with each other is said to have *coincidental cohesion.*

The top three categories, functional, sequential, and communicational, make for the the most reusable and easily maintainable modules because, as the definitions indicate, the activities of the module are all acting on the same data. The middle categories, procedural and temporal, are less likely to be reusable, and the bottom two categories, logical and coincidental, are highly unlikely to be reusable.

While cohesion is a good starting point in examining the potential reusability of a module, other factors must be considered in conjunction with it. In the case of the modules in figure 3, all of them have functional cohesion. However, those that have no subordinates and that perform a generic function likely to be needed in other environments are the most promising candidates to be salvaged as reusable modules. Applying this criteria, Build-word is more likely to be reused than Get-char. From the structure chart we can see at a glance that, considering only this criteria, the most probable candidates for reusability are Read-card, Read-term, Build-word, and Process-word.

Moreover, the concept of cohesion must be looked at at all levels of software hierarchy. In other words, we are not just concerned with individual modules but also with blocks of modules that form reusable subsystems. Thus, in figure 3 we would look for blocks of modules that, as a group, have a high level of cohesion. For example, one can imagine a use in some other application context for the block of modules that include Find-word and its subordinates. Notice that the structure chart itself indicates that all of the modules in this subsystem are involved in the process of getting characters from the input devices and building words from those characters. If there were other modules, such as error message routines, that were included in the subsystem, the structure chart would make those evident and the software designer could decide if they were appropriate in the new context or if revisions needed to be made.

To extend the example, if a subsystem were needed in another context that created words out of characters from only one input device, the structure chart provides a view of what revisions likely would be needed. For example, if the only input device is to be a card reader, then the likely revisions include the omission of Read-term and some modifications to the code in Get-char. Moreover, a careful examination of the code in Get-char and the interface data between it and Find-word, would determine, in large part, if changes needed to be made to the code in Find-word and Build-word and the extent of those changes.

Coupling. The second of the two most important design criteria, which is

also very useful to determining the extent of module reusability, is coupling complexity. The concept of coupling concerns the data and control items that are passed between modules, and the data that is shared between modules by way of mutual access to a common data area.[7] In other words, coupling is the degree of interdependence between two modules. The general principles are the following: The less coupling the better; simple data items (such as scalar variables) on an interface are better than data structures (such as records); processable data items on an interface are better than control items (such as function codes, switches, or flags); and local variables passed as parameters are better than shared access to global data areas. These guidelines are meant to be applied to evaluate the merits of a software architectural design but, in general, these same guidelines also apply to the reusability of software modules and subsystems.

The degree and type of coupling associated with a particular module greately affects the reusability of that module. Generally, a module that has complicated interfaces is less likely to fit nicely into another environment than a module with simple interfaces, and the same is true for module blocks, or subsystems. For example, on figure 3 Get-char has much more complicated interfaces than does Process-word; therefore, we would expect Process-word to be a more likely candidate for reusabiltiy.

Data dependencies. Another factor, closely related to coupling, that plays a role in analyzing reusabiltiy is data dependencies. If a candidate module requires data from a particular file that is not local to the module, or requires a specific type of processing to be done prior to using certain data items (such as the editing of input data), this will impact upon the potential reusabiltiy of the module. Since files are not explicitly shown, the structure chart gives only a limited amount of information about stored data dependencies. However, the source of the interface data in connected modules is relatively easy to find through code examination and provides the necessary information.

State dependencies. The structure chart can also help to locate state dependencies. The diamond on Get-char indicates that for each execution of its code it calls either Read-term or Get-card-char. The decision to

call one or the other is based on which input device is currently active. Thus Get-char must always be sensitive to the state of the system. Individual state dependent modules, in general, are poor candidates for reusability because they are highly context-dependent.

It must be stressed that, in using the criteria discussed in this section to evaluate potential reusability, the difference between examining individual modules and entire subsystems must be kept clear. A module that is a poor candidate for reusability because of either low cohesion, high coupling, or data or state dependencies, might be embedded within a highly reusable subsystem. As an example, it was pointed out earlier that Get-char is a poor candidate for reusability by itself based on coupling, state dependencies, and its contextual location in the structure chart. However, the subsystem headed by Find-word is a good reusability candidate, and it includes Get-char.

Case study

In 1983-84, the author was involved in a project where the techniques discused in this paper were put to work. A "Fortune 500" firm was undergoing a dramatic change in business policy which necessitated an extensive evaluation of their materials requirements planning system. The system was comprised of about 600 programs, ranging in size from very small (five modules totaling less than 200 lines of code) to very large (hundreds of program modules). The company wanted to determine to what extent they could salvage the existing software, the feasibility of making modifications to it, and the extent to which a total rewrite was required. The company decided to perform design documentation retrofitting on about 30 of the programs. A small software consulting firm, with which I was associated at the time, was contracted to do the work.

The system had the classic attributes of large systems that evolve over time: there was virtually no documentation; much of the software had been tinkered with extensively; and no one, not even the original designer (who was still around), actually understood the system and its components. The coupling between modules was extremely complex for the most part, and the names used for modules and data items were non-descriptive. Prior to

the project the client's software analysts considered the system to be "unmaintainable." In short, it was the worst of all worlds: they couldn't modify it because they didn't know what they had, but the new business requirements made it obsolete and in dire need of modification. Moreover, they were under intense time pressure, with upper management pushing them to salvage as much as they could as quickly as possible.

The hope and expectation of the company was that once they knew what they had on their hands they could make intelligent decisions as to where to go from there. Therefore, their main objective was to gain visibility. They hoped this, in turn, would enable them to retrieve whatever code they could, and modify it as needed, while rewriting as little as possible.

Most of the actual retrofitting of the design documentation was carried out by college students employed on a part-time basis, with senior people doing the analysis and refinement of the resultant product. The creation of the structure chart itself was relatively easy to do, and after just a brief tenure on the job it became quite routine for the students. The interfaces, however, were quite another matter. Because the language was COBOL, there really were no argument lists passed between modules: all data was global and contained in a COBOL Data Division. However, the customer wanted to know the "virtual parameters" for *performs* and *calls*. Thus the project team had to figure out the common references to data between modules and show this on the interfaces on the structure chart. Because this got so complicated, an interface table[8] was created to supplement the structure chart. An interface table is simply a two-column indexed table showing the content of the interfaces on the structure chart. Figuring out the "virtual parameters" for *performs* and *calls* proved to be quite time-consuming and tedious, and involved analyzing the code in much more detail than would ordinarily be needed. However, even with this particular difficulty the job progressed at a faster pace than expected. In fact, some project members estimated that they were able to retrofit at the rate of about 50 lines of code per person-hour (unfortunately, no hard data was kept on retrofitting rates).

Initially, many of the optional notations[9] that can be incorporated into a structure chart were included. For example, there is a special symbol to

represent iterative calls from within a loop. As the project progressed, however, it was decided to eliminate most of them. The litmus test in each particular case was whether or not the notation added clarity to the diagram sufficient to warrant the additional clutter. The symbol to represent conditional calls, as shown on Get-char in figure 3, was used sparingly, and only when the calling module was clearly a transaction center.[10]

As mentioned above, naming was a big problem. Renaming was done as best it could be, in an attempt to add clarity to the structure chart. However, the effort was hampered by the fact that the key company people were not available to give us feedback in this task. Consequently, the names of modules and data items were not as good as they could have been, even though much time was spent on this task.

In addition to the structure chart and interface table, a high level PDL (program design language) specification was generated for each program. It was felt that this was needed to give the analysts and designers the requisite view of the workings of the program, especially in light of the unreadability and total lack of documentation of the code.

The overall evaluation of this project by the client company was that it was very successful, especially because the problem of not understanding the system owing to its complexity was conquered by the graphical view. The original designer was particularly awed when he saw the structure charts for the programs. He was described by some as being like a "kid in the candy store" - all of a sudden he had all of the goodies of his system laid out before him, and he could see how they fit together.

Making the components of the system so highly visible enabled top management to make decisions with a relatively high degree of confidence about how to proceed. Thus they decided that they would live with some of the programs, change and modify some, and do a total rewrite on others. They also decided to perform the same technique on many of the programs not included in the initial project, using their own staff.

Some suggestions from the project team

The team that performed the retrofitting on this project felt that some of the tasks requested by the client proved to be of little value for retrieving reusable code, especially given that the source code was COBOL. In particular, there was too much initial concern over the specifics of the interface parameters. The team felt that the high visibility afforded by the structure chart (including basic interface information) and PDL specification, provided the essence of the architectural structure and functional relationships of the system upon which most major decisions regarding reusability could have been made. In certain specific cases, of course, more detail might have been required, and could have been added later. This leaner approach to the initial task would have simplified the project considerably since so much time was spent in generating the interface table.

There was also a feeling that the method used to retrofit the documentation would have been more effective had it been formalized. Leon Young, now Principal Engineer at Westinghouse Electric Corporation, was directly involved in this project from start to finish, and suggests an approach to formalizing the technique using three distinct stages. The first stage is to identify the calling structure and generate the structure chart from the code, using the module names exactly as they appear in the code. As we have seen, this can be done very quickly by junior people. The second stage is to come up with appropriate module names that represent the substance of their functionality, and to identify, and rename if necessary, the key data flows on the interfaces. The third stage is to write a PDL description for the processing done in each of the modules to give the requisite detailed view of the functional relationships.

Another suggestion from the project team concerns the nature of the personnel involved in generating the structure chart from existing code (the first stage mentioned in the paragraph above). It is important to realize that, once the method is learned, it becomes very routine and somewhat boring. The college students who worked on this project, all of whom were computer science majors, were initially very excited about the job especially when they realized the potential benefit of what they were

doing. However, after four or five days on the project it became very routine and tedious and lost its challenge. In fact, with minimal training and oversight, non-computer people can perform the task satisfactorily. Computer professionals, of course, are needed for the analysis functions and to write the PDL.

The final suggestion is to automate the process. Our college students worked with paper, pencil, template, and eraser to perform the retrofitting. It would have been an enormous benefit to have the entire process on-line with appropriate editing capabilities. The recent introduction of computer aided software engineering tools makes this a feasible option now.

Summary

Acquiring a coherent, high-level view of a system can be greatly facilitated by retrofitting design documentation from existing code. Evaluating the resultant design structure chart according to the quality criteria of the structured design method, especially cohesion and coupling, can be of great help in determining the potential reusability of code.

The project cited in the case study provided some valuable lessons for organizations who are in the position of modifying and retrieving code in poorly documented systems. The key to the success of this project was the design visibility that it provided. Retrofitting the design documentation from code proved to be a very cheap and fast way to obtain an extremely valuable perspective on the interrelationships of system components - the method delivered a lot of "bang-for-the-buck."

Organizations whose software people have a rudimentary knowledge of structured design techniques can employ this approach, and should find it a useful tool to assist in maintaining older, poorly documented systems.

Annotated references

1. J. Martin and C. McClure, *Software Maintenance: The Problem and Its Solutions*, Prentice-Hall, 1983, p.7.

2. While the idea of reusing parts of existing programs has always been an attractive one, the tools for doing it are just beginning to emerge. The July, 1987, issue of *IEEE Software* is dedicated to some of these emerging tools.
3. E. Yourdon and L. Constantine, *Structured Design*, Yourdon Press, 1975.
4. M. Page-Jones, *The Practical Guide to Structured Systems Design,* Yourdon Press, 1980.
5. G. Myers, *Composite/Structured Design,* Van Nostrand Reinhold, 1978.
6. See Page-Jones, op. cit. pp. 117-136, for an excellent discussion of the concept of cohesion.
7. See myers, op. cit. pp. 41-56, for a concise discussion of coupling and the trade-offs that must be considered when using coupling to evaluate design quality.
8. See Myers, op. cit. pp. 13-17, for a brief discussion of interface tables.
9. Appendix C of Page-Jones, op. cit., contains a summary of optional structure chart symbols.
10. A transaction center typically has a number of subordinate modules that perform specialized processing on different types of data transactions. The transaction center, therefore, analyzes its own input data and decides which of its subordinates should be called to handle the job. For an interesting discussion of the way a transaction center works in a real-world example see R. Pressman, *Software Engineering: A Practitioner's Approach,* McGraw-Hill, 1982, pp. 192-200.

Pacific Bell's UNIX/C Library Effort: An Example of a Corporate Reusability Library Implementation

Richard I Anderson
Rebecca Green
Pacific Bell

Abstract

Successful reusability library efforts require certain key components. This paper reviews some of these components and describes how they can be achieved by reviewing the implementation of a UNIX/C reusability library at Pacific Bell. The influence of UNIX on reusability is briefly reviewed, and concerns are expressed about means of measuring reusability library value.

Biographical Sketches

Richard received computer science degrees from Iowa State University and the University of Illinois, and he has partly completed a Ph.D. in educational psychology. Software design and development, university teaching, reading comprehension research, automated office user support, and computer-based instruction/examination research and development are among the activities that have comprised his career. Richard is now involved in human factors engineering and reusability library work at Pacific Bell.

Rebeccca was involved in information systems and reusability library work at Pacific Bell until returning recently to the academic world to pursue a Ph.D. in library science.

Introduction

As more and more companies establish or consider establishing software reusability librarys, a need exists to take a global look at the task of library implementation. Some components of library implementation, such as software cataloguing schemes and software tools for library access and use, have been the subject of detailed study (see, for example, Prieto-Diaz & Freeman, 1987, and Burton et al., 1987, respectively). However, few have attempted consideration of the bigger picture.

A global view of library implementation encompasses institutional, political, social, and instructional issues in addition to technical considerations. Funding must be adequate; users must be motivated; access to a library's contents must be easy; ... Many, varied factors must be dealt with to enable a library's success.

This paper describes the nature of mechanisms implemented at Pacific Bell in order maximize the chances of success of a reusability library in the UNIX environment. Means of attaining user/manager buy-in and awareness, substantive and quality inventory, and sufficient support for library users are among the mechanisms discussed. Weaknesses in Pacific Bell's initial mechanism implementations are identified, with concern expressed about the task of measuring library value. The impact of the nature of the UNIX environment on the potential success of a library effort is briefly reviewed.

UNIX & Reusability

Because of the philosophy underlying the basic UNIX system, reusability and UNIX go very much hand in hand. UNIX commands aren't "true" UNIX commands unless they are small, independent, well-defined modules meeting standard criteria for input and output. When combined with UNIX shell facilities such as pipes and input/output redirection (see Snyder & Mashey, 1981), UNIX command output can be easily reused and easily passed to other commands. In short, UNIX is an environment of building blocks that can be easily interconnected in an infinite number of ways.

> ...the [UNIX] approach is to treat an existing program as a black box with known but in some way inadequate properties. Rather than rewrite it, one wraps it up in a (much smaller) program that provides more suitable properties, or that permits the job to be factored into easy pieces. (Kernighan, 1984)

Housed in such an environment, software development often follows the UNIX philosophy. Programmers create tools and other segments of code that almost naturally conform to basic UNIX conventions. As such, these products are often highly reusable not only by those who developed them, but also by many, many others.

Unfortunately, many programmers prefer to tinker and to create - to make changes to programs or to write new ones - even when existing UNIX-based programs can be combined to achieve the goal (Turner, 1985). This preference to "do it oneself" competes with motivation to access and to use the products of others. Furthermore, pride in one's own creations can combine with feelings of competitiveness and product ownership to suppress willingness to share one's products with others. Such preferences and feelings are some of the obstacles to a library effort's success.

Critical Library Effort Components

A comprehensive analysis of corporate, library implementation would benefit many future implementations, because a successful library effort has many complex and interrelated ingredients. No such analysis is attempted here, but some of the necessities of a successful library are delineated.

Buy-in

Many people must believe that they are receiving or will receive high value for the time, money, and effort that they are asked to invest. For example, management must understand and see the value of a reusability library in order to initiate or continue a library's funding, and (potential) users must see and accept value to them as professionals or income earners before they become (actively) involved.

Awareness

Awareness of a reusability library's existence, its contents, and its value is a prerequisite of buy-in and use, and it is a component that requires continual renewal.

Inventory

Various sources of library inventory exist. For the UNIX/C environment, public domain sources are many (see Ward, 1986), and reusable modules are available for purchase (Productivity Products International is one company that markets reusable code). But, the most valuable sources

often lie within the corporation itself.

Quantity is important, as is the nature of the inventory. Holes reduce a library's value, as might the limiting of contents to code only.

Means of Access

A means of accessing a library's inventory is an obvious necessity, and simple storing and sifting through a pile of inventory will be inadequate most often. Sound, easy means of storing, retrieving, identifying, finding, and understanding inventory is likely to be essential.

User Support

Assistance in the use of the library and its contents (and maybe with some things not directly associated with the library) should be provided by a staff that is both accessible and responsive. For inventory about which the staff knows little, information about external support sources should be provided.

Quality Assurance

All library contents should meet basic standards, and efforts to improve inventory quality should be ongoing. A user's removal of one faulty routine from the library could result in severe damage to the library's reputation and dramatic library use reduction.

Input from the user community is a good source of information about things in need of improvement.

Value Measurement

Periodic assessment of the value of a reusability library to individuals, work groups, and the corporation is important. Retention of buy-in without a measure of value will be problematic.

Pacific Bell's UNIX/C Library Implementation

Awareness of the benefits of reuse and a desire for greater unity and sharing among fragmented development groups led to the creation of a Pacific Bell UNIX/C Library in 1986. Two employees proposed the idea, and management listened and bought in. Budgeting followed, and plans were made to assemble a staff that would design, construct, and operate a reuse library.

Before staffing the library, an attempt was made to garner the involvement of as many segments of the UNIX/C development community as possible. Community segments were identified, and each was invited to identify a representative to serve on a committee that would exist long-term to guide the activities of the library staff. Each representative's role was to voice input to the library effort to ensure that the library would benefit their community segment.

Via this governing committee, word was disseminated about the library's pending creation and need for inventory. Collection of future library "books" from multiple sources began immediately, and a design/plan for the library effort began to take shape. Within six months, the library was open for business with over 200 products on the shelves, 150 employees with "library cards" poised for use, and a staff ready to support the library effort with consultation and educational services.

The Library's Inventory

From the beginning, the library governing committee took a broad view of what should go into the library. Three major categories of material were to be represented: software, technical reference material, and training information. Within the software category, the need was seen to collect reusable code, application packages, productivity tools, database managers, and training packages. Technical reference material includes descriptions of applications at Pacific Bell, hardware and software evaluations, white papers, and reports of conferences or other meetings. Information about courses offered in-house and also by outside vendors comprises the training category.

Amidst the variety, the major thrust of the library has been software. But even within this category, there is great variety. Full-blown applications stand alongside subroutine libraries, but stand-alone programs make up the biggest proportion.

Two categories of software are proving to be the most popular - productivity tools and application packages. The productivity tools available include tools both for system administrators and for programmers: kernel monitoring tools, networking tools, special text editors, special shells, product administration and software release tools, etc. Included in this category are utilities common to many versions of UNIX, but which, for a variety of reasons, may be absent from one of the many machines running UNIX within the company. Application packages include a database

manager interface, a screen generator, and a library of functions for terminal control.

When it comes to code that may be used within an application, the library's holdings are not yet very generic. Code that has been submitted to the library from applications has tended to be application-specific. Some (for example, Standish, 1984) have claimed that this misses the reusability mark significantly, for indeed, users who borrow from the library can expect in many cases to spend some degree of time and effort in re-shaping the code for their purposes. However, the benefits of this reuse have been shown to be sizable (Biggerstaff, 1987; Jones, 1987). Reaping greater benefits via "commissioning" generic code to meet needs that arise over and over is under consideration.

Cataloguing Scheme and the User Interface

Enabling users to search the library is, of course, a primary function of the library user interface. In the current library application, all library products have been classified using a list of 17 major categories (e.g. graphics, project management, performance analysis, etc.). In addition, all library products have been assigned some number of keywords. When users come to the library, they specify which category they wish to search (or they may search all categories simultaneously). If they care to make their search less general, keywords may also be specified.

(Users have had some difficulty with this system, due to the ambiguity of major categories and the arbitrary assignment of keywords. Because of these weaknesses, the existing cataloguing mechanism is likely to be replaced by an integrated index language which will better control the assignment of terms to library products, hence, easing library searches.)

Product files fill subdirectories of parent directories representing each category, and a master copy of each product is maintained under UNIX's Source Code Control System (SCCS), allowing the library staff to recreate any past version of the product. Periodic releases are performed from this archive to the library shelves, which are accessible to the user community through the library application.

To access the library, users login directly to a menu-driven application resident on an AT&T 3B20. From the main menu, a user can choose to search the library, request shipment of products (the library will ship files via the UNIX to UNIX copy facility (UUCP), or will make them available on floppy

disk, cartridge tape, or paper), enter a comment in the suggestion box, update their registration data, make arrangements to submit a product to the library, or get help.

Users have been required to log on to the machine in order to request an individual login. By using a particular and well-publicized login, future users access a registration screen on which they note certain information regarding themselves and their job. Shortly afterwards, these people are contacted by a library staff member and given their login and password.

In most library efforts to date, the library is not an integral part of the users' design and programming environment (Shriver, 1987). Unfortunately, Pacific Bell's UNIX/C library is no exception. The remote file sharing and shared library features of AT&T's most recent release of UNIX (V.3; Shomo, 1987) will enable software changes that produce desirable integration.

Submission Processing

As indicated, the process of submitting a product to the library is initiated by the user from within the library application. Having chosen the appropriate menu option, the user is presented with a simple form, asking for the proposed method of submission (e.g., machine-to-machine data transfer or tape) and the approximate size of the product. A library staff member contacts the prospective contributor to make arrangements for the submission.

After the product is received within the library machine, it is assigned to a major category and cross-referenced as appropriate to other categories. It is then assigned to a technical writer for review. For software, this writer insures that the code compiles and behaves reasonably and adds a documentation file, if not already present, which describes the product, and lists its author, the categories to which it belongs, the keywords assigned, any hardware or software dependencies, its size, and the command that will compile and install the product, among other items. Later, as demand for a product warrants, the product is revisited by a quality assurance staff member. For software, this person conducts further tests on other machines and makes sure that the product has its corresponding manual page(s); any obvious limitations or bugs are noted. Products that have undergone this more careful scrutiny are said to have been "certified."

Incentives

Important internal incentives exist for contributing to and using the library. Growth that accompanies the exploration of, the study of, and experimentation with library contents is often intrinsically valued, as is the pride of contributing positively to the work of others via library submissions. But external incentives can supplement intrinsic motivation significantly.

A seemingly trivial yet important component of Pacific Bell's added collection of incentives capitalizes on a basic need of humans to be recognized for their work. Each product contributed to the library includes clear identification of the author [and the contributor(s), if different from the author(s)]. Subsequent recognition by peers motivates additions to the library inventory.

Submissions to the library are motivated in additional ways. For submissions, employees receive small but useful tokens of appreciation such as pocket screwdrivers, coffee mugs, pens, and clipboards with attached calculator. Tangible gifts of greater value are awarded to those who contribute products that produce repeated, significant savings to development projects; these "royalty" awards include cash and trips to professional conferences.

Use of library products is encouraged as well. Users accumulate points for every product removed from the library and used in or for producing an application; top point accumulators are given cash or meals periodically.

(Because the task of tracking product use is so difficult, "royalty" and use awards may be supplemented or replaced by a lottery. For every product removed or submitted, a user would receive one or more chances in a regularly-held drawing for gifts.)

Consultation

Staff expertise is made available to assist users in need. Problems for which help is provided are not always limited to those involving the library or a library product. Loans of staff members for special projects can be arranged, since the expertise of library staff members is a valuable, reusable commodity.

Prompt response to immediate needs is ensured via a hotline phone number. Advertised repeatedly, the hotline is a speedy means of providing user support.

Education/Communication Services

The library governing committee realized that sharing by software developers via the library could be fostered by improving the access of the software developers to each other. Hence, means of exchange of ideas and information among Pacific Bell's UNIX/C community members were created within larger educational frameworks.

One means initiated early was a monthly newsletter. News about the library and UNIX-related corporate and industry developments appear on pages with educational features. Most of the articles are solicited from members of the user community.

Periodic symposia are sponsored also. Industry and user community leaders are provided a structured forum via which to communicate with and educate others. A full day of general and concurrent sessions on a wide variety of topics includes an exhibit room filled with vendors and Pacific Bell personnel eager to demonstrate and advertise their UNIX/C products and services.

Summary & Discussion

Pacific Bell's UNIX/C library implementation reveals examples of how critical components of a library effort can be achieved. A summary appears below.

Buy-in:
- Governing committee of user community representatives
- Periodic reports to management
- Incentive program

Awareness:
- Governing committee reports to the user community
- Monthly newsletters
- Staff member meetings with user community work groups
- Announcements via electronic communications network
- Brochures
- Symposia

Inventory:
- Software (source code, productivity tools, ...)
- Technical references (software/hardware evaluations, ...)
- Training information (internal & external offerings)
- Approximately 400 distinct products
- Archives of earlier versions
- Submissions from user community and staff members

Means of Access:
- Menu-driven application on superminicomputer
- Individual user logins
- Contents associated with categories, keywords, product names
- Copies via uucp or on floppy, cartridge tape, paper

User Support:
- Staff consulting/troubleshooting
- Hotline
- Loans of staff to special projects

Quality Assurance:
- Documentation and certification by staff
- Input from governing committee
- Electronic suggestion box
- Questionnaires to user community
- User focus group meetings
- Research and evaluation by staff

Missing are means of measuring library value.

In theory, the value of a library can be measured by comparing the value of software systems developed, modified, and installed with a library to the value of the software without a library; a similar comparison of the costs of development, maintenance, and installation with and without the library would also be required. However, no adequate means of measuring the quality or value of software have been identified, and the chore of determining what software would have been produced and at what cost in a hypothetical setting or a test environment is difficult and costly.

A measure of value in terms of person-hours saved by using existing code is a possibility, but without methodical tracking of time spent producing code, such a measure would be very subjective. Furthermore, the measure does not address value gained via such things as using library productivity tools, studying library technical references, and browsing through well-written, library code. Nor does this measure account for costs associated with time spent accessing the library and its products, time spent figuring out how to integrate code into an application, etc.

A measure of the value of the library in terms of POTENTIAL savings is another possibility. But is this enough?

Sound means of measuring value remain to be identified, let alone implemented.

Notes

™ UNIX is a trademark of AT&T Bell Laboratories

For additional information, contact the library staff by phoning the Pacific Bell UNIX/C Library Hotline: 415-867-5270.

References

Biggerstaff, T. (1987, June) *Reusability*. Presentation at the Pacific Bell UNIX Symposium, San Ramon, CA.

Burton, B. A., Aragon, R. W., Bailey, S. A., Koehler, K. D., & Mayes, L. A. (1987, July) The reusable software library. *IEEE Software*, 25-33.

Jones, C. (1987, January) *Program quality and programmer productivity: A survey of the state of the art*. Presentation at a Pacific Bell Information Systems Forum, San Ramon, CA.

Kernighan, B. W. (1984) The Unix system and software reusability. *IEEE Transactions on Software Engineering, SE-10*, 513-518.

Prieto-Diaz, R. & Freeman, P. (1987, January) Classifying software for reusability. *IEEE Software*, 6-16.

Shomo, R. (1987, June) *AT&T UNIX System V.3.1*. Presentation at the Pacific Bell UNIX Symposium, San Ramon, CA.

Shriver, B. D. (1987, January) Reuse revisited. *IEEE Software*, 5.

Snyder, G. A. & Mashey, J. R. (1981, January) *UNIX shell tutorial*. Murray Hill, NJ: Bell Laboratories.

Standish, T. A. (1984) An essay on software reuse. *IEEE Transactions on Software Engineering, SE-10*, 494-497.

Turner, D. M. (1985) *UNIX*. San Francisco, CA: Pacific Bell Corporate Television.

Ward, R. (1986, October) *Sources and uses of public domain C*. Paper presented at the C. L. Publications, Inc. C Seminar/Workshop., San Francisco, CA.

FULCRUM: A REUSABLE CODE LIBRARY TOOLSET

Mary Hsia-Coron
Jennifer Marden
Eeman Wong

Hewlett-Packard
Data Systems Division

ABSTRACT

Fulcrum is a prototype toolset for the management of reusable component libraries. It was created for engineers in Hewlett-Packard's Data Systems Division and runs under HP-UX, Hewlett-Packard's version of UNIX[1]. In the course of designing and implementing Fulcrum, we came across a number of issues which we feel need to be addressed in order to optimize the effectiveness of software component libraries in general.

We begin this paper with a general description of the Fulcrum toolset and then discuss possible extensions to Fulcrum. Our goal in this paper is to identify the essential ingredients and preconditions for an optimal software component library. This paper draws from our experience with Fulcrum and from the literature on software reuse.

FULCRUM PROJECT OBJECTIVES

We were asked by our users to construct a set of library management tools that would satisfy the following requirements:

- Archive software components (source code and related documentation, tests, and design documents) in a library on the HP9000 series 840
- Browse the library, using a lexical search method, for software components which match a reuser's needs
- Retrieve a copy of a software component from the library for reuse or updating
- Allow simple and flexible submission of components to the library
- Need minimal data entry during the submission process
- Have minimal library admission requirements

1. UNIX is a trademark of AT&T Bell Labs

Figure 1

```
/* REUSABLE CODE PACKAGE HEADER

&NAME: gpiotalker &
&REVISION: 1.1 &
&AUTHOR: jennifer &
&MAINTAINERS: jennifer ray &
&ONELINE:  Passes messages to and from a gpio card &
&STATUS: complete &
&LANG: c &
&OS: hpux &
&MACHINE: s840 &
&DEPENDENCIES: &
&DOC: < /mnt/users/jennifer/doc/gpiotalker.1 > &
&HIST: used in ddb product. &
&DEFECTS: &
&SOURCE: /mnt/users/jennifer/src/gpiotalker.c  @cardio <2.1>  &
&TEST: /mnt/users/jennifer/test/gpiotalker &
&DESIGN: /mnt/users/jennifer/des/gpiotalker &
*/
```

Figure 2

```
/* REUSABLE CODE COMPILATION UNIT HEADER

&AUTHOR: jennifer &
&LANG: c &
&OS: hpux &
&MACHINE: s840 &
&DEPENDENCIES: &
&TEST: &
&DESIGN: /mnt/users/jennifer/des/cardio &
*/
```

DEFINITION OF A FULCRUM LIBRARY "COMPONENT"

In the Fulcrum Library a component, i.e. the unit of reusable software information, is known as a "package". A package consists of a set of files, each of which may be any one of the following types:

- Source code
- Design -- External Reference Specification (ERS), Data Flow Diagrams, Structure Charts, etc.
- Documentation -- any documentation intended for the reuser of the package
- Test -- tests written for the package

THE FULCRUM TOOLSET

The following section gives an overview of the utilities provided by Fulcrum.

Submit

The "submit" utility allows the user to check a software package into the library. "Submit" needs to know what files make up the package and, if a package is accepted, what package information to add to the library's database. (Fulcrum uses a very simple relational database called RDB, purchased from Unipress.) "Submit" obtains the information it needs from headers which the submitter has included in each of the package's source code files. There are two types of headers: the Package header and the Compilation Unit header. The Package header is included in one and only one of the package's source code files. It contains information about the overall package. (See Figure 1 for an example of a Package header.) A Compilation Unit header is included in each one of the package's source code files. It contains information pertaining only to the source code file in which it resides. (See Figure 2 for an example). "Submit" calculates defaults for all optional header fields if they are left empty by the submitter.

"Submit" behaves like a standard UNIX command. It accepts file name(s) as input parameters. "Submit" checks every submitted file for a package header and uses the information in these headers to locate copies of all the files it needs to make up each package.

"Submit" takes its input from the Package and Compilation Unit headers, which may be outdated or corrupt. This can lead to packages being submitted incorrectly. "Submit" mails the user a receipt containing any error and warning messages resulting from the submission process, information about which package(s) were accepted by the library, and a list of the files in the accepted package(s). The user can use the Fulcrum utility "unsubmit" to delete a submitted package if it was submitted incorrectly.

"Submit" uses many UNIX utilities, most notably "awk", "grep", and "sed". RCS is used for version control.

Browse

Fulcrum's "browse" utility is based on a simple lexical search algorithm. The user specifies strings and "browse" searches a database of one line package descriptions to find occurrences of the strings. An option is available to make "browse" search all package documentation for user-specified strings. The user may also search by package name and by the login name of the creator of the package. Multiple search criteria may be given, and the default search method is a logical OR search on all the parameters supplied by the user. A logical AND search method may be specified with the appropriate option. "Browse" assumes that the user wants a terse output, that is, only the names and one line descriptions of the packages satisfying the browse criteria. If the verbose option is specified, "browse" displays all package documentation.

The following is an example of the command line that would be used to do a lexical search on all package documentation for the string "device driver", to search all one line descriptions for the string "peripheral", and to search the database for all packages created by the person with the login name "taylor":

```
browse -v -d "device driver" -l peripheral -c taylor
```

Full documentation is displayed for packages which satisfy ANY of the above three browse criteria.

"Browse" uses the Unipress database package RDB and "grep".

List

The "list" utility allows the user to print out parts of a package to the terminal. This is useful for examining packages found by "browse" before deciding whether to check them out of the library.

Checkout

"Checkout" allows anyone to obtain a copy of a software package from the library for reuse. It also provides an option which enables the submitter or authorized maintainer of a software package to check the package out to fix or enhance it. The user may check out all, or a subset, of the files in a package. When checking out a package, the user may specify a particular revision of a package, or let checkout default to the latest revision.

"Checkout" also relies on RCS for version control.

Chatt and Frlog

"Chatt" and "frlog" are Fulcrum's versions of the RCS commands rcs and rlog. "Chatt" allows the user to "change attributes" of a packages, such as the list of users authorized to update the package, or to unlock a package which has been locked for updating. "Frlog" allows the user to examine the revision control information for a package, such as the list of authorized updaters, available revisions of the package, who has locked a package, etc.

EXTENSIONS TO THE FULCRUM TOOLSET

We feel that Fulcrum is probably adequate to meet the specific reuse needs of the engineers for whom it was created. However, both enhancements to the Fulcrum toolset, and development of other facilities such as coding guidelines and software classification schemes, etc., are needed to support more general reuse.

ENHANCEMENTS TO TOOLS

The section below describes the enhancements which we would like to see made to the Fulcrum toolset in order to make it a more complete software management toolset.

Submit

There are two improvements which should to be made to "submit". The first is adding the ability to screen or "grade" software by calculating its scores on a set of quantifiable "trust" or "quality" vectors. (See the section on Trust Vectors.)

The second enhancement is the provision of an interactive interface which would help the user fill in the headers, and check and confirm header information before actual submission takes place. This new interface would be straightforward to implement using HPToday, a Hewlett-Packard Fourth Generation Language. It might also be developed using the X windows system and X-Ray, Hewlett-Packard's X windows user interface library.

Browse

There are two directions we could take with "browse". We could replace it with a third party lexical search and text retrieval system such as BRS/Search. Alternatively, we could adopt a component classification scheme with a controlled keyword list supplemented by a thesaurus. (Refer to section on possible classification schemes below.) It might also be possible to combine these two approaches. We will wait to see what feedback our current "browse" utility receives before we make a decision.

Metrics Collection

We have not addressed the problem of how to measure the quality and productivity gains obtained by using components from the library. This is an important area, as the effort of creating and maintaining a library must be shown to produce a good return on investment.

Defect Tracking and Repair

In order to gain user confidence in library components some provision should be made for making and receiving defect reports, fixing the defective components in a timely and reliable manner, and letting anyone who is using the defective components -- or who is responsible for a product which uses the components -- know about the problem. This defect tracking system should also deal with the genealogy problem: checking and fixing if necessary the defective component's ancestors, descendants, and siblings. Additional defect tracking problems may occur when components are included in released products. When a defect encountered by a customer is determined to be in a component from the library, the information should be easily transferable to the library maintenance group.

At present, Fulcrum provides no defect tracking. "Checkout" records the login names of users who have checked out each package. This information needs to be tied into a defect tracking system so that users can be automatically notified if a package they have checked out is changed. An even more useful enhancement would be to have a database which cross references packages against the products they are used in, and notifies product support teams of fixes.

Fulcrum has no library maintenance group to support contributed components. It relies on the authors of components to maintain their submitted software package(s) on a voluntary basis. If the library components are ever widely used outside the group which developed them, some arrangement should be made to compensate the authors for support, or to provide a library maintenance group to take care of support for them.

Distribution Across a Network

Fulcrum does not provide facilities for automatic distribution of components across a network since all of its users are on one machine.

Distribution of components becomes a critical issue when component libraries are shared by groups at different sites. As various groups in HP begin to develop and share their component collections, it will be necessary to construct a library system that can manage the distribution of many collections. Reusable components should ideally be distributed electronically. It is possible at this time to devise fairly simple centralized distribution schemes using electronic mail. An example is the information server

"netlib", developed to distribute mathematical software to anyone who sends it an electronic request [Dongarra]. Another possible approach to the distribution problem is the use of a distributed database, such as Oracle or Ingres.

OTHER FACILITIES NEEDED

An adequate software library requires not only a set of tools to manage the collection of components, it also needs other facilities such as standards, guidelines, methods, and possibly a classification system.

Classification System

The implementation of a browser will vary according to the classification scheme selected for use. The following is a list of often cited software classification systems:

- Enumerative
- Faceted
- None (lexical search method used)
- KWIC (Key-Word-in-Context)

Enumerative System:

The enumerative system is a traditional approach which postulates a "universe of knowledge" and then "divides it up into successively narrower classes" that are arranged in a hierarchical structure [Prieto-Diaz]. Examples of enumerative systems are the Dewey Decimal system and the Library of Congress system for classifying documents. Most software classification schemes -- such as the ones used by Guide to Available Mathematical Software (GAMS), and International Mathematical and Statistical Library (IMSL) -- are enumerative.

Faceted System:

The faceted system differs from the enumerative system in that it does not rely on the breakdown of a "universe of knowledge". Instead, it takes a synthetic approach to classifying components by using a combination of keywords to represent the different characteristics of a reusable component. In the article "Classifying Software for Reusability", Ruben Prieto-Diaz uses an example of a partial faceted scheme (for classifying literature on animals) which illustrates the concept of a faceted system:

(Process facet)
  Physiology
    Respiration
    Reproduction
    Nutrition

(Animal's habitat facet)
  Water animals
  Land animals

(Zoologist's taxonomy facet)
  Invertebrate
    Insects
  Vertebrates
    Reptiles

Mammals
Bears
Grizzlies

For a software classification system, Prieto-Diaz recommends the use of the following facets:

- Function -- specific primitive action performed by software component
- Object -- object manipulated by software component
- Medium -- agent, or the "locales where the action is executed" (e.g. data structures)
- System Type -- e.g. compiler, assembler, relational DB
- Functional area -- e.g. cost accounting, CAD, DB design, auditing
- Setting -- e.g. chemical plant, airport, library circulation, broadcast station

Prieto-Diaz claims that the faceted scheme is a more flexible and accurate method for classifying components in an ever changing, growing software collection because it allows new categories to be added to any facet as the need arises.

No Classification System (Lexical Search Method Used):

A number of software libraries do not use a classification system at all. They rely on a lexical search method. The Fulcrum library is an example of this type of library. There are both advantages and disadvantages to using a lexical search. The advantages include: 1) being able to search for components without being constrained by a keyword structure, 2) saving a good deal of time up front by not having to develop a classification system, and 3) not having to screen all submissions to ensure that they are classified appropriately. The disadvantages include possible degradation in performance of the browser as the number of components in the library grows (unless a powerful lexical search tool is found). A lexical search method also operates on the premise that documentation for components can serve two different functions: 1) accurately describe what the component does and how it should be used, and 2) provide a good selection of string descriptors which users of a lexical browser are likely to utilize during their search. It is unrealistic to assume that all contributors to a library will always be able to do both of the above equally well, or that they will devise descriptors which are consistent with what others have created.

A problem which faces Fulcrum is the future performance degradation of the "browse" utility as the number of software components in the library increases. A lexical browser is appropriate during the early days of construction of a submissions library (i.e. a library comprised solely of voluntarily submitted components) when it is unclear what components may be contributed. But as the component collection grows, and the library moves beyond the prototype stage, a classification system and a browser adapted to this system are helpful. Since faceted schemes are less restrictive than enumerated schemes, they are probably better to use. In the case of designed libraries where all the components are fabricated by an assigned group of engineers, rather than contributed by volunteers, the use of a classification scheme is appropriate from the outset since the components are typically well organized, reflecting the structure of the application domain.

KWIC Classification System:

The KWIC classification system is based upon an indexing strategy known as Key-Word-In-Context. The index generated using the KWIC method is often called a permuted index. This type of index is often placed in manuals to help readers locate a subject of interest. An example is the permuted index found in every UNIX reference manual. The UNIX permuted index is composed of one-line

descriptions from commands, system calls, and subroutines in the UNIX library. The quality of a permuted index depends very much on how well contributors have described their component. Thus, the KWIC classification scheme shares some of the disadvantages of the lexical search method. In general, permuted indexes are not very effective vehicles for locating components in a software library of any significant size.

Thesaurus of Keywords

Component libraries using keyword based classification systems (e.g. faceted, enumerative) usually benefit from the addition of a thesaurus. Besides searching the library for components that match keywords supplied by a user, the "browser" could check a thesaurus to obtain keyword synonyms and retrieve components that also match the synonyms. When aided by a thesaurus, the "browser" can become more effective at locating the components that users want.

A thesaurus helps to solve the problem of people using the same word to mean different things, or using different words to mean the same thing. However, a thesaurus relies on unconditional string substitution so it cannot judge the context in which words are used to determine if the words are really synonymous for a given situation. This can lead to the "browser" selecting components which do not fit the user's needs. So a thesaurus, while very helpful in many circumstances, is not the complete answer to an intelligent browsing system.

Standards for Documenting Components

Documentation standards are probably best developed by the users of component libraries; they should be kept simple at the outset and allowed to evolve over time. Standardization of documentation by committees should take place after separate groups are given a chance to test out their individual standards. Standards will probably vary somewhat for components created in different operating system environments and different languages (e.g. Object-oriented vs. procedural languages).

Guidelines for Designing and Coding Reusable Software

Guidelines for designing and coding reusable software are essential if we wish to promote reuse practices among software engineers. However, genuinely useful guidelines which can be easily put into practice in specific instances are difficult to create. While the concepts of highly modular, loosely coupled components are important, they are too general to substantially increase reusability of code and designs. Guidelines need to be at least language specific to have a significant impact. At best, guidelines should be application specific, combined with individual project standards and guidelines. Some detailed Ada specific reuse guidelines have already been developed. (See [Berard], [Braun], and [StDennis].) In addition to language-specific guidelines, a "language" of commonly used conventions, component interfaces, and functional decomposition patterns needs to evolve for each application so that components can easily be used together. Some of these guidelines and language elements can be derived now from the concepts found in [Page-Jones] and [DeMarco]. Others can only be formulated as we begin to reuse components and discover what characterizes components that are the easiest and/or most productive to reuse.

Reuse in the Software Lifecycle

Having a library of reusable components available will not lead to significant productivity gains until reuse is built into the design methodology and software lifecycle used to develop products. Below are some ideas for how to do this:

- INVESTIGATION PHASE.

Set reuse objectives. How much software should the product reuse? How will this reuse be measured? Where will the reusable software come from? An in-house library? Another in-house product, or a third party product?

- DESIGN PHASE.

  Decompose the product with reuse in mind. Identify common functionality for reuse of new modules within the product itself wherever possible. Work closely with the reusable code library staff to determine how best to decompose and design the product so that it takes maximum advantage of what is already in the library. Also identify modules in the product which are good candidates for the library and plan to spend extra effort on these modules in order to make them more easily reusable for later products.

- CODING PHASE.

  Review those modules identified as good candidates for the reuse library for reusability.

- TESTING PHASE.

  Reuse tests and test plans.

- MAINTENANCE PHASE.

  Have the library staff maintain and enhance all library components as needed.

Trust Vectors

In order to give users confidence in the quality and reusability of the components in the library, there should be either minimum standards for admission, or a set of scores on quantifiable quality or "trust" vectors available for each component so that the user can quickly judge how reusable and reliable a component is. This problem is very difficult for two reasons: 1) It is hard to quantify reliability and quality, and hard to even define reusability; and 2) Many common sense measures of reusability such as "use of mnemonic identifiers" or "completeness and readability of documentation" could be easily assessed by a person, but would be very difficult to recognize in an automated way. Quality and reliability might be measured by automated testing of components, but this is difficult to fully automate with the testing tools available now which require careful set-up and do not recover well from errors. There are metrics tools available which might also be used to measure quality vectors. The more sophisticated of these tools suffer from the same problems as testing tools.

Some possible measures of quality, reliability, and reusability are listed below. Some may be measured automatically; others may require human judgement.

- Quality and Reliability
  - Number of compilation warnings and errors, and lint output (for C programs only).
  - Size of tests provided (number of tests, length of tests, length of test documentation) / Number of Non-Comment Source Statements (NCSS).
  - Does the component pass the tests submitted with it? Does it pass tests devised by the library maintenance group?
  - Results of Branch Flow or Path Flow Analysis of tests used.
- Reusability
  - Design and Code
    - Number of functions and procedures / NCSS
      (A crude measure of modularity)

- Number of functions and procedures which fall into each coupling and cohesion category defined in [Page-Jones].
  (A better measure of modularity)
- Use of standard interfaces and conventions, or interfaces and conventions used frequently by other components in the library.

– Documentation
  - Size of documentation (length and number of documentation files, number of comment lines) / NCSS
  - (Number of functions and procedures with headers) / (total number of functions and procedures)
  - Use of mnemonic variable names
  - Completeness and readability of documentation

THE CONTENTS OF A SOFTWARE COMPONENT LIBRARY

Thus far we have examined only the key features of a library management system. The issue of what library management tools and facilities are needed is distinct from the issue of what components to put into the library. A component library with a sophisticated set of tools and facilities may be useless because of the low level of reusability and lack of organization of its reusable components. The converse is not necessarily true. A highly automated library archival and retrieval system does not appear to be essential to the success of smaller component libraries. Library tools and facilities become more important as the size of the library grows. Some Japanese companies (e.g. Toshiba) have achieved high rates of reuse for several years by simply providing programmers with hardcopy software parts catalogs describing available components [Podolsky]. We have also heard about a U.S. company which achieved a reuse rate ranging from 50-90% by employing a human librarian and a minimal set of library management tools [McGregor]. It is our contention that library management tools are very helpful reuse aids, but there are other requirements for a successful component library. These other requirements include:

- Personnel to support the components, and also the library classification system
- Management commitment to reuse (e.g. provide incentives)
- A set of well organized reusable components of high quality.

The first two items are organizational issues which will be touched upon only briefly in this paper, although we wish to point out their importance. In the following sections we will primarily examine the components that go into software libraries and try to determine how successful component collections are constructed.

Submission Libraries vs. Designed Libraries

Companies generally adopt one of three strategies when they wish to build up a collection of components for their software library. They may:

- Solicit submissions from in-house programmers
- Assign a team to develop the components
- Do some combination of both of the above

The first approach usually results in a collection of functionally diverse components of mixed quality and reusability unless there is a support group established to solicit and screen components, and to

rework them if necessary. Companies that do not provide the latter often find that they end up with a large set of software components that are not used very often because: 1) engineers do not have confidence in their reliability, 2) there is not a critical mass of components for the engineer's application domain, 3) related components are not organized for easy combination since they were created by different individuals, or 4) individual components have not been designed to be reused easily.

The second approach, i.e. assigning a team to develop the components, usually produces a very high quality component collection. Raytheon is one of the first companies to adopt this strategy [Lanergan]. In 1976 Raytheon assembled a small team to study the business applications that they develop. This group surveyed over 5000 Cobol programs and classified them by function. They were able to place all the programs in one of six categories (i.e. edit programs, update programs, report programs, extract programs, bridge programs, data fix programs).

The team then took 50 representative programs and analyzed them for redundant code. This study indicated that logic structures, a kind of program template, could be used to represent code which recurred in many of the applications. Studies such as this led Raytheon to create a collection of reusable logic structures and functional modules (i.e. code modules) for the development of all business applications at their company. The components collection at Raytheon has grown to over 5500 logic structures and 3200 functional modules. They report an average reuse rate of 60%.

The third approach to building a component collection, i.e. gathering component submissions and developing new components, has also proven to be a very successful strategy. Hartford Insurance Company adopted this approach to build up their collection of code modules, program skeletons, and logic structures for business applications [Cavaliere].

The successful component libraries that we found in the reuse literature have typically been functional collections; that is, the components are for specific application domains, such as business, graphics, numerical analysis, statistics, etc.. And these collections were usually assembled after doing some kind of domain analysis -- as was the case at Raytheon. Functional collections provide an excellent example of reuse of domain knowledge. Examples of functional collections, other than the ones we have already mentioned, include the statistical collections (SPSS and SAS), the mathematical component collections (NAG and MATLAB), and the user interface libraries for UNIX (X and X-Ray). In all these instances the reusable components in the collections have been carefully organized to cover the spectrum of needs of software developers in these application domains. Not only have the components been designed to be useful in different circumstances, they have also been designed to complement each other when combined. The architectures of these component collections reflect the domain analysis that went into them.

Levels of Reusable Software Information

Software information comes in different forms, some more amenable to reuse than others. They are:

- Code level information (e.g. subroutines, macros, programs)
- Specification and design level information (e.g. logic structures, program templates, dataflow diagrams, structure charts, state transition diagrams)
- Domain level information

The third level of reusable software information, domain information, is not widely recognized. Some identify reuse of domain knowledge with AI technologies (e.g. expert systems, knowledge based systems) and do not consider it to be at present a practical form of reusable software information. Nonetheless, domain knowledge (residing in a programmer's head) gets reused regularly in every software application that is created or modified. Reuse of personnel is a common way of reusing domain knowledge. But domain knowledge can also be embedded in the architecture of functional collections. Take for example the X and X-Ray libraries. The subroutines in these libraries are arranged in a distinct hierarchy, covering four levels of programming functionality:

- Dialogs (the highest level)
- Field Editors
- Intrinsics
- X Library Primitives (the lowest level)

The subroutines at each level are built upon the capabilities provided by the lower levels. This results in an orderly arrangement of components spanning a broad range of functional capabilities. Developers of applications using X windows reuse not only the subroutines in the X and X-Ray libraries, they also reuse a systematic technique for building windows which is enforced by the architecture of the libraries. The creators of the X and X-Ray libraries have succeeded in taking the results from their domain analysis (i.e. the key concepts and methods for developing an effective windowing system) and embedding them in the structure of the libraries so that no one can help but reuse the results of their analysis.

COMPONENT LIBRARIES: A NEAR RANGE TECHNOLOGY

Several kinds of reuse technologies are being used currently and a vast array of new technologies are being investigated. Figure 3 attempts to provide a conceptual framework for organizing the wide variety of reuse technologies which are presently used or being investigated. We present this framework only to indicate where reusable component libraries fit in the spectrum of reuse technologies. Some people have unrealistic expectations of the gains that can result from implementing reusable component libraries. It is important to point out that component libraries are a near range technology, and can bring about significant, but probably not an order of magnitude, increase in software productivity and quality [Tracz]. Greater gains in software reuse will come with the introduction of tools which automate parts of the software development and maintenance process, and the use of better representation techniques for abstract software components.

Some companies that have accumulated a good deal of experience using component libraries are now developing tools which attempt to automate some steps in the reuse of software. For instance, Nippon Telephone and Telegraph is prototyping a design system which provides the programmer with information about code components from an on-line library that can be substituted for portions of a program design specification [Yamamoto]. The trend in the mid run appears to be toward the development of tool-assisted reuse environments that incorporate effective techniques for representing and utilizing abstract components.

It is difficult to predict what reuse technologies will be available in the long run. Most observers agree that the software process will become fully automated, making the rapid prototyping approach to software development and maintenance truly feasible [Balzer]. The mid to far range technologies listed in Figure 3 represent experimental technologies being investigated at universities and other organizations; they may never be put into software production, however, they give a flavor of the reuse technologies of the future.

GETTING STARTED

Organizations wishing to pursue reuse would probably best start by accumulating experience through the implementation of a reusable component library. Those organizations which are not constrained by use of procedural languages (e.g. Pascal, Cobol, Fortran, C) might find it profitable to investigate reuse opportunities provided by Object-Oriented Programming (OOP) languages such as C++ and Objective-C. It is widely believed that OOP languages are better suited for software reuse than procedural languages because they make it straightforward to do data abstraction, and provide a facility known as "inheritance" which allows users to define new components (called classes) in terms of previously defined components. The optimized library management tools and facilities described in this paper should be able accommodate components created for both procedural and OOP languages. The issues related to setting up successful component collections (e.g. type of classification system,

SPECTRUM OF REUSE TECHNOLOGIES

Time Frame	Near Range	Mid Range	Mid to Far Range
Software Lifecycle	Waterfall (Traditional)	Waterfall --> Evolutionary	Evolutionary (Rapid Prototyping)
Reuse Strategy	Composition	Composition & Limited Automatic Generation	Automatic Generation
Technologies	(Component Libraries Comprised of:) Code Components o subroutines o macros o programs o OOP classes Specifications & Designs o algorithms o program templates o logic structures o skeletons o dataflow diagrams o structure charts Domain Analysis Info.	(Tool-Assisted Reuse Environments Utilizing:) Component Libraries Analysis & Design Tools OOP Languages Limited Code Generators Abstract/Formal Specification Techniques	(Automated Development Environments:) Application Generators Transformation Systems Problem Oriented Lang. Very High Level Lang. Knowledge Based Sys.

Figure 3

submission vs. designed libraries) apply to collections developed for both types of languages.

In general, designed libraries -- where components are assembled after doing some domain analysis -- are much more useful than submission libraries. But it is worth noting that good functional collections can only be created for domains that have already been sufficiently developed. It is probably a waste of time to try to find reusable software information in domains where few applications have been created. Thomas Standish from U.C. Irvine writes:

> "Compare computer graphics as it was... with the way it is today. Today we have a tremendous vocabulary of concepts which are useful for synthesizing artifacts: windowing, clipping, inking, rubber banding, latching, menuing, perspective transformations, shading, hidden-line elimination, and so on. There are great volumes of algorithms, representations, techniques, hardware devices, concepts, and abstractions to support engineering activities in computer graphics.
>
> "Only after a field ... has undergone considerable evolution and considerable traffic in applications can we expect the basis for a useful practice of software componentry to emerge.... We should not expect the useful arts of software reuse to emerge cheaply and rapidly in arbitrary subfields, and least of all in uncultivated ones."

REFERENCES

[Balzer] Robert Balzer, "Evolution as a New Basis for Reusability", *Proceedings of Workshop on Reusability* , ITT Programming, Stratford, Connecticut, September 7-9, 1983.

[Berard] Edward Berard, "Creating Reusable Ada Software", *Proceedings of National Conference on Software Reusability and Maintainability* , September 10-11, 1986.

[Braun] C.L. Braun, J.B. Goodenough, and R.S. Eanes, *Ada Reusability Guidelines* , Technical Report 3285-2-208/2, SofTech, Inc., April, 1985.

[Cavaliere] Michael J. Cavaliere and Philip J. Archambeault, Jr., "Reusable Code at The Hartford Insurance Group", *Proceedings of Workshop on Reusability* , ITT Programming, Stratford, Connecticut, September 7-9, 1983.

[DeMarco] Tom DeMarco, *Structured Analysis and System Specification* , Yourdon Press, 1978.

[Dongarra] Jack J. Dongarra and Eric Gross, "Distribution of Mathematical Software Via Electronic Mail", CSNET Coordination and Information Center, BBN Laboratories Inc., 10 Moulton St., Cambridge, MA 02238, November 19, 1985.

[Lanergan] Robert G. Lanergan and Charles A. Grasso, "Software Engineering with Reusable Designs and Code", *Proceedings of Workshop on Reusability* , ITT Programming, Stratford, Connecticut, September 7-9, 1983.

[McGregor] Scott McGregor, HP Corporate Computing Center, Personal Communication.

[Page-Jones] Meiller Page-Jones, *The Practical Guide to Structured Systems Design* , Yourdon Press, 1980.

[Podolsky] Joe Podolsky, "Observations on Software Reuse in Japan", in the HP *Software Reuse Digest* , Number 6, Edited by Mary Hsia-Coron, Corporate Engineering, January 1986. HP Internal Document.

[Prieto-Diaz] Ruben Prieto-Diaz, "Classifying Software for Reusability", *IEEE Software* , January 1987.

[StDennis] R.J. St. Dennis, "Reusable Ada Software Guidelines", *Proceedings of The Hawaii International Conference on System Sciences* , January 7-10, 1987.

[Tracz] Will Tracz, "Software Reuse: The State of the Practice", Tutorial Notes from COMPCON Conference, Spring 1987.

[Yamamoto] Shuichiro Yamamoto and Sadahiro Isoda, "SoftDA -- A Reuse-Oriented Software Design System", Paper Number 0730-3157/86/0000/0284$01.00, IEEE, 1986.

Session 5

DESIGNING FOR REUSABILITY

"An Environment to Promote Software Reusability at the Design Specification Level"
Peter R. Lempp, Software Products & Services, Inc.

"Software Reuse with Metaprogramming Systems"
Robert D. Cameron, Simon Fraser University

"An Object-Oriented Kernel for Composite Objects"
Ambrish Vashishta and Satyendra P. Rana, Wayne State University

An Environment to Promote Software Reusability at the Design Specification Level

Peter Lempp*

Abstract:

One approach to achieving high software reliability is reusing software already proven to work reliably in other applications. In order to be effectively reused, the software has to be available not only on the code level, but with the corresponding design specification as well as the requirements specification.

This paper first outlines the requirements for successful reuse of software at the design specification level, and then introduces selected features of the software engineering environment EPOS as an example of a framework for fulfilling these requirements. A specific case study of extracting higher-level design specifications from existing source code for the purpose of reuse is discussed in detail, and the advantages of programming language-independent specifications for future application are emphasized. The paper concludes by discussing uses of a knowledge-based component to effectively search for reusable software parts in existing databases of previous projects, and outlines anticipated directions in this ongoing research and development project to utilize advanced techniques.

Biographical Sketch:

Peter Lempp is group leader of Software Engineering and Manager of International Projects at a software engineering and consulting firm. His major areas of interest are software engineering, integrated project support environments, project management of software projects, and software quality assurance.

Mr. Lempp received an advanced degree in electrical engineering from the University of Stuttgart, Germany, in 1983. He was a special graduate student at Northwestern University, Evanston, IL, and has recently submitted his Ph.D. thesis in Electrical Engineering/Computer Science at the University of Stuttgart. Mr. Lempp has lectured and taught extensively in Europe and the United States on software engineering and project management techniques. He has also done significant research and development work in the area of software engineering environments, including work as project leader for the EPOS Support Environment and as a project team leader in the European ESPRIT program in 1984-85.

Mr. Lempp is a member of the IEEE, ACM and the German Computer Science Society (GI).

*Mailing Address:
c/o SPS Software Products & Services, Inc.
14 East 38th Street, 14th Floor, New York, NY 10016
Telephone: (212) 686-3790

An Environment to Promote Software Reusability at the Design Specification Level

Peter Lempp

1. Rationale

As computer usage for planning, implementation and tracking of projects increases, and as the complexity of the projects grows, we find an increasing dependency on software in all applications and missions. At the same time, we find that software is an increasingly significant cost element in the systems designed. To justify the major investment that software represents, we must ensure that the resultant system runs dependably, with minimal error levels.

To achieve the high reliability levels which are crucial to most large software/hardware projects, we can use three different strategies, possibly combining them to increase their effect:

- built-in features in the software itself (fault tolerance);
- tight quality assurance in the project, including verification and validation; and
- the reuse of software that has already been proven to be reliable in other applications /Luba86/.

Reuse promises to contribute to higher reliability, and to cost-effective software development as well: the integration of a proven system component into a new project is, under certain circumstances, less expensive than the creation and testing of a completely new one.

But despite extensive discussions in the literature of the virtues and benefits of software reuse (see, e.g., /Free87/, /BiRi87/), actual implementation of the practice in industry is still meager. Most often it is done informally, through team members who transfer their knowledge and individual pieces of software from one project to another. The number, background and prior project experience of the team members involved has a direct bearing on the degree to which software is being reused.

Aside from personnel considerations, the major reasons why reuse is not yet a standard practice include:

- often, the existing software is not well documented/designed, and thus unsuitable for reuse;
- the existing software is often written in one programming language (such as CMS-2 or FORTRAN) while the new project must use another.

For these reasons, we see two major prerequisites for effective software reuse:

- Reuse must be promoted on a higher (in many cases, programming language-independent) level.

- Reuse is possible only if the original software was developed within a software engineering environment which demands certain specifications and provides the results in a database form suitable for querying.[1]

In such a development and maintenance support environment, even data about changes and/or error corrections can be traced to establish an acceptable level of confidence in a piece of software.

2. Approaches and Requirements for Reuse of Design Specifications

The two principal approaches for reusing software are:

- Definition and implementation of standard software parts, often called "building blocks," which can later be used in the form of software libraries in projects (see, e.g., /McNi86/, /BABK87/). This approach is similar to low levels of hardware design, where standard integrated circuits are incorporated into a design.

- Reuse of already existing software from previous projects, if this software was developed according to known standards and methods.

Whereas the first approach will lead to comprehensive software libraries for specific applications which will undoubtedly have great value in the future, the second approach can be used without major additional effort, provided that the existing software was developed in a consistent manner within a suitable support environment. It should be recognized, however, that the methods of reuse are different also: Whereas building blocks offer the potential of reusing software parts "as is," perhaps with a different parametrization, existing software which was not specifically designed for reuse will in most cases require significant modifications; i.e., the reuse is total only for the concept underlying the piece of software. Thus the main objective of tool support is a structured manner of specification and assistance in finding "similarities" between existing software solutions and their previous requirements, and the problem at hand.

The first approach with its different facets of constructing finished or more generic building blocks is discussed extensively in the literature (e.g., /Free87/); we will especially concentrate on the requirements for a support environment for the second approach.

---

[1]If this was not done, a cost- and labor-intensive reverse engineering process must first be completed successfully to recapture these higher-level specifications.

These basic requirements for a software development support environment can be more precisely stated. The necessities are:

- Support for various levels of abstraction

  This is one of the key elements, since reuse is not restricted to a certain level. In general, small-size parts of certain instruction sequences such as calendar data calculation instructions, medium-size parts of common subroutines, and large-size parts of programs representing a function (e.g., a complete control function including a control strategy and algorithm) provide foundations of reuse.

- Availability of a "rich" design language

  This is essential if the initial software is later to be transformed into a comprehensive language such as Ada™, since only a matching of concepts and constructs allows direct reuse without major re-writing efforts.

  The design language must be capable of modeling all concepts of modern design according to software engineering practices, e.g., modularization, abstract data types, etc.

- Mechanisms for categorizing and attributing design parts

  Besides automatic capture of framework data, such as date of specification, author, etc., user-definable characteristics such as 'characteristic function' or 'test procedure', with their particular outcomes, must be specifiable even as late as in the query and searching process for reusable software parts in existing programs.

- Automatic code generators for different programming languages

  In order to suppress a source of error and avoid tedious verification of source code against design specifications, automatic generation of source code is highly desirable. Since the target languages in projects vary, different generators for the appropriate higher-order language must be available.

- Quality assurance tools for development (to assure that the reused design is *reliable*)

  Obviously the reuse of software demands even higher standards of quality (including robustness, i.e., controlled reactions to use outside the intended range of application, such as a negative input value in a square root solving algorithm).

Ada is a registered trademark of the U.S. Government (AJPO)

- Documentation and visualization features from various points of view

  To understand the full functionality of a piece of software which was not especially designed and documented for reuse, all aspects of it may be useful to present in a preferably correlated manner. Since people have different backgrounds and ways of thinking, the presentation from various points of view, e.g., more oriented towards the internal subfunctions or towards interactions of parameter and internal data, is essential to achieve good results.

- Query functions

  To allow either general search or confirmation of a specific hypothesis, easy-to-use query methods of the specification must be available.

- Capability to track changes and enhancements

  Remembering that one of our objectives is to reuse software parts which have been proven in "real world" applications, any modification history documenting problems, their fixes, enhancements and other indications of instability assist in establishing confidence levels in the pieces of software to be reused.

The fulfillment of these rather generally stated requirements is exemplified below by introducing certain concepts and features of the software engineering environment EPOS. Additional R&D on knowledge-based support, especially for the searching and identification process of reusable software, is discussed thereafter. Efforts aiming at the recapture of the design specification for software reuse, based on the EPOS environment, are described to illustrate further critical issues which must be dealt with before successful transference of software to other projects can be undertaken.

3. The Software Engineering Environment EPOS as a Framework for Reusability

3.1 Synopsis

The **E**ngineering and **P**roject-management **O**riented **S**upport System EPOS provides integrated computer assistance for the development and management of software/hardware systems. Originally designed in Europe and already

applied in numerous projects since 1981,[2] it was enhanced and adapted to American requirements in the last two years, and is now being used in major U.S. companies as well. It includes methods and a structured, homogeneous approach throughout the life cycle. The availability of project databases from numerous previous projects has initiated the enhancement of the original development support environment in order to be able to reuse parts of software and functional concepts developed in these projects.

The environment provides three related languages (for requirements, design and project management, respectively), and tool packages for management, documentation, method, analysis and code generation support /LaLe86/. Only the design representation and corresponding computer support relevant within our context will, however, be exemplified in the view of fulfilling the requirements for reusability assistance.

3.2 The EPOS-S Design Language

The design language is based on the definition of predetermined "design object" and relation types. The restriction to seven basic object types (*module, action, data, condition, event, interface* and *device*) with their various subtypes (e.g., *task, procedure, macro,* etc. for *actions*) ease the search for reusable software parts. In particular, four language features make the design suitable for later use in other projects:

- The modularization concept with its exact definition of interfaces between the modules. (Figure 3.1 on the following page depicts an analysis report of module interfaces and the graphical representation of part of the design after error correction);
- The possibility of assigning specific categories which can be effectively used to create new relations between objects, or to assign specific attributes to objects. (Figure 3.2 on the next page shows the definition of new attributes consisting of an attribute type and attribute instanciations);
- The high level of abstraction possible within the design specification (this feature is exemplified in Section 3.5 in a synchronization example based on a Petri net approach);
- The possibility to define abstract data types with their internal data structure and access procedures, independently of any implementation concept of the target programming language used.

---

[2]At the beginning of August, 1987, over 300 installations had been delivered worldwide.

```
==========================
CHECK ON MODULE CONSISTENCY
**************************

DESIGN OBJECTS ARE EXPORTED FROM THE FOLLOWING VISIBILITY LAYERS, BUT
ARE NOT IMPORTED BY A MODULE ON THE SAME LEVEL:

VISIBILITY LAYER OF MODULE:     DESIGN OBJECT:
---------------------------------------------------------------------
INPUT-OUTPUT (M2.1)
                                FURNACE-ON (A3.4)
```

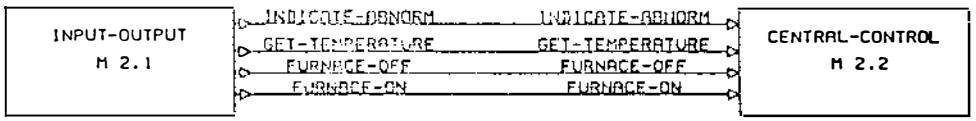


Fig. 3.1: Error report analyzing module interfaces, and documentation of the particular module interface after correction of the error

```
DATA VIEWPERIOD-FILE .

DESCRIPTION :

PURPOSE : "Each viewperiod file contain three parts:
          - a header record for administrative information
              (including a version id)
              -records for information about rise and set events for
              spacecraft
              -records for infromation about special spacecraft
              events.

             All records are of fixed length ".

CATEGORY : < ALIASES : 'VIEWS.Mnn' , 'SCHVIEWSnn' > ,
           < DOCUMENT : 'SSD-DOZ-5217-OP' , 'Vol. 1' , 'page 4-4' > ,
           < FUNCTION : 'external storage' > ,
           < CONCLUDED-FROM : 'design specification' > ,
           < CERTAINTY : 'certain' > .

DESCRIPTIONEND

DECOMPOSITION : HEADER-RECORD ;
                RISE-SET-EVENTS-RECORDS ;
                SPACECRAFT-EVENTS-RECORDS .

DATAEND
```

Fig. 3.2: Excerpt of the textual representation of a design object including user-defined specific attribute types and their instanciations (here, for example, "certainty" and "certain" respectively, which were introduced during a "reverse engineering" of the design; see Section 5).

- The high level of abstraction possible within the design specification (this feature is exemplified in Section 3.5 in a synchronization example based on a Petri net approach);

- The possibility to define abstract data types with their internal data structure and access procedures independent of any implementation concept of the target programming language used.

3.3 Quality Assurance with EPOS

This environment supports both approaches to quality assurance for achieving highly reliable software: the constructive method allowing comprehensive quality planning of goals, requirements, procedures and standards, (especially methodological restrictions of using only proven constructs from the nearly endless variety of design possibilities), as well as the analytical method of providing extensive checks and verification support.

Examples for the guidance to proven constructs are: provision of a pre-defined set of synchronization constructs (which can be combined to obtain all desirable complex situations), the restriction to structured program flow constructs, and the guidance through one of the six available design methods. The available analyses range from type checking to rather complex hierarchical consistency checks.

A recent survey showed that consistent application of a development support environment can have a very beneficial impact: two thirds of the interviewed project leaders of twenty-two major projects in fourteen different companies indicated that the environment helped to avoid errors in earlier development phases (Figure 3.3; /SPS87/). Fewer errors in the software from the start lead to potentially better, less distorted structures within the programs, since there is reduced need to fix and change specified structures. A clear structure is a major factor in identifying software parts later on for reuse.

3.4 Flexible Queries and Visualization

When searching for similarities between existing software and the problem assigned to be solved, success also depends on the flexibility with which various aspects of existing software can be queried and --if found to be at least somewhat related-- visualized. EPOS provides databased mechanisms, with the "atoms" for search being the design objects or related groups of objects. The most frequently used queries are for relations (related to a specific object) or for different aspects of the attributes of objects in the database. These include type dependencies (e.g., modules only), specific attributes (e.g., timing), naming patterns (e.g., *SORT*), or specific design level (e.g., 'level 2').

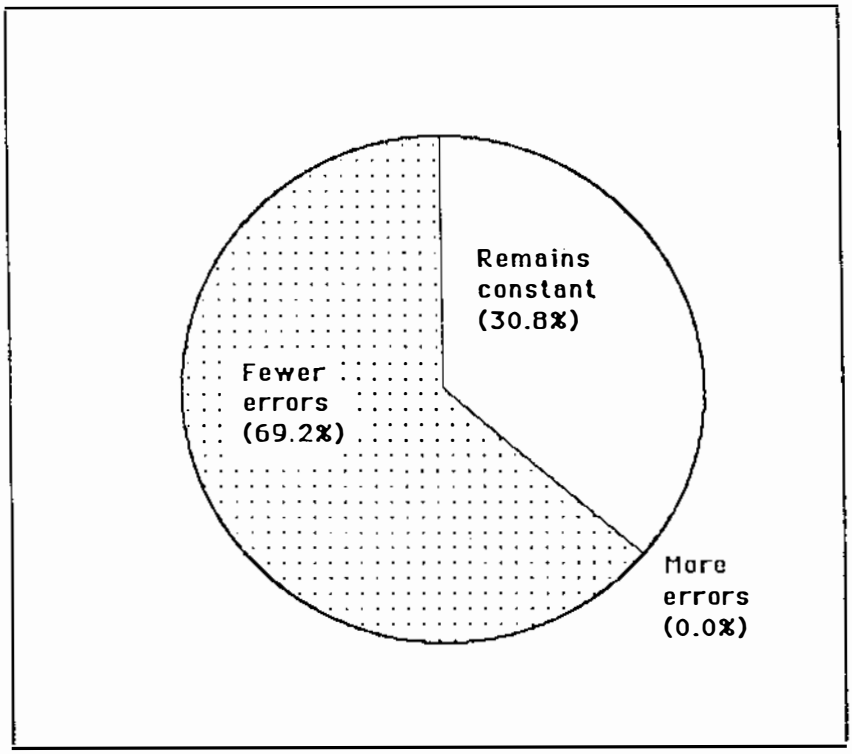


Fig. 3.3: Software quality improvement with a support environment:
2/3 of the projects surveyed reported fewer errors when using EPOS than when working without EPOS support /SPS87/.

The process of evaluating whether an identified piece in the existing software is really suited to solve the problem in another application is dependent on the facilities for visualizing the existing solution. Therefore, EPOS provides a large variety of representations, ranging from block diagrams to data flow and control flow diagrams. For the task of correlating points of view, a key factor in understanding the functionality of the software, two approaches have been identified as useful:

- Visualizing various aspects in parallel, with multiple windows, for correlation by the user (this approach is exemplified in Figure 3.4);
- Correlation in form of a special diagram (e.g., shown by a hierarchy/ control flow chart in the bottom left hand corner of Figure 3.4).

Both techniques, combined with specific queries, allow a browsing of the existing project databases, and provide a foundation for identifying reusable parts.

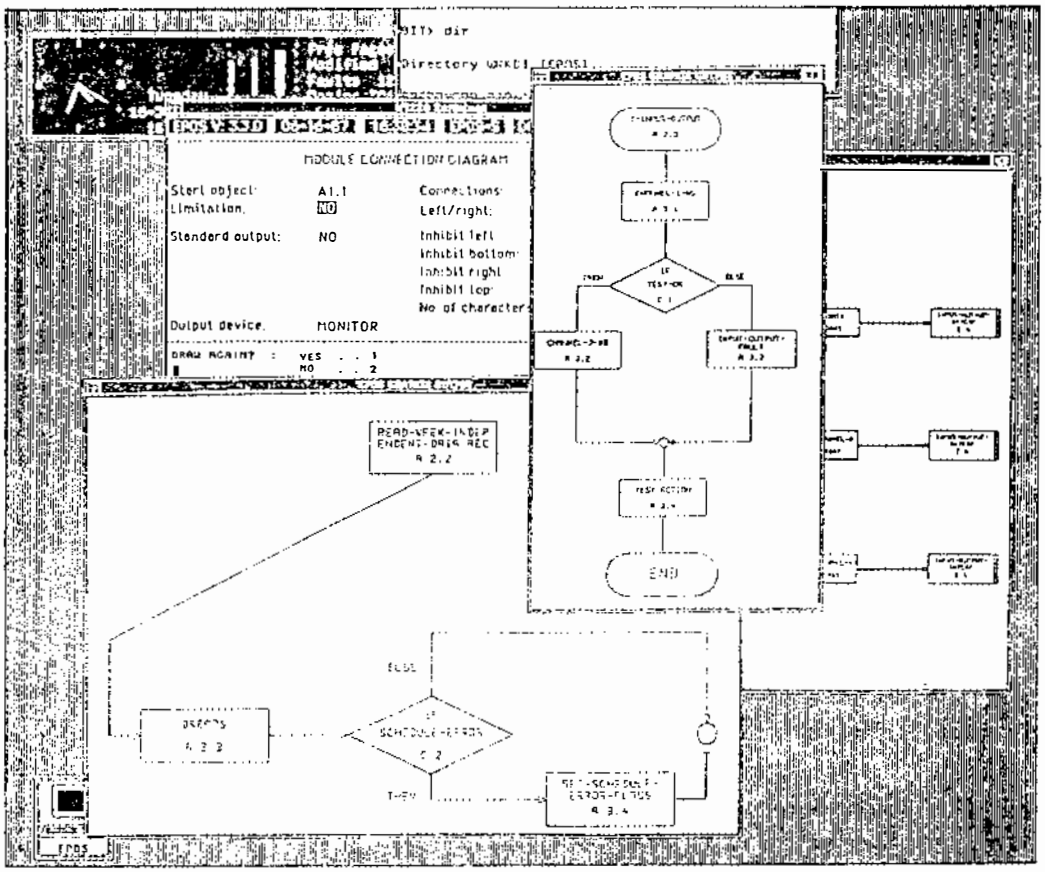


Fig. 3.4: Visualization of various aspects of an identified piece of software potentially suited for solving the problem at hand

3.5 Automatic Source Code Generators

Since reused portions are selected on the design level, the integrity of the reused software depends on the process of transforming the design, with its higher levels of descriptions, to the different concepts and features in the target programming language(s). This transformation task can be done by state-of-the-art code generators. Figure 3.5 depicts two examples of transformations produced by the Ada source code generator: the mapping of one module, and the transformation of a high-level synchronization concept in an EPOS design to the lower-level rendez-vous concept in the Ada language.

```
ACTION MODULE M.
DECOMPOSITION :
(/PROC1, PROC2, TASK1, TASK2/).
IMPORT-ACTION : TEXT_IO, PROC3.
IMPORT-DATA   : D.
EXPORT-ACTION : PROC1.
EXPORT-DATA   : VALUE, LEVEL.
ACTIONEND.

will be transformed to:

with TEXT_IO; use TEXT_IO;
with M2; use M2;
-- package which exports PROC3 and D.
package M is
   type LEVEL is (low, medium, high);
   VALUE : LEVEL;
   procedure PROC1 (x: in INTEGER);
end M;

with TEXT_IO; use TEXT_IO;
with M2; use M2;
package body M is
            .
            .
            .
   procedure PROC2 (Y : out INTEGER);
            .
            .
            .
end M;
```

```
ACTION PROCEDURE NARROW-ROAD.
DECOMPOSITION : PARALLEL
                (CARS-FROM-THE-LEFT,
                CARS-FROM-THE-RIGHT,
                TIMER).
SYNCHRO : EXCLUSIV  (SEND-LEFT-CAR,
                     SEND-RIGHT-CAR).
ACTIONEND.

is transformed to:

task CONTR1 is
      entry request;
      entry release;
end task;

task body CONTR1 is
      locked : boolean := FALSE
begin
      loop
        select
            when not locked => accept request;
            locked := TRUE;
        or
            when locked => accept release;
            locked := FALSE;
        or
            terminate;
        end select;
      end loop;
end CONTR1;
```

Fig. 3.5: Automatic mapping of design-level specifications to target source code:
a) module specification (target language Ada)
b) generation of a control task from a high-level specification, assuring the mutual exclusion of the blocks SEND-LEFT-CAR and SEND-RIGHT-CAR /LeZe87/

3.6 Evaluation during Application

Although the features described have been successfully used in identifying and reusing software from previous projects, they are still restricted in one sense: In the process of identifying reusable software parts, the user must already know some of the underlying principles of the researched software, the basic way it is structured, perhaps even its naming patterns. This basic familiarity with the existing software will be available if a department is basically working in the same application area, or if the researcher has even participated in the previous project.

For a more general approach to searching for "similar" reusable software, this "direct" search /Wagn87/ is not sufficient. We need a "knowledge-based" search which can correlate inexactly specified characteristics by definition of an "application environment." The following chapter reports on the research and development of a knowledge-based component to further increase effectiveness in software reuse.

4. Development of a Knowledge-based Component to Assist in Software Reuse

4.1 Structure and Components

Research has been completed to define a knowledge-based component which can interact within the EPOS system to advise the users, especially in the area of software reuse, and this "advisor" is currently under development /Laub87/. Figure 4.1 shows its basic components /Wagn87/.

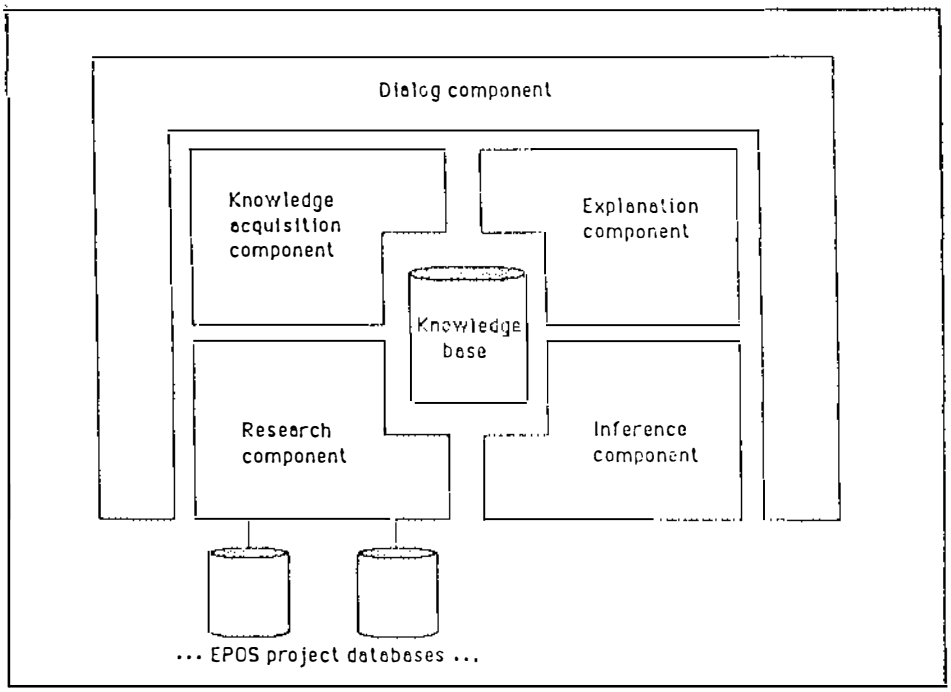


Fig. 4.1: Basic components of a knowledge-based system to evaluate databases of previous development projects

The dialog component is the interface between user and knowledge-based system. The knowledge base contains knowledge gained through experience, in the form of facts, rules and strategies formulated in a knowledge representation language. The contents of this knowledge base can be easily interchanged and expanded. The inference component then evaluates and interprets this knowledge description; depending on the course of a consultation,

the inference component can deduce new facts, i.e., "new" knowledge, from the existing knowledge base. The knowledge acquisition component is designed to support the description of information or knowledge from the users' areas of expertise, and to generate a knowledge base from such descriptions. Explanations, both during and after a consultation, can be given by an independent explanation component.

The research component implements the "intelligent" search for project results in EPOS databases of previous projects. This component comprises both "low-level" functions for string comparisons and selection of object types, etc., as well as functions which evaluate complex search queries and map these to lower functions /Wagn87/.

The frame-oriented knowledge representation language used possesses a spectrum of constructs which permit the engineer to map a problem area in appropriate and easily understandable fashion. Several concepts familiar from other knowledge description languages, such as frame hierarchies and inheritance of values, are incorporated to create a comfortable environment for constructing expert systems /LaPe87/.

The knowledge representation language allows us to describe knowledge in six different formats, or "knowledge objects":

FRAME	.....	Definition of characteristics
RELATION	.....	Definition of object relations
RULE	.....	Definition of causal connections
STRATEGY	.....	Inference engine
PROCEDURE	.....	Integration of procedural components
INSTANCE	.....	Definition of attributes

With the knowledge object STRATEGY, the control flow during a consultation of the expert system can be defined. Here the knowledge engineer specifies in what sequence instances will be created, when questions will be posed to the user, when the inference mechanisms will be activated, and how the results will be presented to the user.

The PROCEDURE allows the expert system to access external programs. These may be graphic output functions, or mathematical library routines.

A FRAME defines a class of similar objects within the problem area. We can create hierarchies of FRAMEs. The objects' characteristics are specified as ATTRIBUTES of the FRAME. These would include information about data type, range and possibly a default for the attribute's value.

The RELATION object describes the relationships which may exist between individual frames. A special feature of the knowledge representation language permits us to assign a value to a relation between two or more objects.

RULES are used to describe the causal or heuristic relationships within the problem domain; such dependencies permit the system to infer new facts from

those already known. These rules may be used in goal driven (backward chaining) or in data driven (forward chaining) modes; they consist of a premise-part and a conclusion-part.

The definition of INSTANCEs in the knowledge base meets the need for providing the expert system, at the beginning of a consultation, with a start-up basis of known objects and facts from which conclusions may be drawn /LaPe87/.

4.2 Extended support for the reuse of existing software

A search description language is being developed to permit description of information applicable to more than one project, and to allow formulation of search queries in external databases.

With the help of this simple command language we can search interactively in EPOS project databases for reusable results. Usually, searches in a large project database or even in several databases are very time-consuming. For this reason the knowledge-based system contains batch mode search facilities also.

For example, a search request may be input with the following command:

SEARCH ALL module WITH category = 'input' OR 'output' ON LEVEL = 1,2

This request restricts the search to all design objects of type "module" and the two uppermost levels of a hierarchical design. Furthermore, it selects only those result descriptions which possess the attribute "category" with the outcome "input" and/or "output". The search language is directly based on a described data model of the EPOS objects (see Section 3.2 above).

To extend the capabilities of such a search beyond simple deterministic queries, new procedures in the knowledge-based system permit the use of search targets which are not sharply circumscribed on the one hand, and on the other hand permit computer-supported similarity comparisons. The procedure selected provides for a range from -1 (no identical characteristics) to +1 (conforms 100% to the search target) in stating the results of the similarity comparison.

To start up an "imprecise" search, the user must select or input so-called "contexts" to give the research component a basis for evaluation and comparison. A context consists of a hierarchical arrangement of concepts which are linked by rules. In establishing the hierarchical arrangement, the concepts are assigned values between -1 and +1; these fix the contentual relationship between primary concepts and sub-concepts. By evaluating the contexts, the research component can assign the project results to specific problem or topic domains.

In addition, the search description language provides constructs which we

can use to state short descriptions of the most important project characteristics, as regards both organization and content.

It uses the following four description formats for its search descriptions and project classifications /Wagn87/:

CONTEXT	.....	to define the context in which the search shall operate
SEQUENCE	.....	to define complex search queries
PROJECT	.....	to specify short project descriptions
CHARACTERISTIC	.....	to define relevant project characteristics

With CONTEXT we state rules to define relationships between terms. A fixed scale is used to define the contentual distance between contexts. We may also state synonyms for terms. During a search with contexts these synonyms are inserted to expand the terms used.

With the object SEQUENCE we can formulate extensive search queries. In contrast to interactively input short queries, the SEQUENCE descriptions are not deleted after the search, and can thus be re-used for later (batch) searches.

With PROJECT we can describe both organization and contents of a previously completed project. Important organizational information includes the database identifier, pathname for calling up the database, time of project execution, project team members, and statement of major sub-projects.

The characteristics for which a project's contents can be flagged depend on the application area, and must therefore be defined by the user for his individual problem domain. To accomplish this, objects of type CHARACTERISTIC are used to define the core characteristics typical for the problem area.

With the described knowledge-based system, searches for pieces of software suited to a problem at hand can be done beyond the more restricted direct search. With this extended capability, the project databases of hundreds of projects done during the last five years with EPOS support are expected to become even more of an asset.

5. Reverse Engineering from Existing Code to Design Specification to Increase Reusability

5.1 The Reverse Engineering Concept

A huge amount of software was written during the last decades in first- or second-generation high-order programming languages like FORTRAN, COBOL, etc., without the structured approach and computer assistance of a software engineering environment. Most of the earlier software has unstructured elements, and its documentation is incomplete, or at least not up-to-date. For

these reasons much of the software is not directly suited for reuse in other applications, but it has often been running reliably for years, and it represents a great know-how value.

Having identified the suitable level of software reuse to be the design level, and recognizing the non-existence of design specifications for most of the software developed earlier, we may find it worthwhile to invest in "reverse engineering." This effort would be particularly justified if the software represents a very common application and incorporates a great deal of heuristic knowledge.

Reverse engineering is the process of unraveling the system and software to its earlier lifecycle development phases /Saya87/. During the reverse engineering process, higher levels of abstraction, especially on the design specification level, are reconstructed from the source code and other available documentation. Figure 5.1 depicts the task graphically, and shows one of the major applications of the reverse engineering process: the functionality transfer of software written in some non-standard language to software written in Ada.

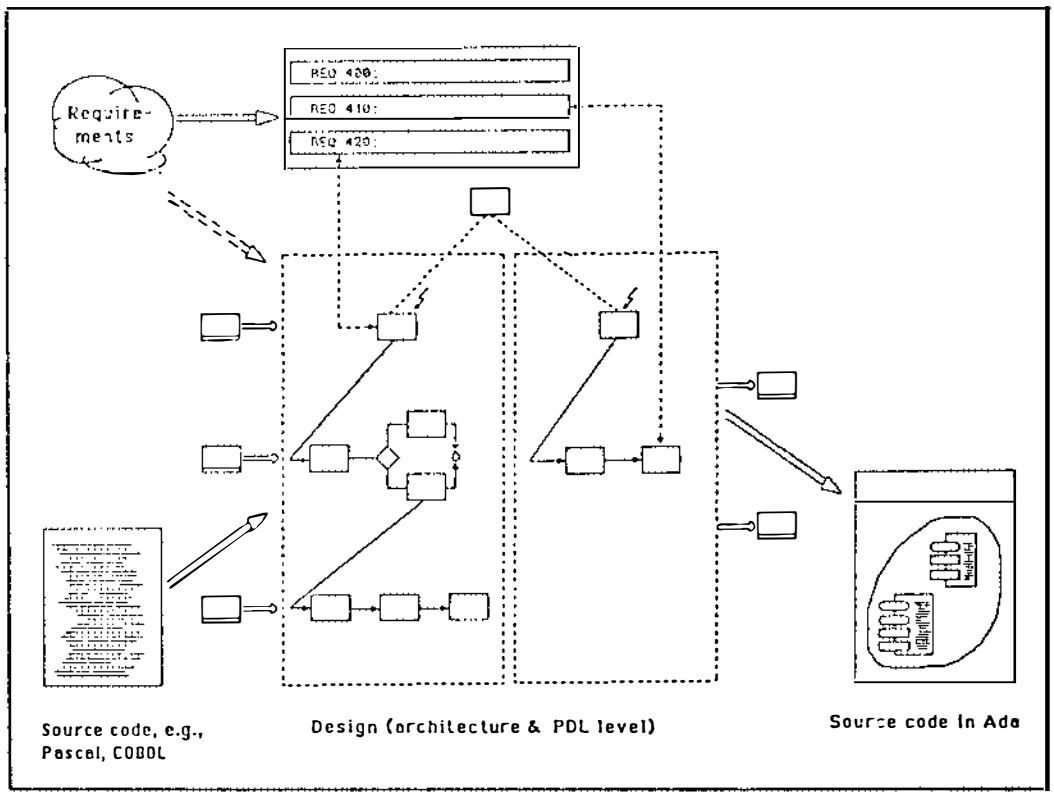


Fig. 5.1: Reverse engineering of existing software (to be reused on the design specification level and, e.g., regenerated in Ada)

As Figure 5.1 also shows, reverse engineering normally starts not with the source code alone, but also takes existing documentation, e.g., informal high-level requirements specifications and especially input/output descriptions, etc., into account.

The requirements for successful and efficient reverse engineering are to some extent the same as have been identified for software reuse: the basic pre-requisite is again an environment which provides comprehensive specification capabilities for the different representations [requirements, especially high- and low-level design (SDL and PDL level)], and tracing, analysis, querying and documentation support such as is provided by the EPOS environment introduced above.

The specification capability determines the degree of fulfillment of the design language/modeling requirements, which also include the necessity to model low-level as well as abstract levels of the design. The possibilities of flexible visualization and interactive queries influence the degree to which the functionality grasping requirements are met. Since reverse engineering is to a large extent based on "understanding" a piece of software in order to abstract specifications, various aspects characterizing the software's functionality (such as input/output data, controlflow, data used and produced) often need to be seen at once or at least in a fast viewing cycle.

Indispensable to the verification needed for reverse engineering is the requirement to produce source code from design specifications in the programming language(s) of the original piece(s) of code. In addition, an analysis package to evaluate the design itself is advisable. Depending on the design language facilities, the checks may vary, but should at least contain completeness and consistency analyses.

5.2 Methods and Tools for the Reverse Engineering Process

The overall procedure to (re-)establish a design specification can be broken down into the following three fundamental activities:

- formal source code analysis
- functionality determination and specification, and
- verification of reverse engineered design specifications

The first activity aims at the capture of all basic structures, such as internal/external data, controlflow and dataflow. It also comprises:

- identifying the physical and logical "modules";
- (re-)specifying the principal calling and controlflow structure;
- identifying and clarifying important data and their naming conventions.

The functionality determination starts by applying any pertinent additional knowledge from top-down documentations; this will often lead to some beginning at input/output or known library modules, and consists mostly of:

- finding data threads through the software, and thus extrapolating from known data to derived data;
- defining abstractions by identifying logical groupings (indicated by a variety of characteristics varying from one programming language to another and from application to application).

The verification of the design for consistency and for correctness is accomplished by generating the source code in the original programming language and comparing with the given source code. Additional guidelines apply for (re-)establishing high-level requirements, and for using top-down approaches as well as bottom-up ones.

The principal tool components desirable to automate the process, and thereby make it cost-effective, are shown graphically in Figure 5.2.

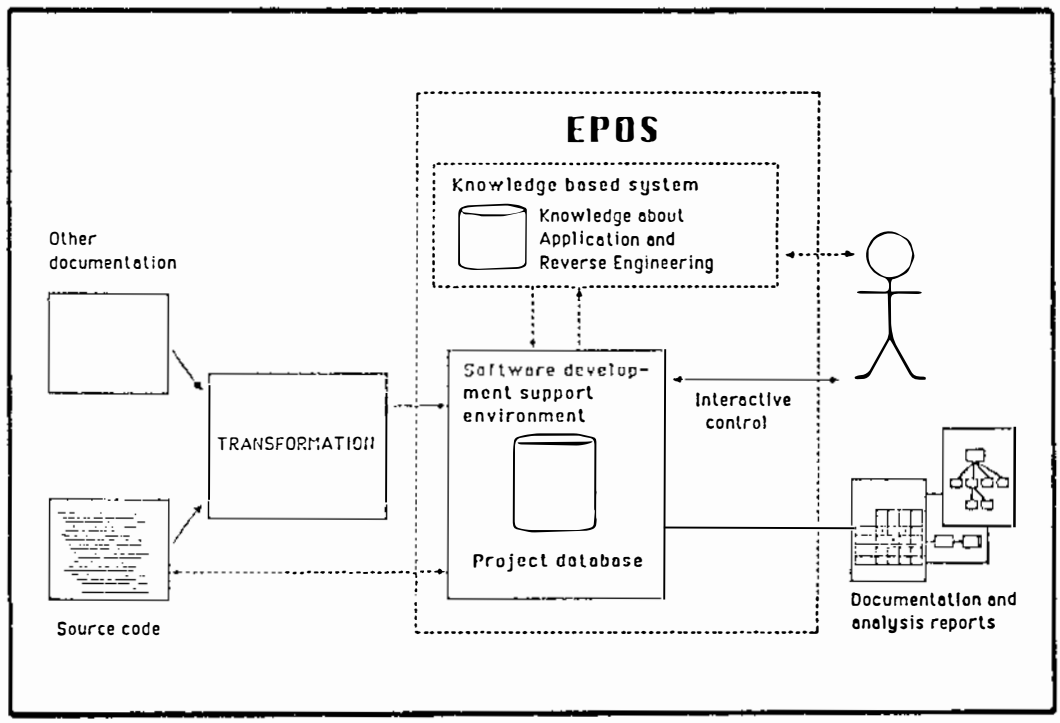


Fig. 5.2.: The tool components in automating reverse engineering

The transformation tools scan and map source code into low-level design, and convert other available documentation into an appropriate representation in the software engineering environment. With assistance of the EPOS environments' visualization and query tools, the functionality determination can be supported to a large degree, even when leaving all creative and active aspects to the user who states the requests and queries. A knowledge base for typical functionalities in specific applications might be used to obtain suggestions in the respecification process about specific functionality hypotheses, based on similar patterns of a function identified in the knowledge base.

5.3 Applications of Reverse Engineering

Complex reverse engineering projects for major avionics systems have been undertaken and successfully completed, utilizing the software engineering environment introduced above (EPOS). One typical example was the TORNADO reverse engineering project, performed as a joint intercompany effort under the direction of West German MOD.

This project was undertaken to recapture requirements and design specifications in connection with the main avionics computer and software elements involving control, executive, navigation and targeting systems.

The problems confronting engineers were typical of the difficulties encountered in any reverse engineering project. During the original development, the avionics package consisted of software requirements specifications, design specifications, and programming code. This documentation was originally developed for the most part manually and without any form of clear, consistent interactive relationship between the elements. During the extended life of the project, programming code was modified on innumerable occasions to meet maintenance or partial upgrade requirements. This modified code was accompanied by varying degrees of annotation and documentation, and often was without complete reference to software requirements or design specifications. As time went on, the design specification documents lagged well behind actual code implementation. The reverse engineering project was undertaken to realign software requirements documents, design specifications and code to obtain a firm basis for further maintenance and incorporation in other projects.

The TORNADO reverse engineering project used the existing program code as the basis for objective information. This code, along with limited annotations and program commands, as available, was then used to tie into old design specifications. In this fashion the specification tree, as well as the requirements documents, were recreated so that at the end of the project a complete EPOS-S design specification was accomplished. The finished specifications are now available for maintenance and reuse in further avionics projects.

Dealing with software written in assembly code, this particular reverse engineering project required approximately eight engineers for a year and a

half. The investment is still paying off, not only because the life span of the software in use has been extended, but also because the project gathered and respecified a valuable fund of experience, information and commonly used functions in avionics.

In another just-completed project which was more in the nature of a feasibility study, the design specifications of a near-realtime control software program written in FORTRAN were recaptured and compared to existing documentation. The respecification in the EPOS database, interrelated and in a form ready to be queried about its various characteristics, succeeded in improving the specification to the level of potentially being reusable.

6. Conclusions

Software reuse must become part of the quality assurance efforts in software development projects, because of its great potential to increase reliability. Software development support environments like EPOS can contribute significantly to software quality during development and maintenance. The formal specifications, particularly of certain relationships and attributes, can provide a basis of reuse even if the software has not been designed for reuse.

If searching and querying schemes (which ultimately must be knowledge-based to be adapted easily to different applications) can be defined to find similarities in design or even concepts, then effective reuse of software is possible. It has also been shown that in order to conform prior work to the "standard" set by an environment, it is even possible to reverse engineer software (with considerable effort, but part of this effort would be necessary to maintain this software anyway).

Future research will concentrate on the automation of the reverse engineering process, and on knowledge-based support in searching for reusable software.

Acknowledgment: The author owes thanks to Dr. Alex Rainer, who reviewed the paper and suggested various improvements. These contributions added considerably to the focus of the paper.

References

/BABK87/ Burton, B.; Aragon, R.; Bailey, S.; Koehler, K.; and Mayes, L.: "The Reusable Software Library". IEEE Software, July 1987, pp. 25-33.

/BiRi87/ Biggerstaf, T.; Richter, Ch.: "Reusability Framework, Assessment and Directions." IEEE Software, March 1987, pp. 41-49.

/Free87/ Freeman, P.: "Tutorial: Software Reusability." IEEE Computer Tutorial Society Press, Washington, D.C., Cat. No. EH0256-8, 1987.

/LaLe86/ Lauber, R.; Lempp, P.: EPOS Overview. New York: SPS Software Products & Services, Inc., 1986.

/LaPe87/ Lauber, R.; Permantier, G.: "A Knowledge Representation Language for Process Automation Systems." 10th World Congress on Automatic Control, IFAC '87, Munich, West Germany, July 1987.

/Laub87/ Lauber, R.: "Automated Software Production." Preprints AIAA/-NASA International Symposium on Space Information Systems in the Space Station Era, Washington D.C., June, 1987.

/Lemp87/ Lempp, P.: "Integrated Support in the Development and Maintenance Project 'Heating Control System' with the Software/-Hardware Production Environment EPOS." National Conference on Methodologies and Tools for Real Time Systems, Tysons Corner, VA, March 1987.

/LeRa87/ Lempp, P.; Rainer, A.: "Systems Development Support Environments - State-of-the-Art and Requirements for the 1990's." Preprints CASE '87. First International Workshop on Computer-Aided Software Engineering, Cambridge, MA, May 1987.

/LeZe87/ Lempp, P.; Zeh, A.: "Developing Ada™ Software Using the Software/Hardware Production Environment EPOS." SDIO Ada Conference and Tools Fair, Washington, DC, January 1987.

/Luba86/ Lubas, M.D.: "Affording Higher Reliability through Software Reusability." IEEE Software Engineering Notes, Vol. II, No.5, October 1986, pp. 39-42.

/McNi86/ McNicholl, D.: "Common Ada™ Missile Packages." Proc. National Conference on Software Reusability and Maintainability, Tysons Corner, VA, September 1986.

/SPS87/ SPS Software Products & Services, Inc.: "Short Summary of the Survey of Industrial Experiences with EPOS." New York, July 1987.

/Wagn87/ Wagner, B.: "The Project Advisor - A Knowledge-based Tool System for the Reuse of Project Results." (in German); Preprints, Technical Symposium on Support Environments, Esslingen, West Germany, Sept. 1987.

/Saya87/ Sayani, H.H.: "Applications in Reverse Engineering." Proc. Software Reusability and Maintainability Conference, Washington, D.C., March 4-5, 1987.

Software Reuse with Metaprogramming Systems

Robert D. Cameron
School of Computing Science
Simon Fraser University

Abstract

Metaprogramming systems are systems designed to support the writing of *metaprograms*, i.e., programs about programs. In essence, such systems provide a set of program manipulating subprograms implemented in one language (the *host* language) to manipulate object programs of another language (the *target* language).

The use of metaprogramming systems to support a style of software reuse which is a cross between program generation methods and parameterized subprogram libraries is described. In this approach, a reusable software unit consists of two components: a parameterized template for the reusable software, and a metaprogram which specializes this template for a given application. This approach is illustrated through a completely worked out example: the generation of input-output routines for arbitrary Pascal enumerated types.

Biographical Sketch

Robert D. Cameron received B.A.Sc. and Ph.D. degrees in Electrical Engineering from the University of British Columbia in 1977 and 1983 respectively. Presently, he is an Assistant Professor in the School of Computing Science at Simon Fraser University. His research interests include programming languages and systems, program manipulation and integrated programming environments.

Dr. Cameron is a member of the the Association for Computing Machinery and the ACM Special Interest Groups on Programming Languages and Software Engineering. He is also a member of the Institute of Electrical and Electronics Engineers and the IEEE Computer Society.

1. Introduction

Metaprogramming systems are systems designed to support the writing of *metaprograms*, i.e., programs about programs. Such systems provide a set of program manipulating subprograms implemented in one language (the *host* language) to manipulate object programs of another language (the *target* language). In essence, the set of subprograms comprising a metaprogramming system implement an abstract data type for the programs and program fragments of the target language.

Following the GRAMPS approach [2], metaprogramming systems can be constructed for virtually any target-host pair. The work reported in this paper has all been implemented with Pascal MPS, a metaprogramming system with Pascal as both target and host language [3].

Metaprogramming systems can be used to support a style of software reuse which is a cross between program generation methods and parameterized subprogram libraries. In this approach, a reusable software unit consists of two components: a parameterized template for the reusable software, and a metaprogram which specializes this template for a given application. This approach is more powerful than simple parameter-substitution methods, however, because the metaprogramming system allows arbitrary manipulations of the template to be programmed.

This paper illustrates the metaprogramming approach to software reuse through a completely worked out example involving reusable I/O routines for Pascal enumerated types. Section 2 describes the example problem in detail and considers various alternative reuse mechanisms for its solution. Section 3 then presents the solution to this example using the metaprogramming approach. Section 4 concludes the paper with a general discussion of other ways in which metaprogramming systems can support software reuse.

2. A Reuse Problem

An interesting problem in software reuse is the generation of readers and writers for arbitrary Pascal enumerated types. For example, consider an enumerated type `Colors` as shown in Figure 2-1.

```
Colors =
  (black, brown, red, orange, yellow, green, blue, violet, gray, white);
```

Figure 2-1: The `Colors` Enumerated Type

A writer for the `Colors` enumerated type is a subprogram which takes as input a value of the enumeration (e.g., *brown*), and generates the corresponding character string (`'brown'`) to a text file. Such a subprogram is quite straightforward as shown in Figure 2-2.

```
procedure WriteColors (var f : TEXT; x : Colors);
begin
  case x of
    brown : write (f, 'brown');
    black : write (f, 'black');
    red : write (f, 'red');
    orange : write (f, 'orange');
    yellow : write (f, 'yellow');
    green : write (f, 'green');
    blue : write (f, 'blue');
    violet : write (f, 'violet');
    gray : write (f, 'gray');
    white : write (f, 'white')
  end
end;
```

Figure 2-2: Writer for the `Colors` Enumeration

Conversely, a reader for an enumerated type reads in character strings and returns corresponding values of the enumeration. The implementation of readers is somewhat more complex than that for writers, however, as shown in Figure 2-3. These two procedures provide enumeration I/O for a particular type;

```
procedure ReadColors (var f : TEXT; var success : BOOLEAN; var x : Colors);
  type
    wordType = array [1..256] of CHAR;
  var
    word : wordType;
    wordLength : INTEGER;
    ch : CHAR;

  function CheckWord (testWord : wordType; testLength : INTEGER) : BOOLEAN;
    var i : INTEGER;
  begin
    if testLength = wordLength then
      begin
        i := 1;
        while (i <= testLength) and (testWord[i] = word[i]) do i := i + 1;
        CheckWord := i > testLength
      end
    else CheckWord := FALSE
  end;
begin
  ch := ' ';
  while not EOF(f) and (ch = ' ') do READ(f, ch);
  wordLength := 0;
  while not EOF(f) and (ch in ['A'..'Z', 'a'..'z', '0'..'9']) do
    begin
      wordLength := wordLength + 1; word[wordLength] := ch; READ(f, ch)
    end;
  success := TRUE;
  if word[1] in ['b', 'g', 'o', 'r', 'v', 'w', 'y'] then
    case word[1] of
      'b' : if CheckWord('blue', 4) then x := blue
            else if CheckWord('black', 5) then x := black
            else if CheckWord('brown', 5) then x := brown
            else success := FALSE;
      'g' : if CheckWord('gray', 4) then x := gray
            else if CheckWord('green', 5) then x := green
            else success := FALSE;
      'o' : if CheckWord('orange', 6) then x := orange
            else success := FALSE;
      'r' : if CheckWord('red', 3) then x := red
            else success := FALSE;
      'v' : if CheckWord('violet', 6) then x := violet
            else success := FALSE;
      'w' : if CheckWord('white', 5) then x := white
            else success := FALSE;
      'y' : if CheckWord('yellow', 6) then x := yellow
            else success := FALSE
    end
  else success := FALSE
end;
```

Figure 2-3: Reader for the `Colors` Enumeration

the reuse problem, then, is how to transform these procedures into reusable generalizations which apply to any enumerated type.

This problem is interesting because it is conceptually simple, but difficult to solve using parameterization techniques. Some reuse problems can be solved by simply defining a generalized version of a procedure which takes an extra parameter. As with many reuse problems, however, this problem essentially requires a programming language *type* as a parameter. Such generalizations are not possible using the standard parameter passing mechanisms (call-by-value and call-by-reference) of conventional languages. In this regard, the generics of Ada are a major development for software reuse

because they do allow types as parameters.

For the present problem, however, the addition of an Ada-style generics facility to Pascal would still be insufficient, even with the further extensions to promote reusability suggested by Goguen [4]. This is because a type parameter only allows access to properties of a type that are made available using language mechanisms. For generalized enumeration I/O, the property of interest is the character-by-character representation of the identifiers of an enumerated type, but this property is not available through any language mechanism in Pascal. Although a further language extension could ameliorate this difficulty, language extension for each new reuse problem is surely not a satisfactory approach.

In essence, the reuse problem for enumeration input-output is an instance of a class of reuse problems which are most readily solved by parameterization over the *source text* of a program. Reusable generalizations in this class consist of program templates which can be specialized to generate the particular program text which applies in some instance. In the case of enumeration input-output the reusable generalization would be a mechanism which takes as input the source text of an enumerated type definition in Pascal and produces as output the particular source code for the reader and writer subprograms which apply to that type.

The conventional approach to such source-level parameterization is to use a text-based facility such as a macro processor. Perhaps the most widely known example of such a processor is the macro preprocessor for the C programming language [6]. Although preprocessors of this kind do achieve a level of source-text parameterization that contributes to software reuse, they are again inadequate for the present problem. In this case, the complication is that an enumeration type may have an arbitrary number of enumeration constants. In the creation of a writer for the type, then, an arbitrary number of clauses for the case statement must be created. Although this is beyond the capability of simple macro substitution, it is conceivable for a slightly more powerful macro processor. In the creation of the reader for an enumerated type, however, not only must an arbitrary number of clauses be created, but further processing is required to somehow group clauses together based on the first letter of the enumeration identifier (see Figure 2-3). Problems of this sort suggest that the power of a programming language is required to achieve truly adequate source-level parameterization in a text-based facility.

Rather than a text-based approach, however, this paper offers a syntax-based approach to source-level parameterization. In this approach, the macro programs that would manipulate and specialize program templates on a textual basis are replaced by metaprograms which perform such manipulations on a syntactic basis. This allows the programming to be simplified by abstracting away from the details of the text-level program representation. Another advantage is that metaprogramming systems will provide additional facilities for processing programs as data objects, such as a symbol lookup facility. Furthermore, using a metaprogramming system whose host language is the same as the target language eliminates the need to learn a separate language to construct such generalizations. Although some effort is involved in learning to use the set of subprograms that comprise a metaprogramming system, that effort will pay off not only in software reuse applications but also in software maintenance [10] and other in-house software manipulation activities. The detailed solution of the generalized enumeration I/O problem using metaprogramming is the topic of section 3.

In practice, a common method for providing generalized I/O for enumerated types is to build it into the compiler, as in Berkeley Pascal [5]. This reflects both the difficulty of providing generalized enumeration I/O using other techniques and the practical importance of enumeration I/O in Pascal programming. One drawback with this approach is that programs which use such a compiler facility have reduced portability. Nevertheless, the incorporation of programming knowledge into a compiler has long been an effective and efficient mechanism for reuse of that knowledge. This suggests that an important research direction to promote software reuse is the concept of extensible compilers [9]. Indeed, if such extensibility is provided through source-to-source techniques [7], then intermediate output from such a compiler would be portable to other installations.

A related approach is to specify an enumeration I/O facility as a standard part of the programming language, as in Ada [1]. Specification of facilities in a language definition is again a very effective form of reuse since all (conforming) compilers must provide such facilities. From the reuse point of view, then,

perhaps small languages are not so beautiful. Alternatively, the definition of separate languages for each new problem domain may be an effective way to achieve software reuse, as in the Draco project [8].

3. The Metaprogramming Solution

The reuse problem for enumeration I/O can be solved using metaprograms which generate the appropriate type-specific routines given an enumerated type definition as a parameter. In general, metaprograms such as these make use of program templates which they specialize for a given application. In the case of enumeration I/O, however, a program template is necessary only for readers; writers are simple enough that they can be generated from a metaprogram alone.

Figure 3-1 shows the metaprogram to construct enumeration writers.

```
function MakeEnumerationWriter (enumTypeDecl : Node) : Node;
  var i : INTEGER;
      OutputStr : StringType;
      EnumParm, FileParm, CaseClauses, ids, id : Node;
begin
  EnumParm := MakeIdentifier(Str('x'));
  FileParm := MakeIdentifier(Str('f'));
  ids := EnumerationConstantsOf(DefiningTypeOf(enumTypeDecl));
  CaseClauses := NullList(CaseClauseList);
  for i := 1 to Length(ids) do begin
    id := NthElement(ids, i);
    OutputStr := AddQuotes(CoerceIdentifier(id));
    Insert(
      CaseClauses,
      -1,
      MakeCaseClause(
        List1(ExpressionList, id),
        MakeWriteStatement(
          List2(WriteParameterList,
                MakeWriteParameter(FileParm, nil),
                MakeWriteParameter(MakeString(OutputStr), nil)))))
  end;
  MakeEnumerationWriter :=
    MakeProcedureDecl(
      MakeIdentifier(
        AppendStrings(
          Str('Write'), CoerceIdentifier(TypeNameOf(enumTypeDecl)))),
      List2(
        ParameterList,
        MakeVariableParameterSection(
          List1(IdentifierList, FileParm), MakeIdentifier(Str('TEXT'))),
        MakeValueParameterSection(
          List1(IdentifierList, EnumParm), TypeNameOf(enumTypeDecl))),
      MakeBlock(
        nil {no label declarations},
        nil {no constant definitions},
        nil {no type definitions},
        nil {no variable declarations},
        nil {no internal subprograms},
        List1(StatementList, MakeCaseStatement(EnumParm, CaseClauses))))
end;
```

Figure 3-1: Metaprogram to Construct Enumeration Writers

The input and output of the metaprogram are both objects of the data type *Node*, which is the fundamental data type of the metaprogramming system for representing programs and program fragments as objects. Thus, given the enumerated type definition of Figure 2-1 as a *Node*, this metaprogram produces as output the *Node* representation of the *WriteColors* subprogram shown in

Figure 2-2. Conversion between *Node* representations and their textual form is handled by parsing and prettyprinting routines.

Inside the metaprogram, various metaprogramming system routines are used in the creation of the enumeration writer subprogram. The basic operation of the metaprogram is to iteratively construct the case statement which provides the main functionality of an enumeration output routine and then embed this case statement in the appropriate code which completes the definition of that routine. The details of this construction process are as follows. First of all identifier nodes *EnumParm* and *FileParm* are created for the subprogram parameters. Then the list of identifiers which comprise the enumerated type are set up in the variable *ids*; note that the program input as illustrated in Figure 2-1 consists of a type declaration which has both a defined type-name and its defining-type. Next an empty list of *CaseClauses* is set up in preparation for the main loop. The loop iterates through the elements of the enumeration, setting up the identifier node *id* appropriately each time. In preparation for constructing the output statement for this enumeration value, an *OutputStr* is created which encloses the string representation of the identifier node (determined by *CoerceIdentifier*) in quotes. The main action of the loop is then to insert an appropriately constructed case clause for this enumeration value into the *CaseClauses* list being built. The *Insert* routine does so, using the parameter `-1` to indicate that the insertion is to be made at the end of the list. Each case clause is constructed from two components, an expression list which is the list of case constants for the clause and a statement which implements the action of the clause. The case-constant list is simply a single element list consisting of the enumeration constant. The statement of each clause is a write statement having two parameters. The first write parameter is simply the file variable for the output text file, and the second is the string to be written out, converted to a *Node* from its *StringType* representation through the *MakeString* function. In Pascal, write parameters have an optional field descriptor which controls formatting; in this case the field descriptors for both parameters are omitted by specifying **nil** as the second argument to *MakeWriteParameter*.

Once the main loop is finished, construction of the enumeration writer procedure is easily completed. A procedure declaration is constructed using *MakeProcedureDecl* from three components: its name, its parameters and its block. The name of the procedure is simply constructed by adding the string `'Write'` to the name of the enumerated type. The parameter list consists of the two parameters for the output file and the enumeration value, each in their own parameter section. The block of the procedure is constructed by specifying that the label, constant, type, variable and subprogram declaration parts are all empty and that the statement list of the block simply comprises a single case statement whose case expression is the enumeration value parameter and whose case-clause list is that constructed in the loop.

Readers for enumerated types are considerably more complex than writers. Fortunately, however, much of the code used for an enumeration reader does not vary between applications. Examining Figure 2-3, one can see that the changing the reader subprogram for a different enumerated type requires changes to only four components of the subprogram, i.e.,

1. the name of the subprogram,
2. the type name of the third parameter of its parameter list,
3. the set expression in the if-statement which is the final statement of the reader, and
4. the case-clause list which is also part of the final if-statement.

Thus, a template for enumeration readers can be set up with dummy "parameters" at these four locations. These dummy parameters are just arbitrary syntactic objects that can be replaced with the appropriate type-dependent code; the *ReadColors* routine itself could be used as a template. Using such a template, then, the metaprogram which generates enumeration readers need only construct the appropriate actual objects to replace these dummy parameters.

The interaction between the template for enumeration readers and the metaprogram which specializes it consists of three phases: reading in the template, selecting certain syntactic components from it, and replacing the template parameters with the appropriate application-specific code. These three activities are easily identified in the *MakeEnumerationReader* metaprogram as shown in Figure 3-2. The

```
function MakeEnumerationReader (enumTypeDecl : Node) : Node;
  var i : INTEGER; ch : CHAR;
      id : StringType; f : TEXT;
      EnumShell, EnumParm, IfStmt, CheckWordFunction, CaseClauses,
        SetElements, Stmt, ids : Node;
begin
  reset(f, 'ENUMSHELL ');
  EnumShell := ParseFile(ProcedureDecl, f);
  EnumParm := MakeIdentifier(Str('x'));
  IfStmt := NthElement(BodyOf(BlockOf(EnumShell)), -1);
  CheckWordFunction := MakeIdentifier(Str('CheckWord'));
  CaseClauses := NullList(CaseClauseList);
  SetElements := NullList(SetElementList);
  ids := EnumerationConstantsOf(DefiningTypeOf(enumTypeDecl));
  for ch := 'a' to 'Z' do begin
    Stmt := AlternateOf(IfStmt);
    for i := 1 to Length(ids) do begin
      id := CoerceIdentifier(NthElement(ids, i));
      if NthCharacter(id, 1) = ch then
        Stmt :=
          MakeIfStatement(
            MakeFunctionCall(
              CheckWordFunction,
              List2(
                ExpressionList,
                MakeString(AddQuotes(id)),
                MakeInteger(MakeIntegerString(StringLength(id))))),
            MakeAssignment(EnumParm, MakeIdentifier(id)),
            Stmt);
    end;
    if IfStatementQ(Stmt) then begin
      Insert(SetElements, -1, MakeString(AddQuotes(ChString(ch))));
      Insert(
            CaseClauses,
            -1,
            MakeCaseClause(
              List1(ExpressionList, MakeString(AddQuotes(ChString(ch)))),
              Stmt))
    end
  end;
  Replace(
    VariableTypeOf(NthElement(ParametersOf(EnumShell), 3)),
    TypeNameOf(enumTypeDecl));
  Replace(Operand2Of(PredicateOf(IfStmt)), MakeSetFactor(SetElements));
  Replace(ClausesOf(ConsequentOf(IfStmt)), CaseClauses);
  Replace(
    NameOf(EnumShell),
    MakeIdentifier(
      AppendStrings(
        Str('Read'), CoerceIdentifier(TypeNameOf(enumTypeDecl)))));
  MakeEnumerationReader := EnumShell
end;
```

Figure 3-2: Metaprogram to Construct Enumeration Readers

ParseFile procedure is initially used to read in the template from a file named ENUMSHELL. The if-statement which is the final statement of the procedure is then selected and assigned to the *IfStmt* variable. As shown in Figure 2-3, note that the else part of this if-statement is the following assignment statement.

```
success := FALSE
```

In the body of the loop, this component is repeatedly selected (using *AlternateOf*) and used in the

construction of case clauses. The final interaction with the template occurs at the end of the metaprogram where four `Replace` statements replace the four dummy parameters with the appropriately constructed code for the given enumerated type.

Aside from interaction with the program template, the remainder of the metaprogram constructs program components in a fashion similar to the `MakeEnumerationWriter` metaprogram. The main loop of this metaprogram is somewhat more complex, however, in order to group the processing of enumeration identifiers with the same initial letter into a single case clause. For each letter which begins one or more enumeration constants, a nested if-statement is constructed with branches for each of those constants and a final else branch which is the failure statement assigning `FALSE` to `success`. The statements so constructed are then accumulated into the `CaseClauses` list being built, and the corresponding letters are accumulated into the `SetElements` list as character constants. When the loop is finished, these values are then used in performing the parameter replacements; the modified template is then returned as the result of the `MakeEnumerationReader` subprogram.

Although these metaprograms seem somewhat lengthy, their design is actually quite straightforward. The biggest problem is to get the details of the metaprogramming routines right. Since the metaprogramming routines are defined in a grammar-based fashion using a GRAMPS grammar [2], however, this simply requires that a reference copy of the grammar be kept close at hand.

4. Discussion

The enumeration I/O example illustrates the most direct application of metaprogramming techniques to the design and construction of reusable software units. In this scenario, each member of a library of reusable software units potentially consists of both a program template and a metaprogram to specialize the template. Some library components will consist of a metaprogram only, as in the enumeration writers example. Other, more conventional, components will consist of a program template only; the templates in such cases will be directly usable without any source-level parameter substitution.

Another way of using metaprogramming to achieve reusable software is to build metaprograms capable of *in-line coding* a given set of abstractions. Suppose, for example, a set of complex arithmetic routines is to be provided as a reusable abstraction. The conventional approach would be to implement them through a subprogram library using, say, a cartesian representation. Alternatively, the routines could be implemented through a metaprogram that was capable of processing application programs and replacing all calls to complex arithmetic routines by equivalent in-line code. Such an in-line coder would eliminate the overhead of procedure calls to the complex arithmetic routines (which would be relatively high in proportion to the computation typically performed in such a routine) and also carry out various optimizations using properties of complex arithmetic. An inline-coder for complex arithmetic that was constructed along these lines was found to improve run-time performance substantially. Another application of this approach is an in-line coder for the routines that implement the metaprogramming system itself; such an in-line coder metametaprogram would not only improve efficiency of target metaprograms but could also be designed to carry out various correctness checks on them.

Metaprogramming systems can also be used to support the reuse of existing software. Consider the implementation of a new version of a large software system using new data structures. It may be that various parts of the old version of the system could be reused in the new system provided that certain systematic changes are carried out. Unfortunately, the required changes are likely to be expressed in syntactic and semantic terms; carrying out such changes using a combination of text-based and manual methods is likely to be too tedious and error-prone to be worthwhile. On the other hand, it may well be feasible to construct a metaprogram to carry out the changes; errors in the final result can be corrected by amending the metaprogram and reapplying it to the original software [10].

Another reuse problem for existing software is when a new implementation of the software is required in a different programming language. Metaprogramming systems can be applied here by configuring a system for two target languages and implementing source-to-source translation tools. We have some positive preliminary experience with a 600-line Pascal-to-Modula translation metaprogram that seems to

carry out about 90% of the translation work needed for typical programs.

Our research has been primarily concerned with using metaprogramming systems as *metatools*, i.e., facilities for constructing software tools in general and integrated programming environments in particular. The direct use of metaprogramming systems themselves to support the reuse and maintenance of software is thus more-or-less incidental to our main emphasis. Nevertheless, the application of metaprogramming systems in these areas seems to have an interesting potential worthy of evaluation in an industrial setting.

References

1. *Reference Manual for the Ada Programming Language.* United States Department of Defense, 1983.

2. Cameron, R. D., and Ito, M. R. "Grammar-based definition of metaprogramming systems". *ACM Transactions on Programming Languages and Systems 6*, 1 (January 1984), 20-54.

3. Cameron, R. D. *Multi MPS Reference Manual.* School of Computing Science, Simon Fraser University, 1987.

4. Goguen, Joseph A. "Parameterized programming". *IEEE Transactions on Software Engineering SE-10*, 5 (September 1984), 528-543.

5. Joy, William N., Graham, Susan L., Haley, Charles B., McKusick, Marshall Kirk, and Kessler, Peter B.. *Berkeley Pascal User's Manual Version 3.0 - July 1983.* University of California, Berkeley, California, 1983.

6. Kernighan, Brian W., and Richie, Dennis M.. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

7. Merks, E.A.T. Compilation Using Multiple Source-to-Source Stages. Master Th., Simon Fraser University, School of Computing Science,April 1987.

8. Neighbors, James M. "The Draco approach to constructing software from reusable components". *IEEE Transactions on Software Engineering SE-10*, 5 (September 1984), 564-574.

9. Ruiz-Huerat, Gabriel. "The programmable compiler". *Computer 16*, 3 (March 1983), 35-39.

10. Terry, B. W., and Cameron, R. D. Software maintenance using metaprogramming systems. 1987 Conference on Software Maintenance, September, 1987.

An Object-Oriented Kernel For Composite Objects

Ambrish Vashishta and Satyendra P. Rana
Department of Computer Science
Wayne State University
Detroit, MI 48202.

Abstract

By utilizing existing pieces of tested code, software quality and productivity can be enhanced. To reuse software, it must be stored in a database for later retrieval during the new software development. For this purpose, we model software entities as composite objects and employ object-oriented approach to support the concept of software reusability. A data model appropriate for software entities is described and the implementation of a corresponding kernel is outlined. The object-oriented kernel provides facilities for routine management of software objects, definition of composite objects, and version support for managing updates of existing software.

BIOGRAPHIC INFORMATION

Ambrish Vashishta received the B.Sc. degree in Mathematics, Physics, and Chemistry from Agra University, Agra, India, in 1968, the M.Sc., and M.Phil. degrees in Mathematics from Aligarh University, Aligarh, India, in 1970 and 1972 respectively, the M.Sc. degree in Computer Science from St. Andrews University, St. Andrews, Scotland, in 1978, and Ph.D. degree in Computer Science from the University of Aberdeen, Aberdeen, Scotland, in 1982. From 1982 to 1984 he was a post-doctoral fellow at the University of Aberdeen.

He is currently an Assistant Professor in the Department of Computer Science, Wayne State University, Detroit, MI.

Dr. Vashishta is a member of the ACM and the IEEE Computer Society.

Address:
Department of Computer Science
Wayne State University
Detroit, MI 48202

Telephone:
(313) 577-3201/2477

Satyendra P. Rana is an Assistant Professor in the Department of Computer Science at Wayne State University. His areas of interest include cooperative computing, object-oriented design of distributing systems and operating systems.

Dr. Rana is a member of the ACM and the IEEE Computer Society.

Address:
Department of Computer Science
Wayne State University
Detroit, MI 48202

Telephone:
(313) 577-2831/2477

1. Introduction

Software development is time consuming, expensive, and labor intensive. The reason is that software tools and methodologies have not improved as rapidly as the needs for large and complex software. Recent studies [BERN84, CARD85, HORO84, NOMU87] indicate that today's tools for requirements specification, design and development are still in their infancy.

An emerging concept for enhancing software productivity is to utilize existing pieces of software in a newer context or application. This is known as software reusability. A basic premise of software reusability is that potential users gain use of the functionality, transportability and operational environment of the existing software. Further, the software must be usable as a building block in the construction of a larger software system much in the same way as one would use small data objects in the specification of larger data objects. Merits of the concepts of software reusability and version aspects of it are discussed in [HORO84].

Reusability offers the potential for increased productivity and quality in many stages in the life cycle of software development and maintenance [JONE87]. Examples of potentially reusable entities are data, program, module, applications, design etc. Reusability to a limited extent is possible in the traditional software in various ways. A library of subroutines is an example of such a restricted form of reusability. Here the objective is to provide a global access to pieces of software. Another distinct approach is to enhance the applicability of a routine by making it general purpose and parametric or generic. Designing a reusable or generic code by itself is not enough to exploit reusability. Additional support for retrieval and composition of reusable components is required. It is this support for reusability that we are concerned with, in this paper.

We adopt the object-oriented approach and model software entities like program, module, packages etc. as objects. This allows us to apply methods (procedures) to software objects. The construction and inheritance of such methods in essence allows us to build the requisite support for exploiting reusability to the maximum.

Our approach to software reusability is to employ a software management system which can store pieces of software as objects. Objects in a software database can be searched and manipulated to meet certain user requirements. The user requirements that can be obtained through a software development system can translate software design into code requirements.

In this paper, we describe a data model to store and

manipulate software pieces in an object-oriented manner. The kernel implementation describes how objects are stored in classes and manipulated. Also explained is a version management that takes into consideration the maximum data sharing among all versions. Finally, concluding remarks and extensions to this work are outlined.

2. The data Model

Entities in a database are modeled by defining the data structures, operations and integrity assertions [DATE86]. Data structures are manipulated in a value-oriented or an object-oriented manner. In a value-oriented approach, operations are performed on copies of data, while in an object-oriented approach information about data representation is hidden and data values are accessed and manipulated by invoking available methods (the property of encapsulation).

2.1 Basic Definitions

The two virtues of object-oriented paradigm are encapsulation and inheritance. An object is a block of some private data and a set of operations that can access that data [COX86]. An object is represented by a collection of instance variables which draw their values from different domains. Values in a domain correspond to a data type in a programming language. An object is uniquely identified, among other objects, by a pointer (or a handle), called an object identifier (OID). Thus an object's OID points to the block of its data.

The operations associated with an object capture the behavior of the object and are referred to as methods. Methods are invoked by messages which provide the semantic definition of the object. Each method is a piece of code which carries out the task to be performed. Instance variables and methods together constitute an object's definition. The state of an object is determined by the state of the instance variables which are represented by the private data of the object.

Objects with identical definitions are grouped together to form a class. For example, Figure 1 shows a class 'department' which groups together all department objects. Each department has its data values stored in instance variables : 'Name', 'Chairman', 'Number_of_employees', and 'Names_of_projects'. The methods available to the instances of the class 'Department' are defined as 'Cost_analysis' and 'Annual_report'. The common definition becomes the class definition and is shared by each member of the class. The members of a class are called instances. Objects in a class have the same properties e.g. instance variables and methods, however the private data of each object is distinct.

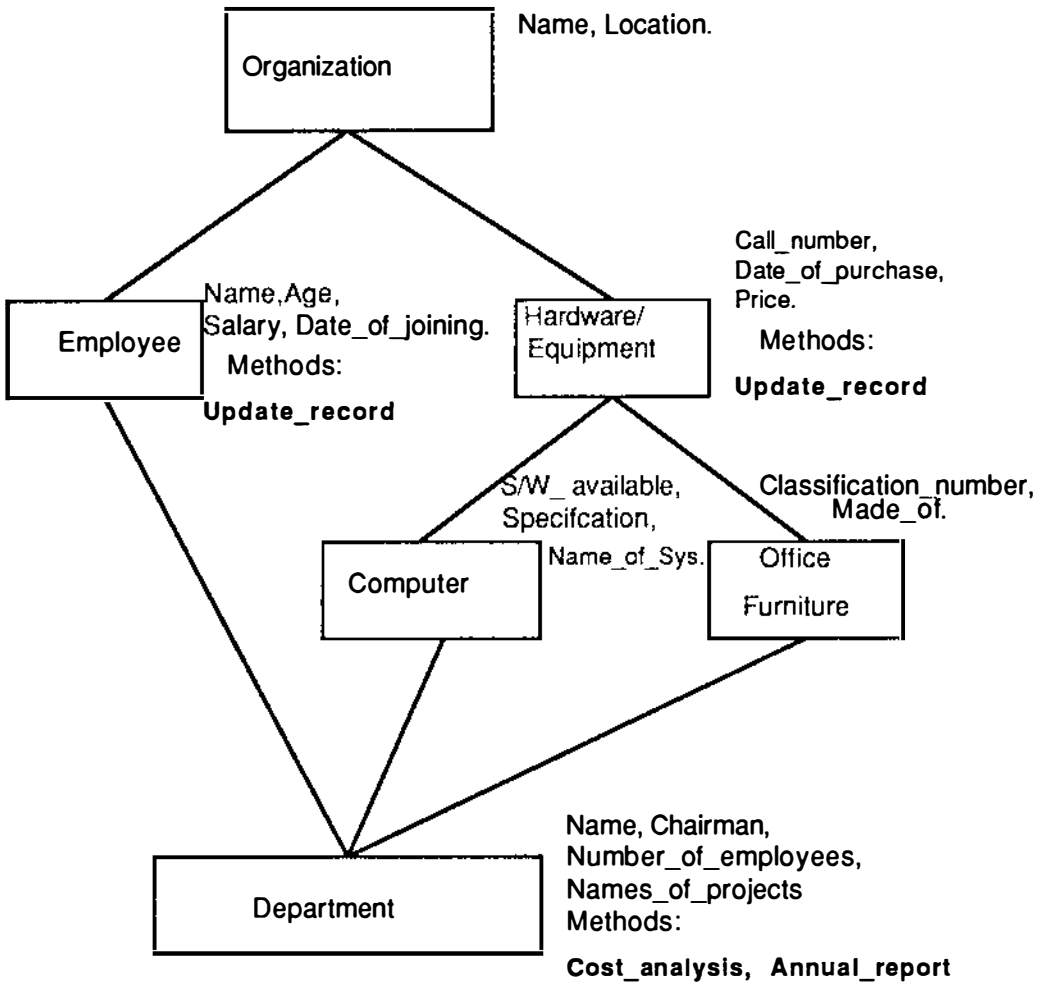


Figure 1 : Class hierarchy among data objects.

A class may inherit properties from other classes as well as have additional properties of its own. In Figure 1, the class 'Department' inherit properties from the classes 'Employee', 'Computer', and 'Office_furniture'. In a graphical representation, the classes are represented by nodes and the inheritance relationship between classes is represented by a directed edge. Such a graph induces a class hierarchy among the classes. A class is a superclass of all classes to which it is connected by a directed edge pointing away from it. Similarly, a class is a subclass of all classes to which it is connected by a directed edge pointing towards it.

Our data model has a predefined class called system class (SC). In the class hierarchy, the SC is at the highest level. All other classes are subclasses of the SC directly or indirectly. For example in Figure 1, the class 'Organization' is a superclass of the class 'Employee', and the class 'Employee' is a superclass of the class 'Department', implies that 'Employee' and 'Department' are direct and indirect subclasses of the class 'Organization' respectively. If a class is defined with no superclass then SC is its superclass by default. It is apparent that all classes belong to a single class hierarchy with SC at the top. It should be noted that the system class SC provides certain methods which are inherited by all classes. There are instance variables in SC used for integrity and version management of software objects. These instance variables are inherited by all classes, but their data values remain transparent to the user. The SC also provides primitive types such as integer and strings. Primitive types have values which are called primitive objects.

2.2 Composite Objects

Objects can be composite. A large software system consists of a collection of self-contained pieces of programs, for example, modules and subroutines. Programs have subprograms in them and subprograms have other subprograms and so on. Thus a large software system is a composite object which contains other objects as its subobjects. Subobjects in turn can be composite.

Our data model provides support for composite objects - the objects which are made up of instances from various classes. Instance variables of a composite object contain data as either a value from a domain or a reference to another object (a subobject). A reference is an object identifier (OID) of a subobject which is a component of the composite object. There may exist more than one instance variable in a composite object as references to different subobjects, therefore allowing a composite object (superobject) to have more than one subobject. Subobjects can in turn be composite object themselves and have instance variables which are references (OIDs) of other subobjects.

Program

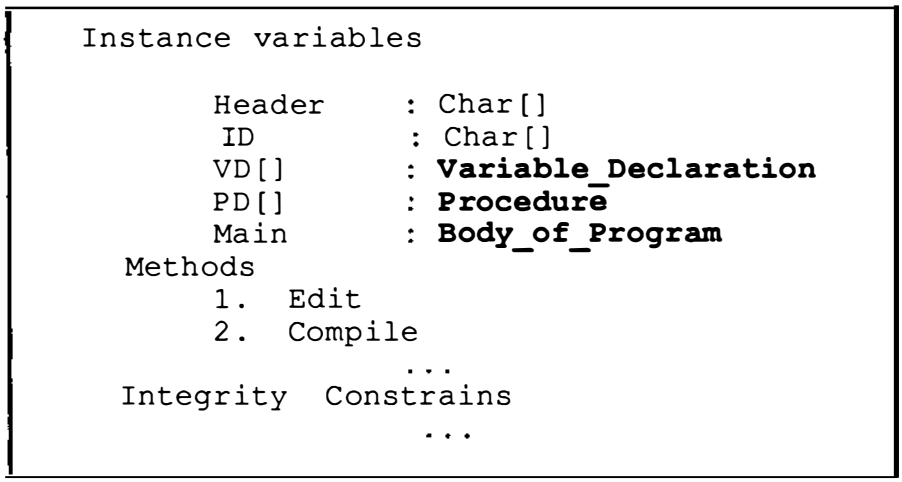


Figure 2 : A Program as composite object

Figure 2 illustrates a program as a composite object which is defined in terms of instance variables the Header, ID, VD, PD, and Main. The square brackets following VD and PD represent multiple occurance of objects. In this example, the instance variables VD, PD, and Main are defined as subobjects of the program and may have subobjects of their own. The subobjects Variable_Declaration, Procedure, and Body_of_Program are assumed to be defined as objects elsewhere.

When assigning the values for instance variables, the Header is initialized with 'program' and the ID with "name" of the program. Other instance variables are references to subobjects of the composite object 'program'. The object 'program' can be manipulated by messages that will invoke methods such as edit, and compile. The integrity constraints are not necessary in the defintion of all objects but can be specified if there are any.

Superobjects and subobjects in the construction of a composite object can be viewed in a hierarchy - the composite object hierarchy, in which the root represents the composite object, the leaves represent instance variables, and all other nodes as subobjects.

2.3 Integrity of Composite Objects

As mentioned above a composite object contains instances from various classes. At the time of creation of composite objects the data model ensures that the integrity of such objects is maintained. Instance variables that are subobjects, and represented by OIDs of instances, are verified for existence in their classes. Subobjects are not given default values, and therefore all subobjects of a composite object must exist before the root can have a representation in the software database. Primitive objects can be given default values.

2.4 Version Support

Objects can evolve over time. This concept requires support in the data model to capture changes in objects. Changes can be managed if all previous copies of objects are maintained. Similar requirements have been mentioned for office applications [ZDON84], Computer-Aided-Design (CAD) [KATZ86], database applications [BERN84], programming environments [PERR87], and families of large programs [WINK87]. Such requirements are to be incorported in the data model as part of version support.

Our data model provides version support for objects so that the users can exercise control over the version histories. We provide a version management facility at two levels- the object definition level and the command level. When an object is defined, it can be specified whether all objects in its class will have versions or not. This allows uniform treatment of version mechanism to all objects of individual classes. If, in the object definition, it is specified that the objects in a class have versions, any changes in the state of objects in that class will be preserved.

It is possible that not all objects in a class would require version support. This can be handled in our data model through the user-interface. The user-interface provides a version mechanism for individual objects. Individual objects can be activated to resume versions or vice versa, that is, if an object is currently maintaining versions, its version activity can be suspended and all previous versions are dropped except the most recent one pointed to by the currency indicator (term also used in [KATZ86]). Among many versions of an object the user can designate a version to be the current version (by setting the currency indicator). The currency indicator may not necessarily point to the most recent or up-to-date version. Whenever an object is accessed the version pointed by the currency indicator is made available automatically, unless a specific version is requested by the user. Provisions exist in the user-interface for requesting specific versions of objects (explained below). The currency indicator has the precedence over the most recent version.

Figure 3(a) illustrates a class of six objects where three objects identified by OID1, OID2, and OID3 have been activated to maintain versions. Object OID2 has two versions identified by version identifiers V2.0 and V2.1 where V2.0 is the original object and V2.1 is its version. Version activity of any of these OIDs can be changed. In Figure 3(b), object identified by OID2 has terminated its version activity while OID6 has resumed its version activity and has V6.1 as a new version. OID7 and OID8 are new insertions in the class. It is assumed that this class definition specifies that new insertions do not maintain versions

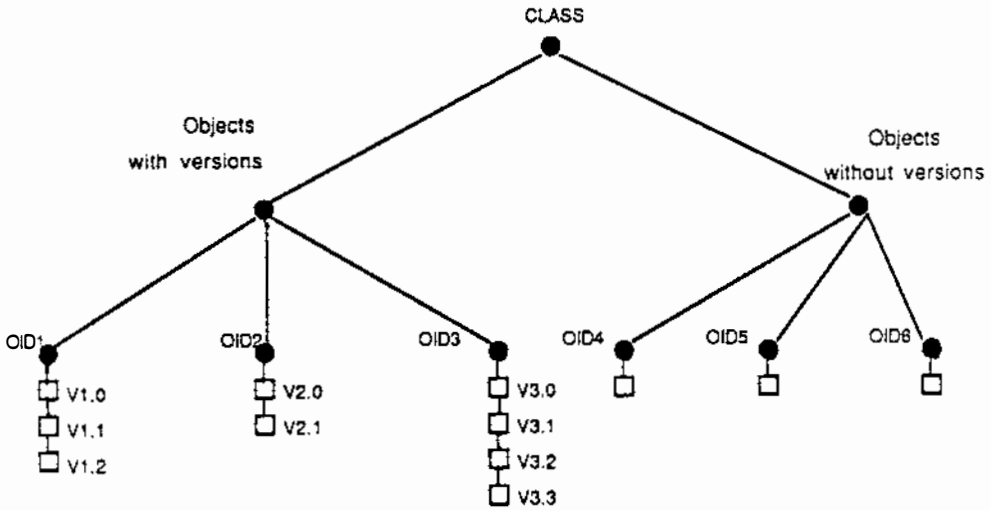


Figure 3(a): A view of versions in a class

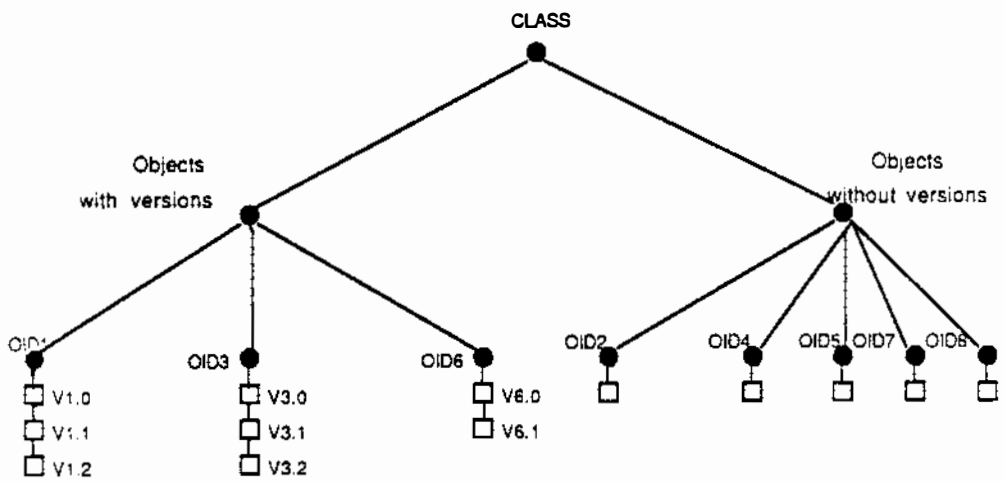


Figure 3(b): View updates of versions

unless individually activated. The reverse is also possible in a class definition which may specify objects to maintain versions unless individually inactivated for this activity.

As individual objects have access to the version mechanism through the user-interface, classes can also be activated or inactivated for the version mechanism through the user-interface, allowing classes to alter their version characteristics altogether.

2.4.1 Version Identifiers

Within each version set an object is identified by a unique version-identifier (VID). A VID is a sequence number. In order to select a particular version the complete list of VIDs is available for consultation. The VID-list provides many options. It can be chronological (update sequence), reverse chronological, forward sequence (in which the versions were originally created), and backward sequence. One of the VID in the list is a currency indicator. A new version may be created from the currency indicator or any other VIDs in the version-list. Thus a sibling of any version can be generated. This creates version-hierarchy of objects (VID-list can be viewed to show parent-child relationships). Figure (a) shows the version hierarchy where the currency indicator is set on V2. A typical sequence of steps to manipulate a version is: inspect the VID-list; select a VID; set currency indicator (if it is not already assigned to it); make modifications using this VID, remove this VID, or create a new version from this VID.

3. The Kernel Implementation

The kernel is the central component of the software management system and responds to the commands that it receives through the command interface. A user request submitted in the form of a query is decomposed into several individual commands that are executed by the kernel. The command executions make the kernel interact with the software database (SDB)(see Figure 4). There are three logical components that constitute the kernel; namely, the object manager (OM), the storage manager (SM), and the index manager (IM).

The kernel accepts a user request through command-interface. The object manager (OM) is responsible for identifying and interpreting commands which are decomposed into various services that are available from the storage manager (SM) and index manager (IM). The storage manager handles storage related functions such as memory management, database I/O, maintenance of database files and provides mapping between classes and file segments. One important task of the storage manager is to create and maintain system directories (other than the indexes) that are frequently accessed in command executions. The index

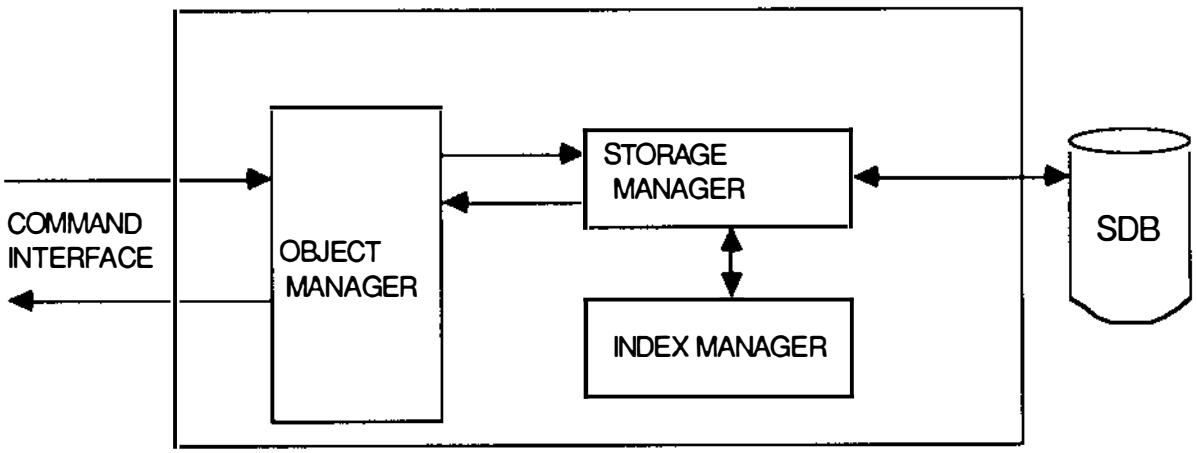


Figure 4: The kernel process structure.

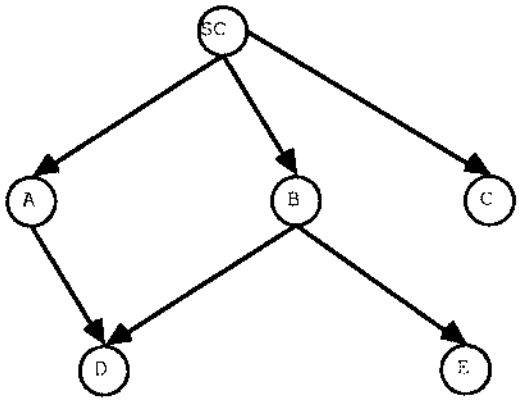


Figure 5(a): Object Inheritance Profile (OIP)

	SC	A	B	C	D	E
SC	0	1	1	1	0	0
A	0	0	0	0	1	0
B	0	0	0	0	1	1
C	0	0	0	0	0	0
D	0	0	0	0	0	0
E	0	0	0	0	0	0

Figure 5(b): Adjacency matrix of Figure 5(a).

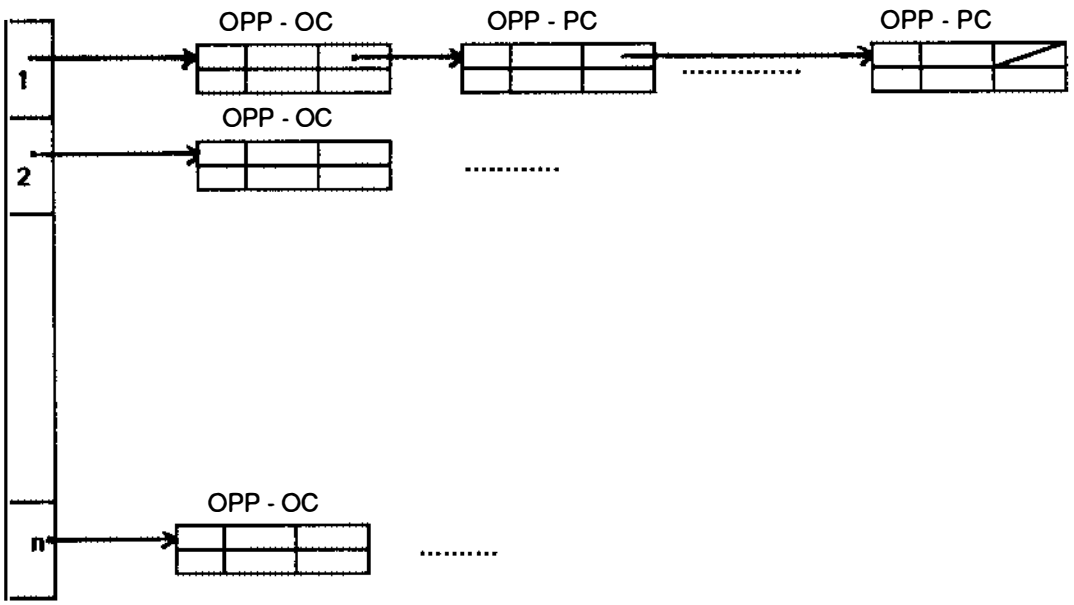


Figure 5(c): Object Property Profile (OPP)

manager (IM), creates and maintains indexes of OIDs and VIDs, and provides support for the version management.

3.1 Object Manager (OM)

The commands received by the object manager (OM) are categorized into two groups : commands which do, and those which do not alter the database state.

The OM receives a new object definition and it processes the definition by adding this object in the class hierarchy. The structure that implements the class hierarchy is known, to the kernel, as object property inheritance profile (OIP), which is a directed graph (Figures 5(a) & 5(b)) . The OIP is a list structure in the OM workspace and is saved in the form of an adjacency matrix in the database [SING85]. Semantic information of software objects is stored permanently in the software database (SDB). A control file (CF), part of the database, stores essential database structures that manage software database from one user-session to another. When an instance of a class is stored in the database, the OIP is responsible for identifying the object class and to provide instance variable names for the user to specify data values for the private data of the object (Figure 5(c)). All objects in the software database are allocated unique OID's by the system by which they can be retrieved.

In multiple inheritance, superclasses are ordered with respect to individual subclasses. This ensures instances in the subclass to have identical ordering of instance variables. Leftmost superclass is the first in sequence to provide its properties then the second from the left, and so on.

Object manager validates that all classes are uniquely named. It also checks that instance variables within classes are differently named. Conflict arises if different owners happen to choose identical names for instance variables for two different classes. The OM resolves this by associating instance variables to their objects. For example, if an object A has instance variables I_1, I_2, and I_3, then $A.I_1$, $A.I_2$, and $A.I_3$ will be their unique instance variable names when conflict arises.

Another structure used by the OM, to store properties of objects, is called object property profile (OPP), which is an array of list-structures. An individual array element in the OPP points to an object's properties. The OPP is responsible for providing properties of existing objects when definitions of new objects are added in the class hierarchy.

Each element in the array of OPP points to the object-name (OPP-object-cell) and a list of properties (OPP-property-cells). OPP-OC contains details on an object such as object name, number of instance variables, and instance size. An OPP-PC stores

information on an instance variable such instance variable name, its primitive type, size, etc.

Methods inherited by all subclasses from the system class (SC) are described below :

Create : creates an instance of a given class in the user workspace (the user provides the data values of instance variables of the object).

Store : stores an instance from the user workspace in its class.

Get : locates an instance of a given class. (The OID of the instance is returned which is used for fetching the instance.)

Fetch : retrieves an instance in the user workspace through an OID.

Delete : deletes an instance of a given OID.

Modify : updates data values of a given instance.

Select : retrieves instances of a class that satisfy a given predicate.

3.2 Integrity of Superobjects:

A subobject can not be deleted if it remains part of some composite object. When deleting an object, the user does not need to know whether the object is a subobject of a composite object. Deletions are performed in the following manner. Every object maintains a transparent value, called a counter, indicating the number of superobjects in which it participates. As long as the counter remains greater than zero the object is never physically removed. When a superobject is deleted the counter in all its leaf objects are decremented by one. Any object with counter as zero can be physically deleted. When an object is initially created the counter is set to zero. The counter is an instance variable of the system class (SC). The inheritance of this instance variable throughout the class hierarchy is part of the integrity management of objects.

3.3 Version Management

The class SC, has a predefined "time" instance variable which is used in version management of objects. The following commands can be used to access and manipulate versions:

Version-History (OID1) : This command accepts an OID1 and

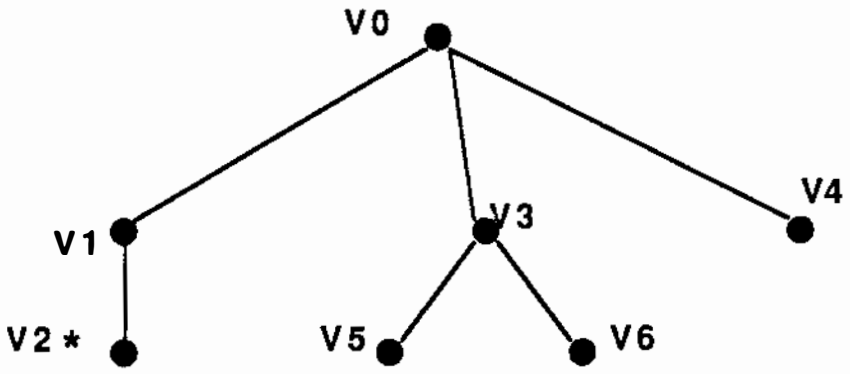


Figure 6(a): Version history

OID1	V0	V1	V2*	V3	V4	V5	V6

Figure 6(b): Version list

Figure 6(c): VID, time sequence, and page segment list of versions

V0	T4	t1 PS1	PS2	PS10	
V1	T2	PS4	PS5		
V2*	T3	PS6	PS7	PS8	PS9
V3	T7	t5 PS16	PS17		
V4	T6	PS13	PS14	PS15	
V5	T8	PS18	PS19	PS20	
V6	T9	PS21	PS22	PS23	

V0: has been updated after V2's and before V3's creation.

V4 has been created before V3 was updated.

PSi is a triplet (page-number, offset, page-segment-size

All PSi's in an object's versions are unique.

After an update in a version new PSi's are assigned to it, for example in versions V0 and V3.

displays the version hierarchy for OID1. As shown below, the VID as V_0 has generated versions with VIDsas V1, V3, V4. V4 has not generated any of its versions. The currency indicator is indicated by the Symbol "*". V_0 is the root of all versions (Figure 6(a)).

V_0 : (V_1, V_3, V_4);
V_1 : (*V_2);
V_3 : (V_5, V_6);
V_4 : ();
V_5 : ();
V_6 : ();

GET (OID1,VID1) : accesses the version identified by VID1 for the object of identifier OID1.

REPOSITION (OID1,VID1) : moves the currency-indicator to VID1 for object of OID1.

NEWEST_VERSION (OID1) : provides the VID of the newest version of a given object identifier OID1.

CREATE_VERSION (OID1,VID1) : creates a new version of version identifier VID1 and connects the new version to VID1. The created version is allocated a new VID by which it can be retrieved.

DELETE (OID1,VID1) : deletes the version identified by VID1 in the version hierarchy of object identified by OID1.

The implementation of version management is targeted at minimizing data replications and maximizing data sharing among versions. This is achieved by storing changes in data that take place when a version is created and then subsequently updated.

The kernel treates versions as objects. Objects are physically stored in data pages of the software database. A data page is logically divided into one or many page segments of variable lengths. A page segment (PS) is represented by a triplet : page number, offset (within the page), and segment size. A list

of page segments (PSs) identifies a particular version. In Figure 6 (c), V0 and V1 are represented by lists of page segments (PS1, PS2, PS10), and (PS4, PS5) respectively. The version sequence and the version hierarchy of OID1 are shown in Figure 6(a) where the currency is set to V2. Figure 6(b) shows OID1 and its version list; objects with versions maintain a version list. In Figure 6(c), the second column indicates the time sequence of version updates. Page segment numbers are allocated in sequence and share data among themselves.

3.4 Storage Manager

A software database is a collection of files. Files are logically divided into file segments, each of which is designated to be a class (class segment), an index (index segment) or a directory (directory segment). A segment is a collection of consecutive pages.

The storage manager (SM) deals with large and small objects of fixed or variable lengths. An object which occupies more than a page is treated as a large object, otherwise it is considered a small object. A large object may spread over contiguous or non-contiguous pages within a file segment due to subsequent updates.

The SM maintains an arbitrary number of pages in its workspace, which can be from any of the three kinds of segments. This workspace is used to bring in the main memory the data objects, an index or directories such as, OIP, OPP, and VID-lists. When index pages are requested by the Index Manager (IM), the SM issues an I/O to bring these pages from the database. We make use of existing techniques for buffer management [EFFE84, SACC86] to make optimum use of the workspace. Our next step is to provide a combination of various strategies which can be automatically invoked by changing conditions in the system.

3.5 Index Manager

The index manager (IM) creates and maintains indexes on OIDs and VIDs. The IM allocates an OID for every new object inserted in the software database. An OID determines the address of an object in its class (the file segment). For data placement, a hashing technique is employed to allocate OIDs for clustering and for fast retrieval of objects [DEEN82]. We have improved upon the data placement and retrieval technique described in [DEEN82] to cater for large and variable length objects and the management of their versions.

When an object resumes its versions, its OID is reorganized and assigned to its version table which in turns addresses to individual versions of the object. The reverse happens when an

object suspends its version activity, in which case the version table and all versions, except the currency, are deleted and the OID is assigned to its data (the currency).

3.6 Object Directory

As stated earlier, objects that already exist in the software database are utilized in construction of the superobjects and also in new object definitions that are inserted to the class hierarchy. The object directory contains information on the class hierarchy, names of super and subclasses of a node, properties and methods, and instance variables of objects. The object manager (OM) accepts directory commands through the user-interface in much the same way as the commands described in Sections 3.1 and 3.3.

4. Summary and Concluding Remarks

We have described a data model for reusability of software components. The kernel implementation allows users to define new classes of software objects, to create new instances, and to search objects which satisfy certain predicates.

Composite objects may be utilized in developing large software systems from existing objects in the software database. Such a composite object, however, can not be utilized in a straight forward manner since these contain reusable pieces of software and not a complete software system. Further additions to the code will be required to determine the exact calling sequence of these reusable components.

Currently, the kernel services are used by invoking its system functions as described in the previous sections. The version management is designed to minimize data replication, that is, whenever a version is created only the changes are stored in the database to reflect updates in the new version. The common data among versions is shared.

We plan to extend the work presented in this paper in the following directions:

- development of a query language to search objects, which will accept specifications for user code requirements.
- enhancement of existing system directions for viewing object definitions, version list, class size etc.
- development of a system log facility for tracing user activity between sessions.

Acknowledgements

We are thankful to William I. Grosky, Abad A. Shah, and Punkaj K. Jain for insightful comments on early versions of this paper. Thanks also to Tan Ping, Shetwai Seto, and Shi Nin Zhu for coding part of the software. The research was sponsored by the Institute of Manufacturing Research, Wayne State University.

References

[BERN84] Bernstein, P.A., "Database system support for software engineering", IEEE, Ninth Intl. Conf. on Software Engineering, March 1987, pp. 166-177.

[CARD85] Card, D.N., Page, G.T., and McGarry, F.E., "Criteria for software modularization", IEEE, Eighth Intl. Conf.on Software Engineering, August 1985, pp.372-377.

[COX86] Cox,B. J., "Object-oriented programming", Addison-Wesley Publishing company, 1986, (Chapter 4).

[DATE86] Date, C.J., "Introduction to database systems", Fourth Edition, Addison-Wesley, 1986.

[DEEN82] Deen, S.M., "An implementation of impure surrogates", Proc. Eighth Intl. Conf. on Very Large Data Bases, Maxico City, Sept. 1982, pp. 245-256.

[EMBL84] Embley, D.W., and Woodfield, S.N., "Can programmer's reuse software ?", IEEE Software, July 1987, pp. 52-59.

[HORO84] Horowitz, E., and Munson, J.B., "An expansive view of reusable software ?", IEEE Trans. on Software Engineering, Vol. SE.10, No.5, Sept. 1984, pp. 477-487.

[JONE84] Jones, T.C., "Reusability in programming: A survey of the state of the art:, IEEE Trans. on Software Engineering, Vol. SE.10, No.5, Sept. 1984, pp. 488-494.

[KATZ86] Katz, R.H., Chang, E., and Bhatega , R., "Version modeling concepts for computer-aided design databases", Conf. Proc. ACM-SIGMOD, 1986, pp.379-386.

[NOMU87] Nomura, J., "Use of software engineering tools in Japan", IEEE, Ninth Intl. Conf. on Software Engineering, March 1987, pp.263-269.

[PERR87] Perry, D.E., "Version control in the Inscape's environment", IEEE, Ninth Intl. Conf. on Software Engineering, March 1987, pp. 142-149.

[SING85] Singh, B., and Naps, T.L.,"Introduction to data structures" , Chapter 9, West Publishing Company, 1985.

[WINK87] Winkler, J.F.H.,"Version control in families of large programs", IEEE, Ninth Intl. Conf. on Software Engineering, March 1987, pp. 150-161.

[ZDON84] Zdonik,S.B., "Object management systems concepts", Proc. of 2nd ACM-SIGOA Conf. on Off. Inf. Syst., 1984, pp. 13-19.

Session 8

REUSABILITY: TODAY AND TOMORROW

"XRLIB: An Overview of the X Window System"
Frank Hall, Hewlett-Packard

"Technology Update: Object-Oriented Programming"
Alan Snyder, Hewlett-Packard
(No paper prepared)

XRLIB: AN X WINDOWS TOOLKIT

Frank Hall
Corvallis Workstation Operation
Engineering Systems Group
Hewlett-Packard

ABSTRACT

Xrlib is a professional quality user interface library, based on the X window system, which has been developed by HP and placed in the public domain. It provides a consistent, easy to use message-based programmatic interface that facilitates access from objective languages as well as from standard C. It provides an extensible set of standard user interface components including pop-up walking menus, panels, message boxes, scroll bars, title bars, various buttons, text entry (single or multi-line), raster entry (scrollable pixel map or bit map), raster selection, and static placement of text or raster. This paper reviews the features of Xrlib and its future directions.

XRLIB AND THE X-RAY USER INTERFACE PROJECT

Xrlib is the result of efforts that began about two years ago at the Corvallis Workstation Operation (CWO, then part of the Personal Software Division). During the development of software for the Integral PC, it became obvious to a team of engineers that in order to do applications with a high quality graphical user interface, a programmer's toolkit was needed to reduce development time and to promote consistency in look and feel across a set of applications.

The team set out to create such a toolkit, since none was available for HP-UX. After a few stops, starts, charter shifts and reorganizations, during which time the team's research continued, the CWO User Interface Management System (UIMS) project was formally launched to develop a user interface toolkit product for HP-UX. At about the same time, another CWO team working on windowing systems identified the X window system as a suitable future basis for HP Windows/9000. (X is a public-domain, network transparent, portable windowing system developed by MIT with funding from DEC and IBM. It is supported as a networked windowing standard by HP, DEC, and a consortium of other vendors.) The UIMS team decided to base the toolkit architecture closely on X, and code-named it X-ray. Xrlib is the first phase of that effort.

HP has donated to MIT, and thereby into the public domain, the source for Xrlib's code, documentation, and sample programs. They are now included on the X distribution tape from MIT. Before the donation, Xrlib was run through its verification suite on not only HP 9000/Series 300 workstations, but also on DEC and Sun workstations. Xrlib is also available as part of HP's X window system product.

One goal of the X-ray UIMS team has been to promote more rapid formation of advanced user interface applications for the Unix (TM of AT&T) workstation market by the pro-active development or sponsorship of user interface standards and productivity tools within the industry as well as within HP. The strategy is to first address current market needs including standard programming languages, and then to progress toward

full object-oriented programming.

While working with interested parties within the X community to define and develop a common X toolkit, HP plans to foster ongoing software development by enhancing and supporting Xrlib, eventually bridging it into whatever X toolkit standard that stabilizes.

OVERVIEW

Xrlib is a library of routines that allow the programmer to easily create the user interface portion of the application. At the lowest layer of the library are intrinsics that provide input coordination, resource management, and various support functions.

Field Editors

The middle layer consists of an extensible set of interactive gadgets, such as push buttons and scroll bars, which are called field editors in reference to their extension of the concept fields in a form. In MS Windows, they are referred to as controls.

Field editors allow the user to easily set options and cause action in the application. The type of field editor used to set a particular option is dependent on the option. For example, when setting text attributes, the user may wish to set both bold and underlining. In that case, the field editor will have to allow either or both of the options to be set. This type of field editor is called a "Check Box." But when selecting a font, only one can be selected at a time, so the font selection boxes must be mutually exclusive. This type of field editor is called a "Radio Button." In all there are 11 predefined field editors that come with the current Xrlib. In addition, custom field editors can be defined. The available editors, shown in Figure 1, are:

Titlebar	
Scrollbar	
Checkbox	
Radio Button	
Push Button	
Static Text	
Text Edit	(single line text entry)
Page Edit	(multi-line text entry)
Static Raster	
Raster Edit	(handles pixmaps or bitmaps)
Raster Select	

Dialog Layer

The highest layer of Xrlib consists of menus and active composites of field editors such as message boxes (dialog boxes) and panels (or forms).

The pop-up walking (SunView-style) menu is very easily programmed. A mock-up of a cascaded menu is shown in Figure 2.

Message boxes are used to prompt simple decisions from the user. A mock-up of a message box is shown in Figure 3.

Panels are one of the most powerful tools provided by Xrlib. A panel is a collection of field editors displayed in a window, which can be manipulated as a unit. Figure 4 shows

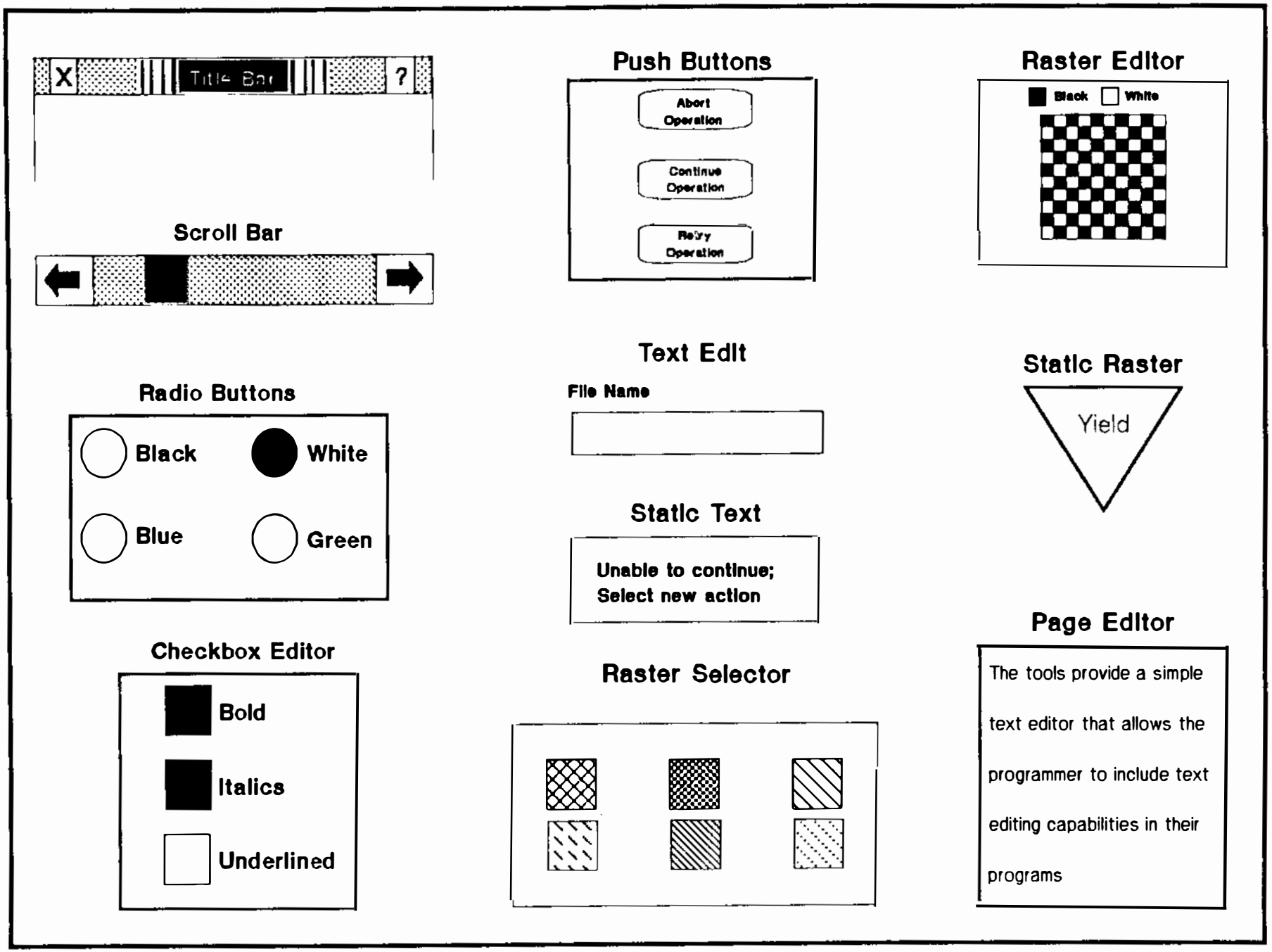


Figure 1. Field Editors.

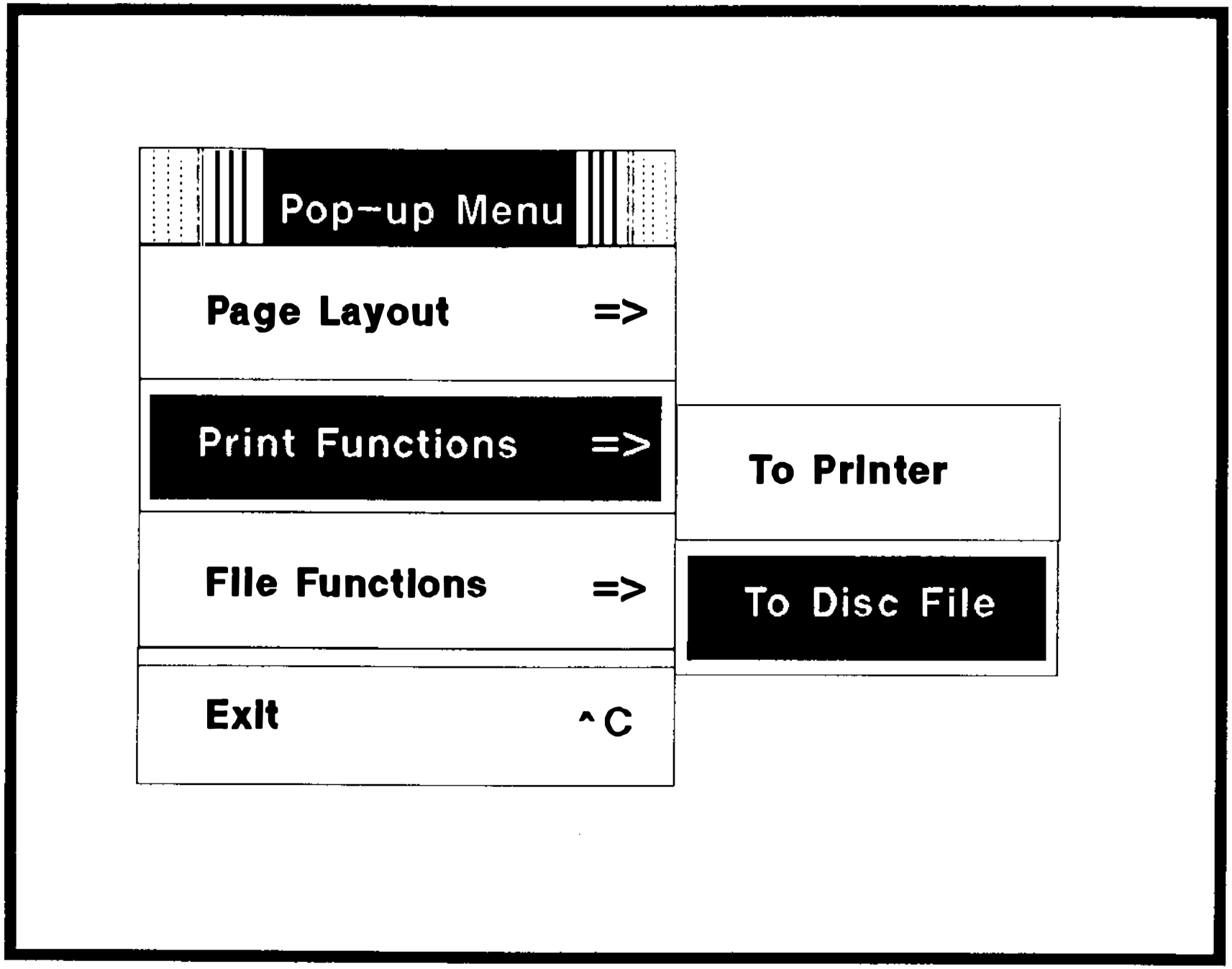


Figure 2. Pop-up Walking Menu.

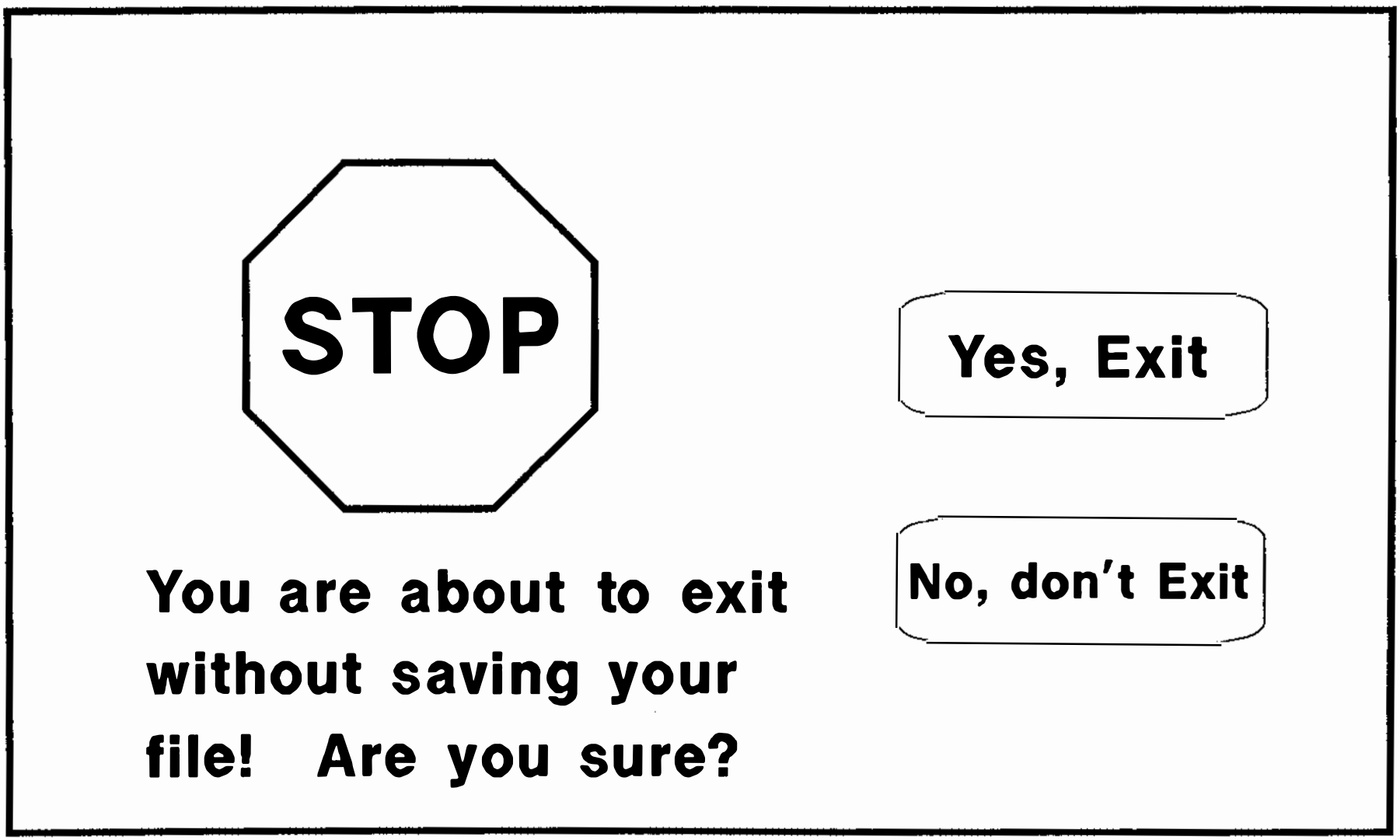


Figure 3. Message Box.

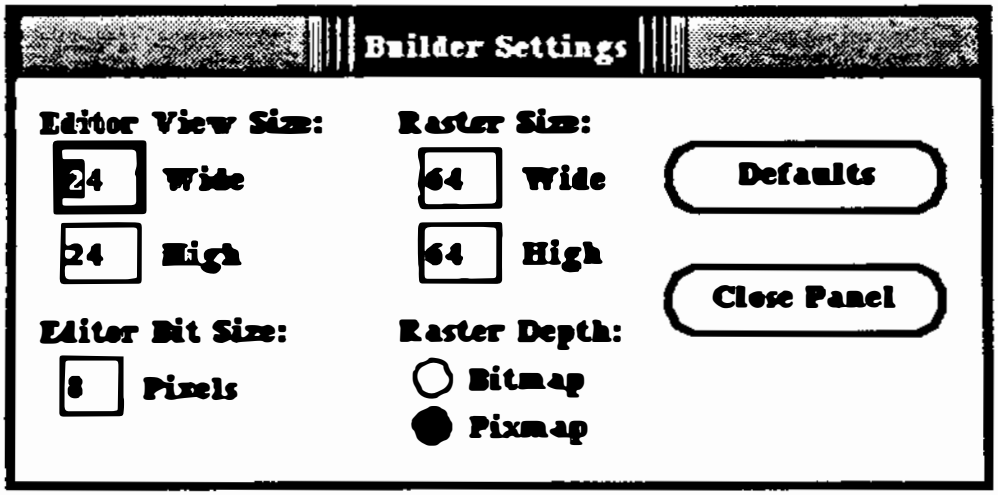


Figure 4. Sample Panel.

an actual panel taken from the Raster Builder that is provided as a sample program along with Xrlib.

DESIGN HIGHLIGHTS

Let me cover briefly some highlights of the Xrlib design.

Message Passing

In several toolkits, each interactive gadget is represented by a set of procedures, each of which does one thing to the gadget. This procedural approach was rejected by the X-ray team due to drawbacks found by our research, of which two are worth mentioning here.

First, the cluster of procedure names becomes unwieldy for the programmer and for the documentation. Each name in the cluster needs to be unique yet reflect the name of the gadget it helps represent. Names get longer than is convenient, and inevitably get abbreviated in inconsistent ways. Programming errors force irritating trips to the documentation. There, the cluster of procedure names, each with its own calling sequence, is difficult to document cleanly in one place such as a *man* page.

Secondly, the procedural approach does not allow the panel manager to properly handle arbitrary field editors. Since panels must be able to accept custom field editors as well as the standard ones, there is no convenient way for the panel manager to know which specific procedure to call for this specific field editor in order to do something to it, such as make it insensitive.

Instead, all field editors were designed to accept the following syntax:

XrScrollBar (instance, message, data);

Given the specific editor instance, the panel manager can invoke it with the desired message, in this case MSG_SETSTATE, to accomplish the operation without knowing what kind of field editor it is.

Note that this allows custom field editors to be included in panels along with the standard ones, for the panel manager can control it without specific knowledge about it.

This solved the above disadvantages and added some other advantages. Since at first we used the instance-message-data format in only some areas of Xrlib, we found that programmers using Xrlib made errors by trying to use it in other Xrlib routines where it could apply as well. It became clear that a worthwhile transfer of programmer learning was occurring within the toolkit, and the Xrlib design was modified to reinforce it.

A final advantage to note is that although this format is not true object-oriented programming, it sets the tone and prepares the programmer's style for a future shift into true object-oriented programming.

Input

The Xrlib input routine XrInput provides an important level of abstraction above XNextEvent. Input can be selected not only from the X queue but from an arbitrary set of sources, blocked or unblocked, with specified timeouts. Uninterpreted reads can be done for special purposes, but the application normally does a "hot" read which performs

useful preprocessing. In this mode field editors and specialized functions are activated by user actions without the application needing to get involved, and menus are handled transparently to the application. The application receives only those X events (and Xrlib pseudo-events) that are relevant to it.

Ease of Programming

The content of simple message boxes and menus can be specified easily by literals. For example, a non-hierarchical menu can be specified by an array of ASCII strings. Keyboard equivalents for power users can be added to menu items by simple syntax within these strings.

Portability

Xrlib ported easily to DEC and SUN workstations. Since the Xrlib code has no known Unix dependencies beyond the BSD *select* mechanism, it should be portable to any worthwhile multi-tasking operating system which supports X and C.

In particular, Xrlib should be portable to MPE and multi-tasking DOS, given an implementation of X on these systems.

Extensibility

Xrlib can be extended in terms of additional field editors and dialog managers. Some work is already underway within various entities inside HP which make use of some custom components, some of which may be added to the planned Xrlib set. Contributions from parties outside HP will be accepted through a process to be worked out with MIT.

COMMON QUESTIONS

These are some of the more common questions that are asked during presentations and discussions of Xrlib.

What toolkits did you include in your research?

The X-ray team spent two years of research on Xrlib. During that time we closely examined a number of leading tool kits, including Apollo's Dialog, Apple's MacIntosh, and MicroSoft's MSWindows. Our choice of field editors, and much of our look and feel, are based on MS Windows but with a much, much simpler programmatic interface. Our resource management is based on MacIntosh.

Because a toolkit must be closely matched to the underlying windowing primitives in order for programming with the toolkit to be comfortable and effective, a straight-out programmatic adaptation of existing toolkit designs onto the X system was not attractive. Now that X has emerged as a standard for the windowing primitives, the question remaining is what a toolkit needs to provide in order to closely match X, remove some of the tedium of programming in X, and actively advance toolkit technology.

What about the X Toolkit?

Xrlib is the toolkit we recommend customers and ISV's to use. It is here today and of high quality, and has been run through its verification suite on HP, DEC, and SUN workstations. It is the first phase of a well-researched toolkit strategy which we can complete so quickly that we cannot in good faith hold back while waiting for a "standard" to be defined. Because there are so many good ideas in this area, any "standard" will have to be well thought out, well-crafted, and well accepted by a variety of hardware and software vendors. The joint effort between MIT, HP, and DEC to define an X Toolkit is based on the concept that "widgets are windows", which is only one possible idea to follow. We based Xrlib on current customer needs, and we have solid plans to complete the Xrlib library and bridge onto the X Toolkit after it becomes defined, reviewed, implemented, documented, tested, and stable. Given the current lack of a formal specification, we think that process could easily take until this fall, possibly longer, to reach an ease of use and level of quality comparable to that of Xrlib.

We regard the X Toolkit as experimental at this point, and will provide it as such to our customers as it becomes available. However, we stress that the Unix workstation market sorely needs advanced user interface applications now, and its development should not be delayed indefinitely for a promised toolkit. We recommend that developers who have immediate needs to use Xrlib now, and to follow Xrlib onto the X Toolkit as it stabilizes.

Why did HP donate Xrlib into the public domain?

We get asked this and similar questions: Doesn't it help HP's competitors? What does HP get out of it?

The Engineering Systems Group divides the model of the technical workstation market into three sections. At the top are the applications, which ideally are supplied by independent software vendors (ISV's) and which provide a healthy, broadening range of solutions for Unix customers. We want to do all we can to increase the rate of growth of Unix workstation software, which up to now has been seriously constrained by lack of user interface standards and standard productivity toolkits. ISV's as well as volume end users (VEU's) are calling for standardization in this area.

On the bottom of the model is the hardware upon which the software runs. Customers have learned the hard way that it is a mistake to buy hardware from a vendor whose non-standardization will prevent the customer from easily switching over to another vendor if a more satisfactory solution comes along from someone else. We see more and more customers who have a heterogeneous installation of workstations from many vendors, and they want all of them to play together on the local area network as well as to port software easily among them. They encounter the same barrier as the ISV's: no standards in windowing or user interface libraries. They are calling strongly for standardization in this area.

In the middle of the model are the interfaces, which we call the platform, which allow the applications and the hardware to work together. Planks in the platform include the operating system, commands, networking, and windowing and user interface in general. It is at this "plankware" level of the market that standards must emerge to remove the barriers being encountered at the software and hardware levels.

HP is confident that we can compete effectively at the hardware level on the basis of performance, reliability, price, and support. We are also confident that we can compete with proprietary applications in terms of quality, functionality, human factors, and

support. But to do this we now recognize that we must work actively to promote standards at the platform level to remove the barriers that now impede software formation in Unix. This will not only allow Unix to compete more effectively with MSDOS in the workstation market, but we are confident that HP will sell more workstations and more applications if we can work effectively to promote standards at the platform level.

It is true that "standard is better than better." But HP wants to take that one step further. We want to work openly with the Unix community at the platform level to make good things standard, and standard things better.

Session 3

USER INTERFACE

"Quality Issues in Product Planning Work"
J. P. Delatore and E. M. Prell, AT&T Bell Labs

"Making Menu Interface Systems More User Friendly"
Barbara Beccue, Illinois State University

"Automated Testing of User Interfaces"
Mark A. Johnson, Mentor Graphics

Making Menu Interface Systems More User Friendly By Use of Human Factor Design and Prototyping

by
Dr. Barbara Beccue
Applied Computer Science Department
Illinois State University
Normal, Il. 61761
(309) 438-8338

Abstract

In many instances, the existing menu interfaces of systems can be enhanced to facilitate their use without a large expenditure of resources. The designer can apply concepts of human factor design in redesigning the menus. However, many professionals in the field of system development are not aware of guidelines for human factor design. By educating them about the guidelines, they can then apply the guidelines to redesign menus which are user friendly. In addition, the designer can prototype the menu for the user's comments and reaction. The use of these two techniques will enable the designer to create menus which better suit the needs of the user.

This paper discusses two examples of using these techniques. It also discusses some of the human factor design guidelines which were incorporated in each example. When these techniques are used during maintenance or system enhancement of existing systems, the menu portion of the system will usually be more user friendly. The changes seem to be primarily cosmetic and organizational, and therefore, have only minor impact on existing code. Consequently, relatively little of the information systems department's resources will be needed. However, changes made to the user interfaces result in the users viewing the system more favorably.

Biographic Information

Dr. Beccue is an associate professor in the Applied Computer Science Department at Illinois State University. She teaches graduate and undergraduate courses in the areas of information system development, human factors, quality assurance, and database. Her B.S., M.S., and PhD. are from the University of Illinois. Prior to teaching, she worked as a programmer and a system analyst in industry and in government.

Dr. Beccue has presented papers on Fourth Generation Tools and on Data Modeling at several conferences such as ACM SIGSCE, and DPMA ISECON. She co-authored a recent article, "Effective Communication with Users Improves Data Base Design", Data Management, 25:1, January, 1987.

Making Menu Interface Systems More User Friendly By Use of Human Factor Design and Prototyping

The widespread use of computers in all areas of business has caused a rapid increase in the number of people who are end users of computer application systems. Furthermore, the environment in which these people use computers and obtain information from them has changed. These end users are not just recipients of computer generated reports as the majority of them have been in the past. Instead they are actively involved in interfacing with the system to input data and to generate output or obtain information. Rapidly changing technology has provided the computing power to support this increasing end user involvement. It is important to recognize that the increasing involvement has two components: one component is the increase in the actual number of end users; the other is the increase in the amount of actual use of the computer. Consequently, people who use computer based application systems, end users, are having a significant voice in determining the acceptance and success of a system.

In attempting to create systems to meet this new demand, problems have arisen because system designers were not prepared to design systems for end users. In the recent past, the emphasis has been to design systems for more efficient use of the equipment and for ease of maintenance. Education and training for the system developer has emphasized techniques for developing flexible systems which meet functional requirements and for improving maintainability. Currently, the designer who is still learning how to best accomplish system design under the old guidelines and requirements is expected to change his orientation. Now he must emphasize design characteristics needed for the end user to interface with the machine. However, many professionals in the field of system development are not aware of guidelines or the underlying concepts of human factor design. Consequently, many new systems and most existing systems are not user friendly.

By educating these professional system developers about human factor design guidelines, they can then apply the guidelines. Of particular importance to companies with many existing menu systems is the fact that the guidelines can be applied during maintenance to redesign menus so that the menus will be more user friendly. The menus can be further enhanced by prototyping the menus for the users' comments and reactions. The combination of the two techniques provides a simple procedure which will enable the developer to create or redesign menus that better suit the needs of the user.

This paper will discuss some of the guidelines for developing user friendly menus and the application of these guidelines when modifying an existing system. It will do this through the description and discussion of two examples of the application of human factor design guidelines in the redesign of

menus in existing systems. Finally an attempt will be made to evaluate the process which was followed and the success of the redesigned menus as perceived by the end user.

Human Factor Design Guidelines

There are several general purpose human factor design guidelines for the human/computer interface. These general purpose guidelines apply regardless of the type of interface that is being used. In addition, there are guidelines or rules of thumb that apply to particular types of interfaces such as menus. Many of the guidelines are derived from psychological concepts such as cognition, memory, perception, and motivation. However, a standard set of guidelines which would guarantee the desired level of user friendliness and productivity does not exist at least is not currently known. So the developer needs to be familiar with a variety of guidelines and be able to select or use the ones that best fit a particular situation.

There is a fairly large volume of literature that is relevant to those interested in human factor design. Four books have been included in the list of references for the paper. These four books provide a starting point for someone interested in doing further reading in the area of human factors. Each reference discusses a variety of human factor design guidelines. Rubinstein and Hersh [3], and Heckel [2] both offer an interesting introduction to the subject area and many useful guidelines. Shneiderman [4], and Bailey [1] offer more advanced reading in the subject area and give many additional references in addition to giving guidelines. Bailey [1] also has chapters on the underlying psychological and physiological concepts that pertain to human factor design.

The most important commonly recognized guideline is to know the system's users [1,2,3,4]. This basic guideline, "to know the user," is deceptively simple. It is a simple idea to express, but represents a difficult goal to achieve. Probably the most common problem with achieving the goal is the designer's assumption that either he understands the users and their tasks or that he is representative of the users. In order to overcome this problem, the designer needs to realize that people are different and that they do things in different ways depending on their background and skills.

Another basic guideline is to know the subject or task that the system is to accomplish. Of course, defining system requirements has long been a recognized need in system development. This guideline addresses that same need, only in this case, the emphasis is on knowing the task as the user perceives or views it.

Many of the other guidelines are directed toward ensuring that the user can easily use the system without being confused and frustrated. The guidelines are either ones that simplify the

user's task, or facilitate his use of the system by helping the user determine what he should do next or helping him recover from some action.

Project setting

In the Spring semester of 1987, I taught a graduate level course entitled, Software Quality Enhancement. Half of the course was devoted to human factor design. Each student in the course had to complete some type of project or paper about the topic. Students were encouraged to evaluate the use of human factor design in an existing system and/or to apply human factor design criteria in a system development or maintenance project. Eight of the 13 students in the class were currently employed as system analysts or analyst programmers and two of the remaining five had worked as programmers. Three students worked for Caterpillar Tractor Co., three worked for State Farm Corporate Headquarters, one worked for Illinois State University Computing Services, and one worked for Country Companies Corporate Headquarters.

From the projects that were done for this class, two were selected as examples for this paper. One project was done by Kieth Simmons, a systems programmer for Computer Integrated Manufacturing Technology, Caterpillar Tractor, Inc. The other project was done by Mark Watson for Miller Electric Corp., an electrical contracting firm.

Project One

The system that Kieth Simmons modified is a specialized system used by a select clientele. The system is used only by Tool Selection Managers who have considerable experience and knowledge in the area of tool selection. The system must perform 5 primary functions in addition to file management functions. The particular system was in a maintenance phase already because of proposed system changes which had nothing to do with the user interface. The system uses menus to accomplish the human/computer interface. Kieth was very familiar with the system and with the user community. He felt that user satisfaction and performance could readily be improved with modifications to the menu hierarchy and design.

Kieth evaluated the current menus and redesigned them based on human factor design criteria. The two design criteria that Kieth used most extensively were the following: know the user and know the task which the system is to accomplish. He also frequently applied the criteria which indicate that communication should be done with the user's language and that the system should build on the user's knowledge. The application of other criteria helped direct the general design of the menus and were important considerations in selecting a particular menu design.

Some of these underlying criteria were the following: let the user rather than the computer control the situation, be consistent with the user's view of the task, avoid confusing and/or frustrating the user, and facilitate the user's task.

In conjunction with applying the human factor design criteria, Kieth prototyped the menus. The tool selection managers then experimented with the prototype. Next he made changes that the users requested unless the requested changes were mutually exclusive. After a new prototype was created the users again had a chance to evaluate the prototype. The users saw three different prototypes before the menu redesign was completed.

Figure 1 shows the menu hierarchy of the resulting system and the layout for the main menu for the Tool Selection system. It is important to note that this system has a small trained end user group. They have worked in this area for a number of years and have a jargon of their own. The terminology on the menus may be meaningless to most people, but it is familiar terminology to the end users of this system. So by using this terminology, the designer was following these human factor design guidelines: communicate with the user's language, and build on the user's knowledge.

The main menu (Figure 1) contains the five primary functions that the tool selection manager must perform plus the file management function. In the original version, the files option was not included in the main menu. Instead, it was included as the first option or menu choice for each subsequent function menus. While there was justification for either location of the files option, the tool selection managers did not like it in each of the secondary level menus. The managers felt that having the files option repeated in each secondary menu was confusing. So in following one of the guidelines which says that the design should not confuse or frustrate the user, the file option was placed in the main menu. The file option was first placed in alphabetical order on the main menu, between DCLASS and GRAPL. However, the managers wanted it in the first position since that was where a similar option was in another system that they use. To avoid confusing the user, the placement of the file option was such that it maintained consistency with another system.

Figure 2 shows the menus which are related to the DCLASS option on the main menu. Menu 2, DCLASS Development, is in functional order. In order for the manager to make a change in a DCLASS program, he would get the file, make the required changes (edit), compile the modified program (prep), test the program and return the program to the production area. The utility programs option refers to seldom used DCLASS utility programs. These programs are used only once or twice a month. By placing them in a submenu, the DCLASS development menu is clearer and less confusing to the user. Also, the extra effort required by the user to get to the submenu is not a problem for the user since

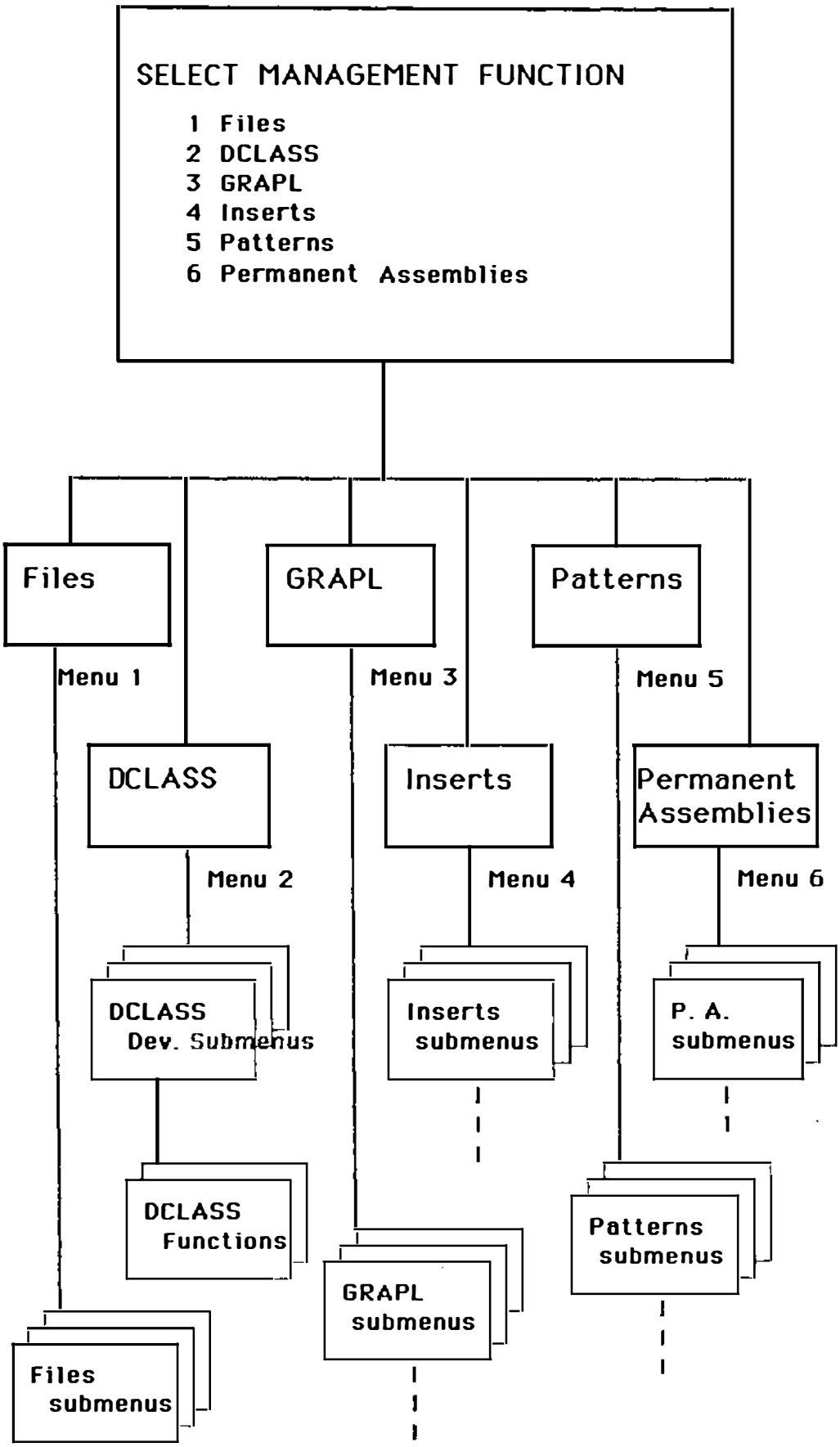


Figure 1: Tool Selection: Main Menu and Menu Hierarchy

the option is infrequently used. In this instance, having the menu in functional order makes it consistent with the user's view of the task he is completing. In this way, the interface accurately reflects his viewpoint of the job.

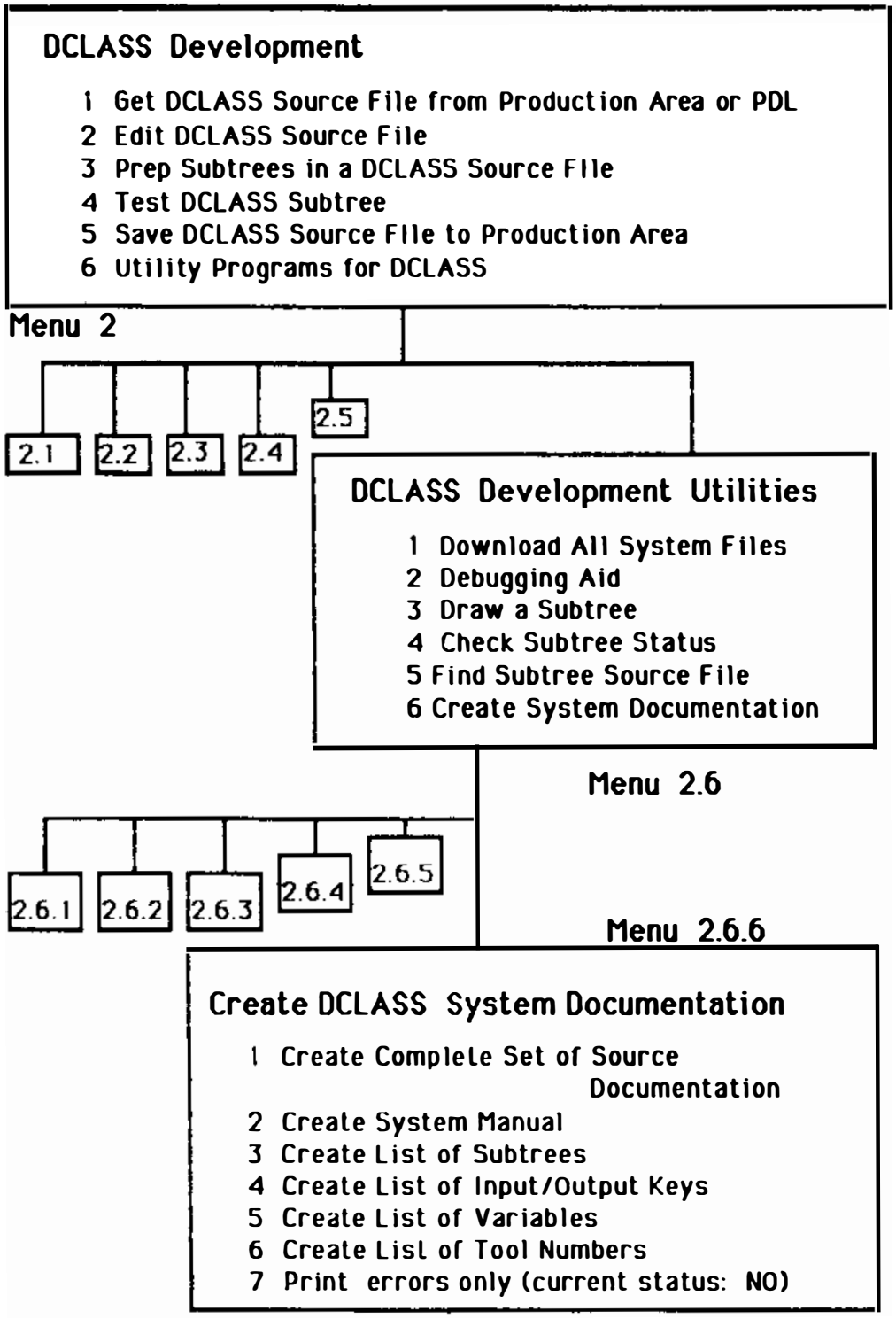


Figure 2: Tool Selection: DCLASS Menus

Menu 2.6, DCLASS Development Utilities (Figure 2), is arranged in descending frequency-of-use order. The higher selection numbers are used less frequently. The first version of this menu was in alphabetical order; however, since five additional options are to be added to this menu over the next two years, the frequency order was selected. This selection was based on department guidelines which discourage changing the menu number of a function. So new functions would have to be added to the end of the list and the alphabetical order could not be maintained. It was thought that the addition of new options to the end of the menu would cause less confusion for the manager if the initial order of the menu was frequency rather than alphabetical.

Menu 2.6.6, Create DCLASS System Documentation (Figure 2), is an example of a standard DCLASS submenu. The system designers started with standard DCLASS functions and added the features most requested by the managers. In this case, the function has not changed in two years, and the users were satisfied with the menu. Therefore, no changes were made to this particular menu.

Menu 3 (Figure 3), GRAPL Development, is in functional order. First the manager gets a file, then makes changes to it, and finally returns it to the production area. Changes to GRAPL libraries take different forms. For example, the user can edit a GRAPL program, move the program from one library to another, extract a program from a library, or insert a program into a library. Managers at each installation have their own change method, so each option needs to be kept as a separate option. Designing the menu in this way follows another human factor design guideline which is to let the user control the situation rather than letting the computer or the system control the manner in which the task is done.

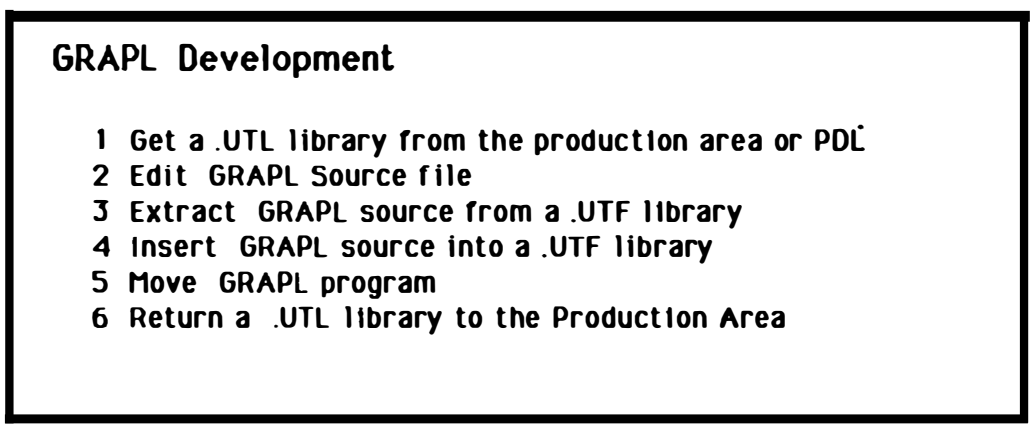


Figure 3: Tool Selection : GRABL Menu (Menu 3).

Menu 4 (Figure 4), Insert Management, is in frequency-of-use within functional order. The manager gets the insert files, makes changes, and saves the files. There are two ways to change the files. One way is to mark the inserts as preferred and to add new inserts by replacing the current library with a new library. The list function does not change the insert files, but the list of inserts is seldom printed unless there is a change made to the file. In the first prototype version of this menu, the list function was after the replace function. But the managers requested that the list function be after the mark function, since the replace function is only used two or three times a year. The placement of menu options for this menu is an example of applying information learned from the psychology of memory. Given a list of similar items, people tend to remember the first items in the list and the last and forget those in the middle. In this case, the user is apt to remember that the first thing he does is option 1 on the menu list and the last thing is the last option on the list; however, he may not remember exactly what the middle three do. So the user will probably need to read the choices for options 2,3, and 4, but merely check the number of the last choice for saving the files. Thus this menu then is more user friendly since the options that the user will most often use are located where it is easy for him to find them.

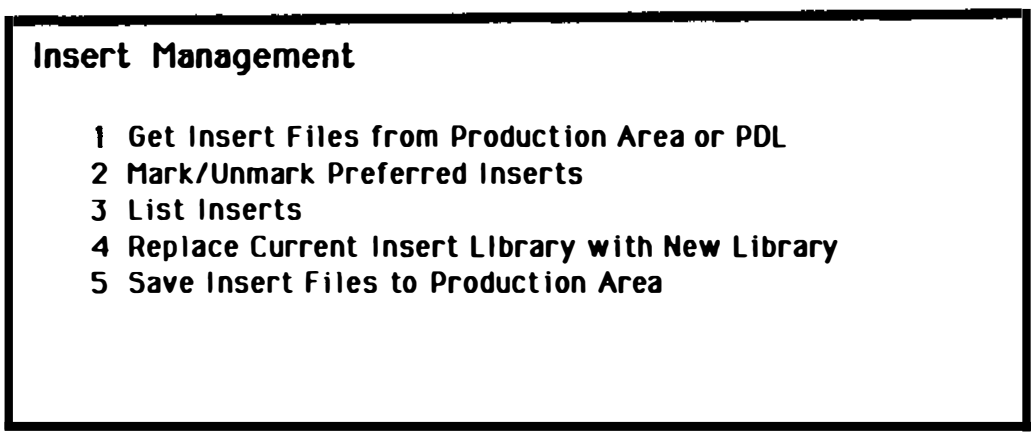


Figure 4: Tool Selection: Insert Menu (Menu 4).

Menu 5 (Figure 5), Pattern Management, is in functional order. In other words, the manager gets a pattern library, makes a change to it, and then returns the library to the production area. In the first prototype, the sequence of options 2-5 was Rename, Move, Merge, and List. The managers requested the current order since they felt that 90% of the time they would be renaming and listing the patterns rather than moving or merging them. The sequence of items in this menu provides an example of applying the following two guidelines: letting the user control the process, and making the system consistent with the job that the user is trying to accomplish.

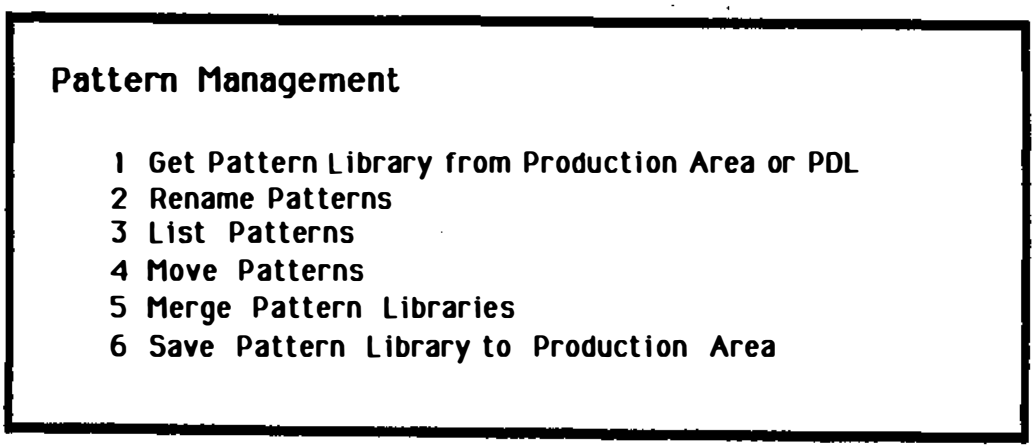


Figure 5: Tool Selection: Pattern Menu (Menu 5)

Evaluation of Project One

This project was quite successful in making the menu interfaces more user friendly. The maintenance project which is being done is scheduled for a January, 1988 completion date. The original intention was to include the menu changes in the production system at that time. However, the users who were involved with evaluating the prototypes kept requesting that the changes be made operational in Spring, 1987 rather than waiting the additional 6 to 8 months. By June, the changes were made to the current system for one installation to use as a pilot study. Reaction of the users has been so favorable that other installations are requesting that the change be made for them too. The designer/programmer spent less than 80 hours making these menu changes which was a very small percentage of the scheduled maintenance time. The users did not keep a record of the time that they spent on evaluating and working with the prototypes.

This project provides a prime example of how creating an awareness of human factor design criteria in the system developer can result in more user friendly system interfaces. In this case, the maintenance had been planned without any consideration given to making changes in the menu design other than adding new functions. When Kieth Simmons, the person responsible for the maintenance changes, realized the possibility of increasing user satisfaction with the system by applying some of the human factor design guidelines, he made appropriate changes to the system as a class project. Given the ease with which these modifications were made and the users' enthusiastic response, planning for this type of change in future maintenance projects is a more likely possibility. The project certainly gives some basis for the justification of making this type of change during maintenance.

Project Two

The system that Mark Watson evaluated and recommended changes to redesign the menus is a system to aid in estimating job costs and recording current information about customers, jobs, inventory, and employees. The system had been developed for Miller Electric Corp., an electrical contractor. This package is more general purpose and has a wider variety of business functions than the system in project one. The user base is also more diverse for this project. So the changes that were recommended for this system, were based on different guidelines and on the designer's application of the guidelines. However, the guidelines about knowing the user and designing the system for the user were still a key factors in the redesign of the interface. Some of the other criteria which Mark applied in modifying the menus were the following: facilitate the user's task, avoid confusing and/or frustrating the user, anticipate potential problems, be consistent with user's view of the task, build consistent user interfaces, and maintain user's interest.

The first change that Mark recommended was a cosmetic one to make the screens more interesting and appealing to the user. Figure 6 shows the before and after versions of the main menu. In the redesigned screen the menu has been enclosed in a box and the size of print has been varied to help distinguish between the screen heading and the menu selections. These two changes should improve the readability of the menu and make it more interesting to the user. All screens in the redesigned package have this same basic format thus consistency is added to the interface design making the users' task easier.

One problem with the existing system was that different codes were used to exit from the current menu. The design was changed so that the entry of a common code, 0, caused the system to return to either the main menu or the previous menu. Again this was a change made in order to have consistency in the package. Figure 7 shows the screen that is used when someone wants information about a part. The original screen used "Q" to return to the previous menu: the redesigned screen uses "0".

Another feature which was incorporated in the redesigned menus was a subheading which identified the menu and the general function of the menu. In the original version the menu entries were often cryptic and did not identify the function that was being performed. In Figure 7, shown on the previous page, the original screen gives no indication of the screen function. This same screen is used to add parts to inventory, to view current information about a part, to change information about the part, or to delete the part from inventory. Since the persons who perform these functions are subject to frequent interruptions, not having any indication of the current function on the screen was a problem. The redesigned screen in Figure 7 indicates that this screen is part of the inventory control function (choice 5

MILLER ELECTRIC

1 - Estimating
2 - Customer Information
3 - Add/Delete A Job
4 - Update & Query Jobs
5 - Inventory Control
6 - Employee Information
7 - Re-Index Data Files
0 - Finished

- Pick One

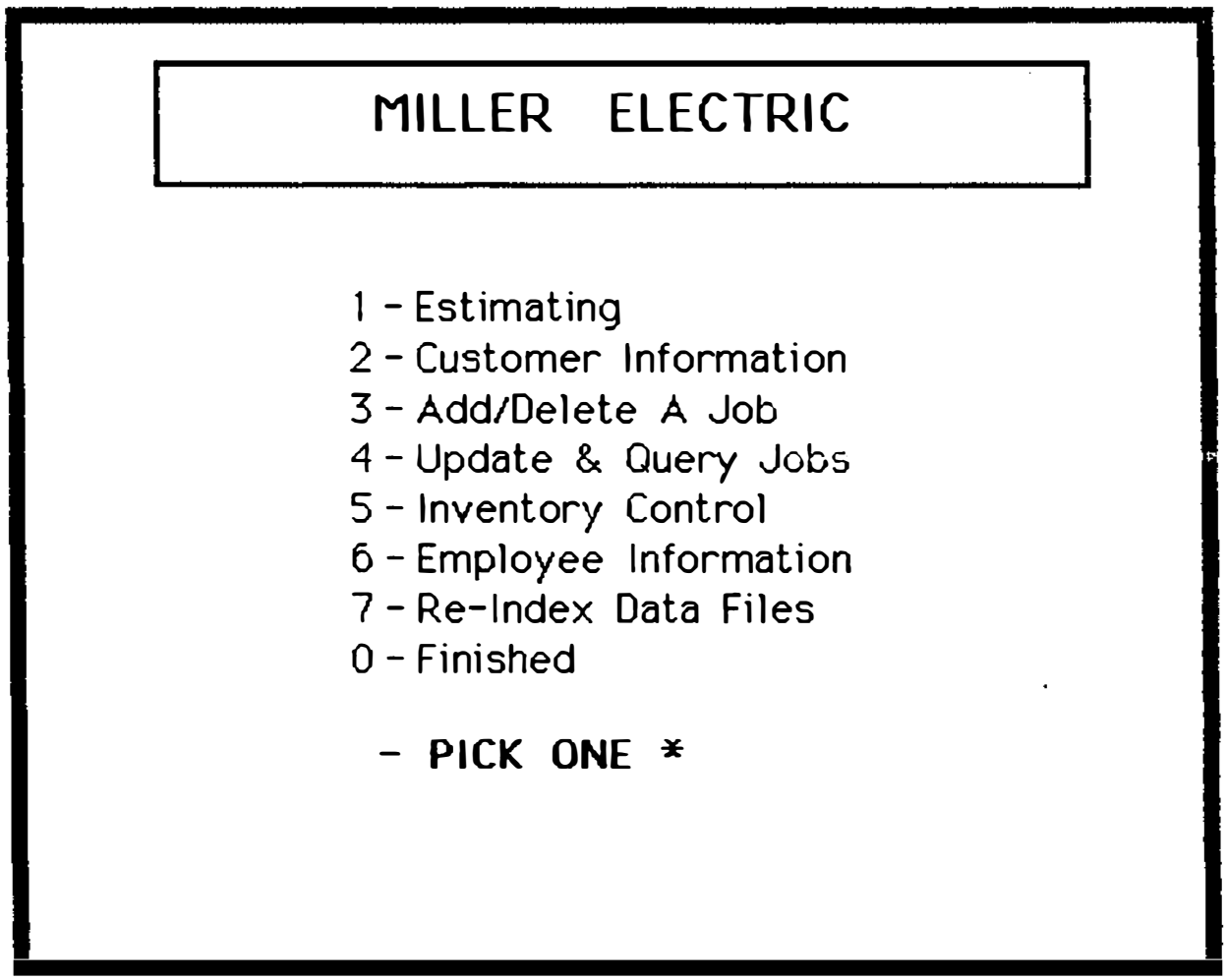


Figure 6: Miller Electric: Original and Redesigned Main Menu

on the main menu) and is a result of an inquiry about a part (choice 4 on the inventory control menu). The revised menus are less apt to confuse the user thus the users are less apt to get lost in the process of completing some function. The original version might have been adequate if the user could have completed a task without interruption; however, that was not true in the environment in which this system is used.

```
                    MILLER ELECTRIC

DESCRIPTIONWIRE MARKER BOOK

LAST COLUMN       850

COST      657

QUANTITY        0

PRICE 1      765

PRICE 2      995

LABOR      0

LOCATION158-19A

ORDERPOINT        0

ENTER Q TO RETURN TO PREVIOUS MENU
```

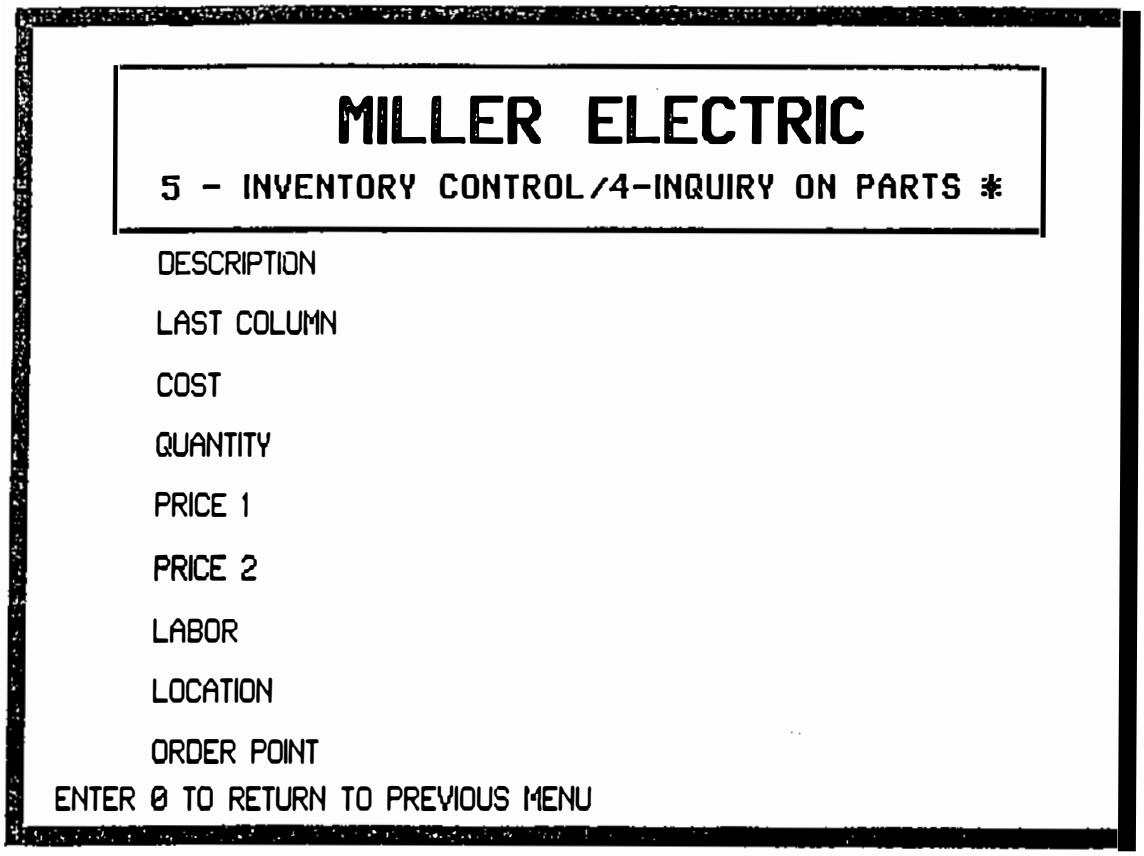


Figure 7: Miller Electric: Original and Redesigned Inventory Control Part Inquiry Menu

Other changes that were suggested involved changing labels or making the menu options more clear. Figure 8 shows the before (Figure 8a) and after (Figure 8b) sets of screens that are used to print an invoice for a particular contracting job. In the after set, the heading gives an indication of the function that is being done: in the original set there is no indication that an invoice is being printed. The change in the message "now printing" to "now printing invoice" helps to clarify for the user what is taking place. The screens could still be enhanced by giving some indication of the nature and length of the job description lines, or the format of the job number. In general, the human factor design criteria that were being implemented in these cases were those concerned with helping the user cope with the system and avoiding user frustration and confusion. In other words, the designer was trying to make using the computer to accomplish the task simpler than not using it.

First Screen

MILLER ELECTRIC

ENTER THE JOB NUMBER TO USE: 0

Second Screen

ENTER 1st LINE OF JOB DESCRIPTION THIS IS

ENTER 2nd LINE OF JOB DESCRIPTION ONLY

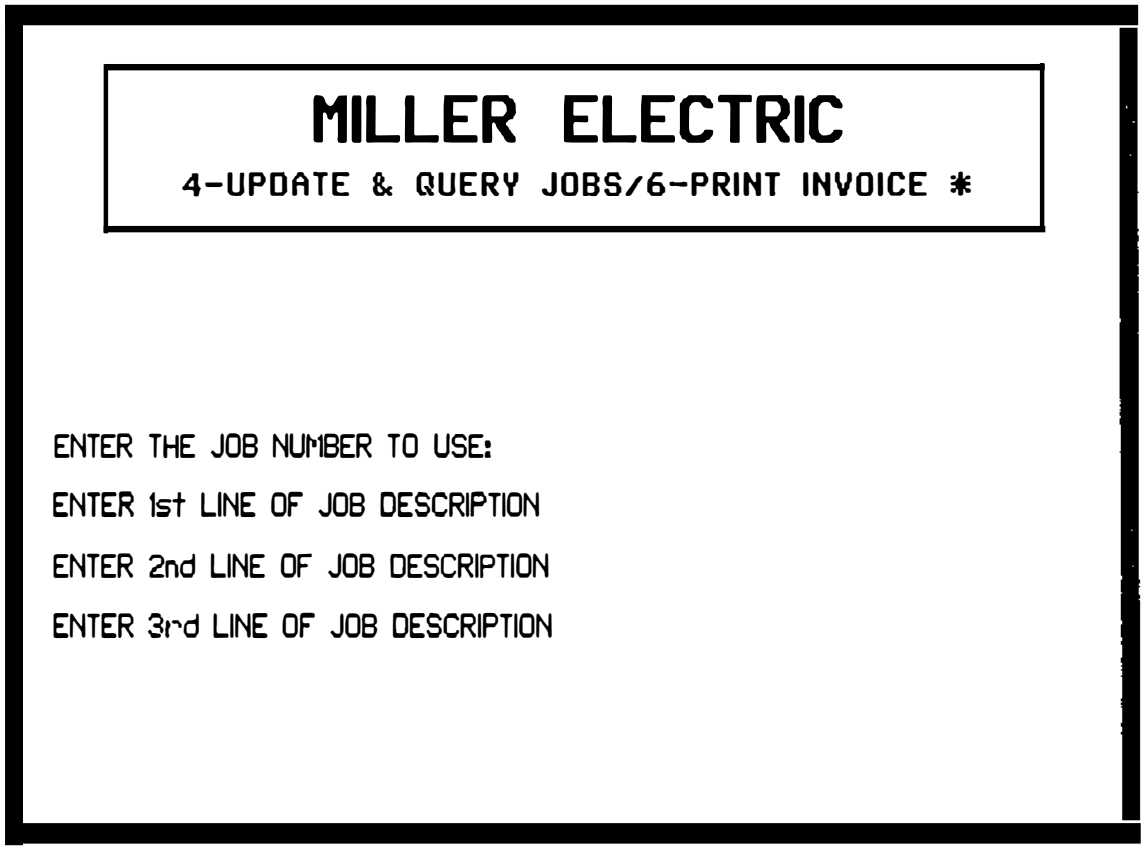
ENTER 3rd LINE OF JOB DESCRIPTION AN EXAMPLE

Third Screen

ADJUST PRINTER FORM
Press any key to continue. . .

NOW PRINTING

Figure 8a: Miller Electric: Original Set of Screens for Printing Invoice



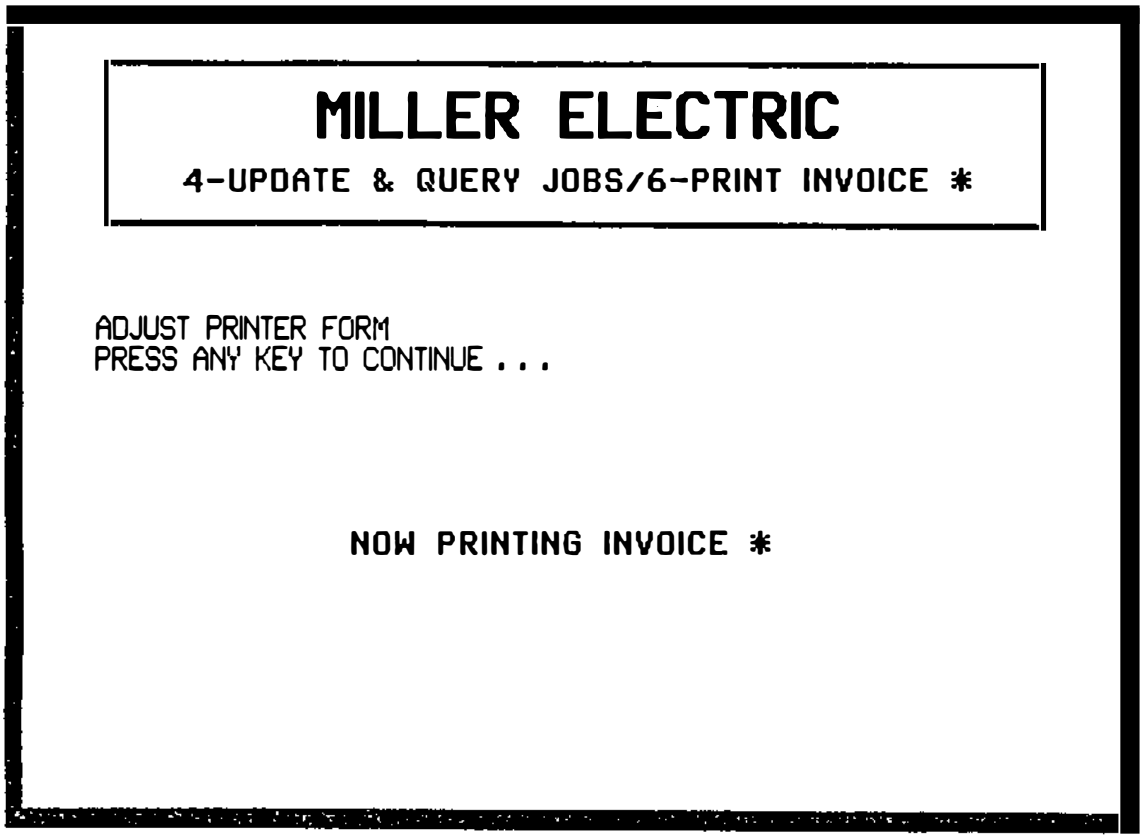


Figure 8b: Miller Electric: Redesigned Set of Screens for Printing Invoice

Evaluation of Project Two

In this project, the suggested changes to the menus were cosmetic in nature and not much code needed to be changed. However, the impression given by the redesigned menus is one of a more polished product which impacts the user's perception of the adequacy and reliability of the system. In this case, by favorably affecting the users' first impression of the system, the users' perception of the system will be more positive. Consequently, most users will perceive the system as easier to use and will be more satisfied with it. Furthermore, the time and cost involved in making these changes seem to be relatively low for the benefits received in terms of increased user satisfaction with the system.

In this project, the users did not have an opportunity to experiment with a prototype and make suggestions about the screen design. It is expected that the users would have recommendations to make about the design so that the menus would be easier for them to use. A prototype was not done because the designer did not have access to a prototyping tool and because of the limited amount of time available.

Conclusion

In each of the projects discussed in this paper, simple human factor design criteria were applied in the redesign of the menu interface. Criteria which were used most often are the following: know the user, know the task which the system is to accomplish, communicate with the user's language, build on the user's knowledge, let the user control the situation, be consistent with the user's view of the task, avoid confusing and/or frustrating the user, facilitate the user's task, anticipate potential problems, build consistent user interfaces, and maintain user's interest.

These projects demonstrate that menu interface systems can be made more user friendly by applying human factor design criteria. One key consideration in the success of such a project seems to be the designer's awareness of human factor design guidelines. A second key consideration is the designer's ability to appropriately apply the guidelines to the particular system. While the evaluation of only two projects does not provide conclusive evidence of the success of the redesign procedure used, it would seem to offer suggestions for ways to enhance the user interface of existing systems.

If the procedure of applying human factor design criteria to the redesign of menus is followed when maintenance or system enhancement is needed, the menu portion of the system is apt to be more user friendly. Furthermore, the system's users will probably perceive the entire system as more user friendly and of

better quality. These perceptions would increase their acceptance and satisfaction with the system. The changes made to the menus would be primarily cosmetic and organizational and should have only minor impact on existing code. Consequently, relatively little of the information systems department's resources will be needed to effect the changes. However, the user interface will have been changed so that the users will view the system more favorably. The results seen in these two instances would seem to have significant ramifications for companies as decisions are made regarding proposed maintenance of existing systems and proposed training of system development personnel.

References

[1] Bailey, Robert W., Human Performance Engineering, Prentice Hall, 1982.

[2] Heckel, Paul, The Elements of Friendly Software Design, Warner Books, 1984.

[3] Rubinstein, Richard and Harry Hersh, The Human Factor, Digital Press, 1984.

[4] Shneiderman, Ben, Designing the User Interface: Strategies for Effective Human-Computer Interaction, Addison-Wesley, 1987.

AUTOMATED TESTING OF USER INTERFACES

Software Engineer
Mentor Graphics Corporation
Mark A. Johnson

ABSTRACT:

The evolution of Mentor Graphics Corporation's products to a graphical user interface required the development of a new method for capturing and driving test cases. The graphical user interface depends heavily upon the use of a graphic input device (such as a mouse, bitpad puck, touchpad, etc.) to cause menus to "pop–up" or "pull–down" and forms to "pop–up" to be filled out. The user actions which exercise the graphical user interface cannot be duplicated by the traditional transcripting of executed commands being replayed into the application command line. Therefore, an Input Capture Tool was developed to allow the logging and replay of the graphical and keyboard input which drives the graphical user interface. This tool has been effective for automating testing, but is sensitive to changes in the graphical user interface.

BIOGRAPHY:

Mark A. Johnson is a software engineer in the Corporate Quality Department at Mentor Graphics. Part of his work includes research and development of software tools for quality improvement. He has previously worked for Intel Corporation as a software engineer and test engineer. He has a Bachelor of Science in Computer Science from California Polytechnic State University, San Luis Obispo, California.

Mark A. Johnson
Mentor Graphics Corporation
8500 S.W. Creekside Place
Beaverton, OR 97005–7191
(503)626–7000
tektronix!tessi!mntgfx!pdx!mjohnson
ogcvax!sequent!mntgfx!pdx!mjohnson

Automated Testing of User Interfaces

1. Introduction

Automating software testing is a major productivity aid in utilizing the limited resources available for testing software. Automation provides the ability to easily re-use tests as software is changed during development and maintenance. Once a product is shipped to customers, having automated tests can be an aid in correlating reported problems to testing previously done for identification and correction of weaknesses in tests. The automation of software testing generally involves developing a means to drive the software with test cases, and to compare the results produced to a known good reference.

2. The Problem

At Mentor Graphics our products are software applications which an electronics engineer would use to design, simulate and layout electronic circuits. These applications use engineering workstation graphics to display the design and use a graphic input device, such as a mouse, for pointing at objects, making selections, etc. The input of commands was originally done by typing the commands at a command line, or through the use of function keys. As the commands are executed, they generate a transcript which can be saved as an ASCII file and resubmitted to the application. This allowed the creation of suites of automated tests which would test the application, and exercise the user interface at the same time. To verify the results of a test, the transcript from the current run can be compared to a known good reference transcript. Also, the design created by the test run can be compared to a known good design.

As our applications became more sophisticated, they have increasingly used the graphic input device for command and data input. This interactive graphical interface, where a device such as a mouse causes menus and forms to pop-up, pull-down, or cascade, performs the same functions as the command line interface. See Figure 1 for an example screen from an application. A transcript of the command execution is still produced. However, rerunning the transcript exercises the command line interface and not the graphical interface. To test the graphical interface, a person could manually use the graphic input device to exercise the application. This has at least two problems. First, by manually exercising the application, neither the transcript nor design produced will exactly match a reference. Therefore, checking the results of a test case becomes subjective. Second, it is very tedious to do the same sequences of operations repeatedly, which can lead to missing test cases, performing test cases incorrectly, etc.

A summer student, John Gregor from Oregon State University, was assigned to do some of this graphic interface testing. After doing a few test sessions, and repeating them as the software was debugged and changed, he decided that there should be some way to automate this process. His initial experimentation in this area led to a testing tool we call the Input Capture Tool.

3. The Input Capture Tool

The automated testing of the graphical interface required developing a means to capture and replay the non-command line input to an application. This input consists of thousands of events created by sequences such as pressing a mouse button causing a menu to pop-up, then the mouse movement traversing the menu to select a command, and possibly keyboard input filling out a form which was popped-up by the command selected from the menu (this sequence would be

several hundred events long). The key to automating testing of the graphical interface was finding a way to intercept or inject input events in the stream between the keyboard or graphical input device and the application.

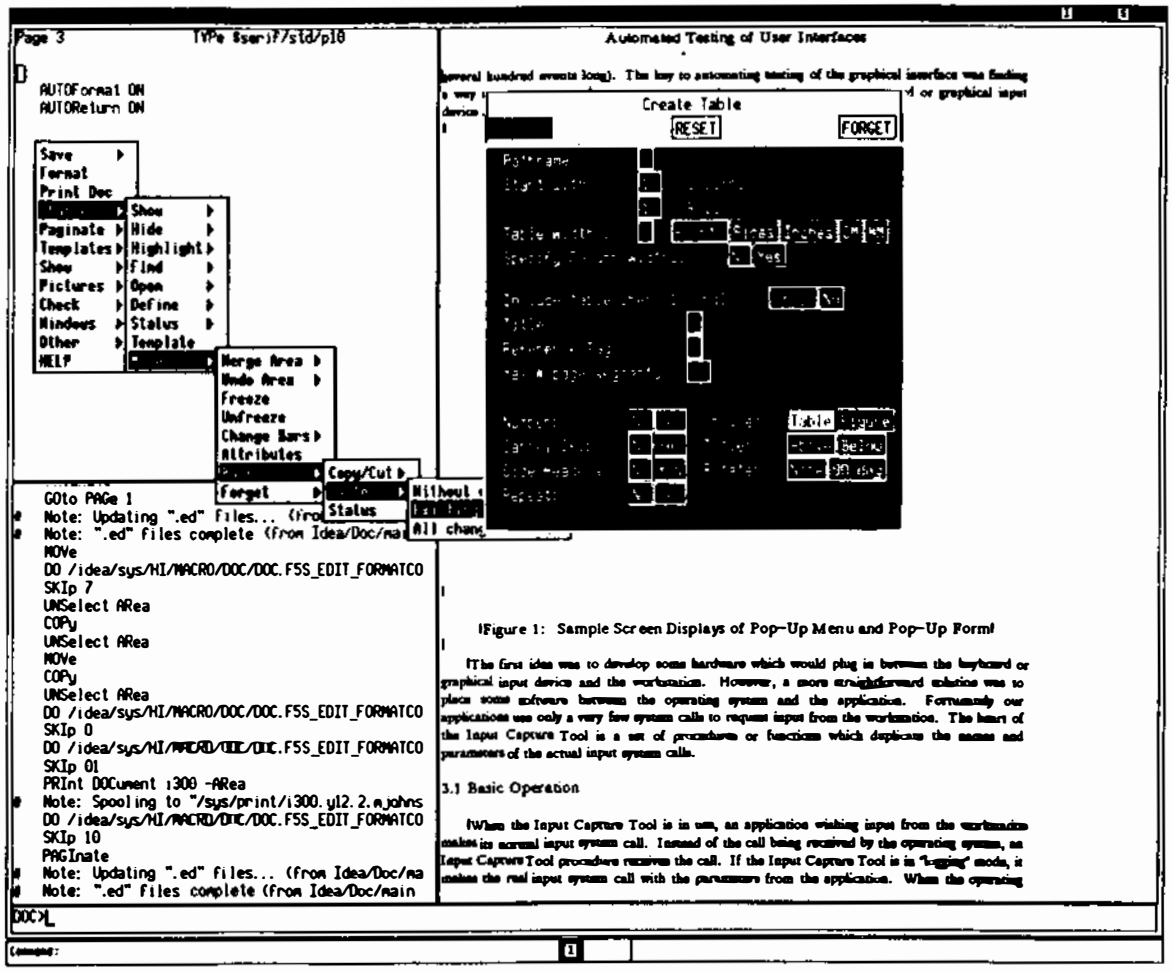


Figure 1: Sample Screen Displays of Pop-Up Menu and Pop-Up Form

The first idea was to develop some hardware which would plug in between the keyboard or graphical input device and the workstation. However, a more straightforward solution was to place some software between the operating system and the application. Fortunately our applications use only a very few system calls to request input from the workstation. The heart of the Input Capture Tool is a set of procedures or functions which duplicate the names and parameters of the actual input system calls.

3.1 Basic Operation

When the Input Capture Tool is in use, an application wishing input from the workstation makes its normal input system call. Instead of the call being received by the operating system, an Input Capture Tool procedure receives the call. If the Input Capture Tool is in "logging" mode, it makes the real input system call with the parameters from the application. When the operating system returns to the Input Capture Tool, the returned parameters are logged in a file, and then

the parameters are returned to the calling application. If the Input Capture Tool is in "replay" mode, it reads the parameters from a file and returns them to the calling application. See Figure 2 for a graphical representation of this flow.

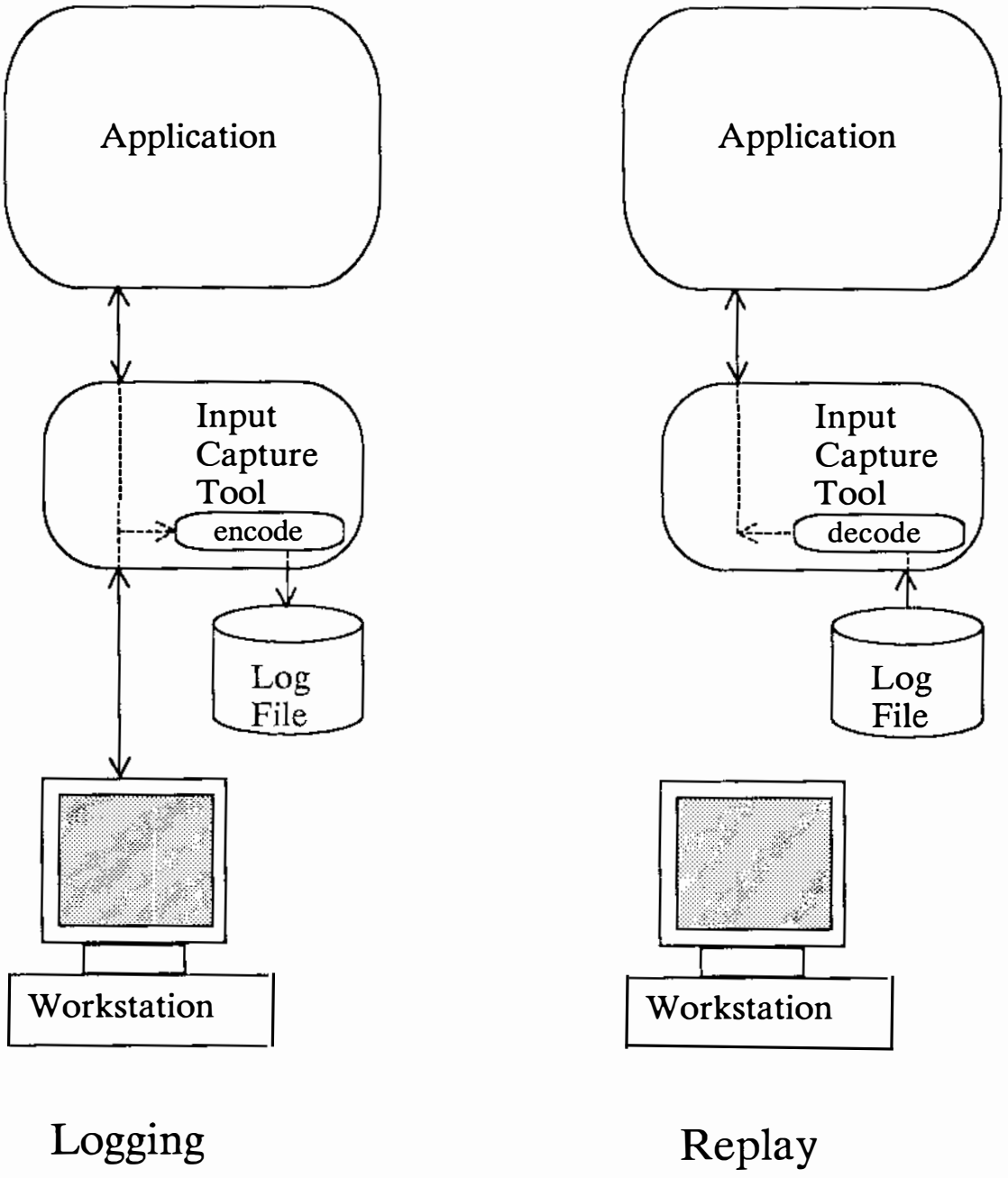


Figure 2: Input Data Flow to Application with Input Capture Tool

To initiate a session with the Input Capture Tool, the system call made by an application to initialize the graphic display is intercepted. The Input Capture Tool displays its sign-on message and version and then queries the user for the mode of the session -- "logging" or "replay". The user is queried regarding the log file to be used and the file is initialized as required. The Input Capture Tool performs its own initialization tasks, calls the operating system to initialize the graphic display for the application, and returns the results of that call to the application.

Termination of a session occurs when the application makes a system call to release the graphic display. This call is intercepted and the Input Capture Tool closes the log file, does any other cleanup it needs, makes the system call and returns the results of that call to the application.

3.2 Usage Example

To use the Input Capture Tool, it is bound to the application to be tested. The application is then invoked in the normal manner. During the application's initialization, the Input Capture Tool's initialization procedure is called. The user is queried for the mode of operation, and "logging" is selected. The user is then queried for a file to be created to receive the logged input events. The file is created, and initialization of the application continues. Once the application is up and running, each input action of the user, such a keyboard stroke or mouse movement, is logged. During idle time, when the application is not actively doing something and the user is not providing input, the Input Capture Tool records the "no-events" which occur. When the user decides the session is complete, they terminate the application in the normal manner. During the application's cleanup the Input Capture Tool's cleanup procedure is called and the log file is closed.

To replay a session, the application with the Input Capture Tool bound to it is invoked in the normal manner. During initialization of the application, the Input Capture Tool's initialization procedure is called. The user is queried for the mode of operation and "replay" is selected. The user is then queried for a log file to be read. The log file is opened and initialization of the application continues. As the application makes requests for input, the Input Capture Tool feeds the application events from the log file. These events cause the application to perform exactly the same actions as when the session was logged. The "no-events" in the log file provide spacing in time so that the replay occurs at the same rate as the logging. The last event in the log file is the action the user took to terminate the application during logging. This causes the application to terminate in the same manner as the logging session. During the application's cleanup the Input Capture Tool's cleanup procedure is called and the logging file is closed.

3.3 Additional Functionality

To make the Input Capture Tool a useful tool, additional functionality is required. Editing, to be able to make additions, deletions, and changes to log files, is important. User control over replay, to allow stopping, starting and stepping through the log file is useful. Inserting comments into the log file while stopped during replay could help with editing. Control of the replay rate, to allow speeding up or slowing down of replay would add convenience.

3.3.1 Editing

The present level of editing functionality is provided by the log file being encoded in ASCII format. The parameters of the logged events are returned by the system calls as various data types, such as integer, character (non-ASCII) or enumerated types. Integers are put out in ASCII, characters are converted to ASCII (or short ASCII strings for non-printing characters), and enumerated types are converted into short ASCII strings. Also, the type of system call who's parameters are logged is indicated, because the parameters vary from system call to system call. See Figure 3 for a section of a log file. A text editor can be used to read and modify this ASCII log file. For instance, unwanted actions of a test case can be removed, or additional actions can be added or several test cases or pieces can be merged into one test case.

Advanced editing, in an interactive mode while replaying the log file with the application, has been considered. This could be fairly complex to implement and would require being able to go forwards and backwards in the log file, being able to switch from replay to logging and back, being able to indicate areas of the log file to be deleted, etc. During this editing, the actual design being viewed could end up deviating substantially from the design which would be created by just running the log file, thereby misleading the user in their editing. Also, the Input Capture Tool would have to be able to recognize events in logical sequences so that it did not present the application with situations where an up-keystroke preceded its matching down keystroke, or only one of the two was deleted, etc.

3.3.2 Replay Control

The user should have some control over the replay function of the Input Capture Tool. The user should be able to signal to the Input Capture Tool, via the workstation keyboard, to stop replay, single step replay -- once replay is stopped, insert comments in the log file -- while replay is stopped, and continue running the replay. Single stepping should be available on a per event basis and on a logical sequence of events basis. An example of a logical sequence of events is pressing a mouse button, moving the mouse, and releasing the mouse button. This sequence may be several hundred events long, from the first button press, through dozens of mouse locations as the mouse is moved (interspersed with "no-events"), to the final button release. Also, once replay is stopped, the user will be able to key in text which will be inserted into the log file as comments. In this manner the user will be able to easily identify specific locations in the log file for study or modification using a text editor.

Controlling the speed of replay of a log file would be an added convenience. It may be desirable to slow down the replay rate so that the events leading up to a software crash can be closely observed. For normal testing, running the test faster will allow more tests to be run in the same period of time. The rate of replay is controlled by how fast the application can process events from the log file. Delay can be added by simply waiting between returning events to the application. Speed up can only be done by removing some of the events the application would normally process. The "no-events" signify points in time when the application was waiting for user input and there was none. Removing "no-events" therefore reduces the amount of waiting between actions by the application, and speeds up replay. However, the removal of "no-events" can cause problems in the behavior of the application, as mentioned under Problems in Section 5 below. Implementation of some control over replay speed is being investigated.

```
h 2 ICT 0.8  Wednesday, June 10, 1987   10:39:44 am (PDT)
#first line of file is header with file format version and Input
#  Capture Tool version
#Format of data below (# signifies a comment):
#s = smd type system call
#nev = no-event; first number is count of no-events between events; last
#  number is status returned from call, 0 = ok
#loc = location; next two numbers are x, y coordinates; next is character
#  (key stroke) 0 = no key, M2 = mouse button 2 depressed; last number is
#  status returned from call, 0 = ok
#inp = keyboard input; first is character (key stroke), b y e; last number
#  is status returned from call, 0 = ok
#the next sequence is operation in a pop-up menu.  Mouse button M2
#activated the menu
s loc 268 412 0 0
s loc 268 412 M2 0
s nev 2 0
s loc 291 411 M2 0
s nev 2 0
s loc 346 411 M2 0
s nev 2 0
s loc 390 411 M2 0
s nev 2 0
s loc 408 427 M2 0
s nev 2 0
s loc 415 487 M2 0
s nev 7 0
s loc 426 490 M2 0
s loc 426 490 0 0
s nev 60 0
#the command "bye" is typed at the keyboard to exit the application
s inp b 0
s nev 7 0
s inp y 0
s nev 4 0
s inp e 0
```

Figure 3: Sample Log File

4. Using the ICT in a Testing Program

The Input Capture Tool provides a means to drive our software applications with test cases. The other part of automated testing is capturing some type of result and comparing it to a known reference. To verify the results produced by a test case, different means can be used depending upon what is being tested. If the ability of the graphical interface to drive the underlying application is being tested, then a normal output from the application can be generated, such as a

circuit design. If the proper appearance of the graphics being displayed is being tested, the screen appearance can be occasionally dumped to a file with a command built into our applications, or an Output Capture Tool could be developed.

In practical use the ICT is built into some of our automated regression test suites. The tests are run by a monitor program which starts the application which has been bound with the Input Capture Tool. The monitor responds to the Input Capture Tools queries for operation mode and input file pathname. Then the Input Capture Tool feeds the test cases in the log file to the application. As the application executes the test cases, a design is built, and a transcript of all commands executed by the application is recorded. When the application finishes, the monitor program compares the transcript and/or design to a known good reference, and decides if the test passed or failed.

5. Limitations and Problems

The major limitation of the Input Capture Tool is that it captures physical data, at a low level, which is highly detailed. This is a problem in a number of areas. First, our software runs with several different graphic display sizes and several types of graphical input devices. For each display size, our applications use a different window layout. Therefore the physical locations where actions occur vary depending on the display, and a separate log file needs to be captured for each display size. For each graphical input device, different characters are reported for similar keystrokes. The translation of these keys to be the same operation is done within the application, which detects what type of graphical input device is attached and translates accordingly. Therefore, a log file from a session with a mouse will not work on a workstation with a touchpad.

The Input Capture Tool's use of physical data makes it sensitive to minor changes in the user interface. A change which may not be visible to the user can cause problems for tests. In one case, the area of a pop-up menu which is sensitive to cursor location for highlighting was reduced by 2 pixels. It turned out we had a test case which stopped working because it depended upon selecting a menu item near the edge of the sensitive area and the selection was now outside of the sensitive area. Or changes which are visible, but do not effect user functionality can cause problems for tests. For instance, if the window layout of an application is changed, that may invalidate all of the test cases for that application.

A large amount of physical data passes from the workstation to the application for minor actions in the application. To pop-up a menu, traverse the menu and select an item may take hundreds of physical events. The simple act of adding an item to the middle of the menu will invalidate any test cases which selected items below the new one. To edit these test cases is difficult. A starting point for the menu would have to be estimated, and the distance down from this starting point used to decide if a specific action falls within an area of the menu needing to be edited. If it does need to be edited, some value would have to be selected from the possible range of values and used to create additional location events to move the cursor to the proper section of the menu.

In testing the types of editing which can be done on log files, problems have been encountered. It appears that some interactions between the user and an application have time

dependencies. One example is in attempting to edit pop-up menu sequences. If the number of "no-events" or graphic input device locations during the traversal of a pop-up menu are reduced (to speed up replay or reduce log file size), the pop-up menu tends to stop working. Problems with sequences such as pop-up menus can be avoided by observing the start of the sequence which is a mouse button or key down stroke, and not making any changes until the matching upstroke is found. This would allow the removal of "no-events" for replay rate speedup. However, if too many "no-events" are removed, replay may over-run the application's input buffer, so some limit is required on the number of "no-events" which can be removed.

6. Future Directions

The development of the Input Capture Tool has been useful in helping to understand what is needed to automate the testing of interactive graphical interfaces. A comparison of the sensitivity to change of test cases for the Input Capture Tool to that of test cases for the command line interface provides some interesting insight. The data for the Input Capture Tool is very low level and detailed (physical). The commands for the command line are fairly high level (logical). Moving the command line in the window layout of an application has no effect on test cases. Even changing the name or parameters for a command is easy to deal with. A search of the test case can be made and all old names replaced with new names or parameters changed to match the new requirements. Equivalent changes in the graphical user interface would be difficult to deal with for Input Capture Tool test cases.

Work is presently being done on a modified tool to capture input data at a higher level. The data to be captured will have been translated by the general input utility used by applications. The major advantage to using this data is that only the "no-events" which are relevant to the proper operation of the application will be captured. Also, because "physical keyboard key" to "logical key definition" translation will have been done, the log files will be independent of the type of graphical input device (mouse or touch-pad). There are some drawbacks to this method. Because the keyboard keys are already translated, if the logical key definitions of an application change, this will not be tested by this tool. Because this tool operates at a higher level within the software, there are more types of data which need to be captured, and more calls in the software which need to be monitored.

What is really needed is some set of high-level commands which could be used to drive the graphical interface. Test cases could be composed of "pop main menu", "move to item 4", "pop sub-menu" and "select item 2". Then when a new item 3 was added to the main menu, test cases could be edited and all main menu item numbers 3 or greater would have 1 added to them. Because these were high-level commands, much less editing would be required to make a change. If this functionality was built into applications, it would have to deal with twelve different major modules which handle different types of input, and have to understand the present "state" of the input to know which module should be used. Another possibility would be to create a "macro" processor which would accept some type of high level commands, such as "goto 345 560", "mouse key down", "move to 345 890" and "mouse key up", and convert them into input data for the Input Capture Tool or other input tool.

Session 6

THEORY OF TESTING

"Computer Systems as Scientific Theories: A Popperian Approach to Testing"
John C. Cherniavsky, Georgetown University

"A Model for Code-Based Testing Techniques"
Larry J. Morell, College of William and Mary

"RELAY: A New Model for Error Detection"
Debra J. Richardson and Margaret C. Thompson, University of Massachusetts

COMPUTER SYSTEMS AS SCIENTIFIC THEORIES: A POPPERIAN APPROACH TO TESTING

John C. Cherniavsky†
Department of Computer Science
Georgetown University

ABSTRACT

An approach to software testing is advocated that emphasizes a positive attitude towards the search and discovery of errors. Philosophically the approach is reminiscent of Karl Popper's theory of scientific discovery and analogies to that theory are presented identifying scientific theories with software systems.

The approach is supported mathematically using recent results on functional testing and inductive inference. Classes of testable functions are identified, applications to communication protocols are outlined, applications to reliability and software safety are suggested, and some complexity results are presented.

BIOGRAPHICAL SKETCH

John Cherniavsky received a B.S. in mathematics from Stanford University in 1969 and an M.S. and Ph.D. in computer science from Cornell University in 1971 and 1972 respectively. He was an Assistant Professor of Computer Science at SUNY Stony Brook from 1972 to 1979, and Associate Professor in 1980. From 1980 to 1984 he was Program Director for Theoretical Computer Science at the National Science Foundation. Since 1984 he has been Professor and Chairman of Computer Science at Georgetown University. Since 1978 he has also consulted in the area of software engineering and testing with the National Bureau of Standards in the Institute for Computer Sciences and Technology. He can be reached at:

John C. Cherniavsky
Department of Computer Science
Georgetown University
Washington, D.C. 20057
(202) 625-8629

† **Also affiliated with the Institute for Computer Sciences and Technology of the National Bureau of Standards.**

Even a perfect program still has bugs.
The Tao of Programming - Geoffrey James

I. Introduction

Karl Popper viewed the development of a scientific theory as a series of steps consisting of conjectured theories followed by experiments whose purpose was to refute the conjectured theory. In Popper's view all scientific theories were refutable — past theories have already been refuted; current and future theories were necessarily refutable. This belief arose from a view of man as essentially incapable of fully understanding the universe.

The labor of the scientist in Popper's theory of scientific progress, although seemingly akin to the labor of Sisyphus, is actually an extraordinarily creative activity. The search for refutations is controlled scientific experimentation. The control resides in the choice and subsequent characterization of experiments as "good" experiments. In brief, an experiment is "good" provided that the results of the experiment can refute the theory. This has consequences on the type of experimentation performed ("random" experimentation is generally not effective) and the type of theories that are to be viewed as scientific (a theory for which no refutations are possible is not to be viewed as a scientific theory). Theories that pass many good experiments are to be more trusted than those that have passed less good experiments. Furthermore, a good experiment that refutes a theory provides information for the creative process of developing a new theory.

We borrow from Popper by viewing computer systems as scientific theories. Thus computer systems are never correct, just hypothesized to be correct. The search for refutations becomes the search for test cases that exhibit faults in the computer system. The tester becomes the ultimate skeptic. As a skeptic the tester believes that the computer system is incorrect. Furthermore, this belief is never changed; any and all systems are incorrect and tests must be continually sought. This does *not* mean that systems are never put into production. We use imperfect scientific theories (recognizing their imperfections) and we should use imperfect software ***provided that we recognize its imperfections.*** In fact in many cases we use refuted scientific theories (e.g. Newtonian mechanics) if their imperfections are understood. We will frequently choose to use buggy software (with known bugs), provided we understand the consequences of the bugs. We of course use buggy software with unknown bugs since that is the state of all of our software and indeed the state of human existence.

We argue that such a pessimistic view of software is justified. There is no system that is absolutely (in a Platonic sense) correct. There may be no faults in that the program currently satisfies its requirements, but as requirements change, the program changes. Thus the requirements were in error. This type of behavior is very typical of large systems. Changes (also called maintenance) are continually made to "correct" or "enhance" such systems. Eventually the accumulation of these changes becomes so great that the old system is retired and a new one constructed. This is similar to the Popperian refutation of an old theory and its replacement by a new theory.

As bugs are discovered and fixed, the computer system changes (or as some like to say, becomes more refined). Similarly, refutations of scientific theories lead to theory changes (or corrections). Many like to believe that scientific theories are truth (just as

many like to believe that the current software version is correct), but in fact they are human conjectures and subject to error as are other human constructs. The corrections or changes result in new or refined scientific theories just as corrections to computer systems result in new or refined computer systems.

The point of view expressed by Popper is in contrast to other philosophical theories of scientific discovery — most especially the various inductive theories. These theories take it as given that a correct theory exists and that the task of the scientist is to discover the theory by observing, through experiments, instances of the theory. Applying the analogy again, an initial computer system is built and the system is observed on both test data and actual data. As errors are discovered, the system is "improved" with ultimate convergence to the "correct" system. The Popperian believes that refutations lead to theory creations. The inductivist believes in the evolution of theories by refinement from refutations with the ultimate achievement of the correct theory. For an inductivist, the discovery of a refutation is a tragedy for this means that the current theory is not correct and the process of refinement must continue. For a Popperian, the discovery of a refutation is glorious for it means that a new and better theory can be discovered. The Popperian is not disheartened by the incorrectness of the existing theory for he believes that all theories are incorrect. Thus a Popperian is more likely to thoroughly test a scientific theory than an inductivist since refutations please the Popperian and displease the inductivist. The above points have not been entirely lost on the testing community. Glenford Myers [24] especially emphasizes the positive aspects of error discovery in his monograph on software testing.

The traditional, and practiced, view of system testing is more inductivist than Popperian. The result of an individual test is viewed as an instantiation (with the test values) of a "theorem" that reflects a partial (one point) proof of the program's correctness. If we only could test over the program's entire domain, we could discover if it were "correct" — that is we could logically infer the desired "theorem". The problem with this approach is that we generally cannot test over the program's entire domain. Thus we must test over a sample of the program's domain. How do we choose the sample? Our contention is that the correct choice of sample ("good" tests in the Popperian sense) can lead to far more information than random samples or samples based upon ad hoc and unproven criteria. We even claim that in some instances the correct choice of a test set can prove the mathematical correctness of a program.

It has been long noted by Popper and others [26] that testing over additional, related, values decreases the value of each additional test. This is expressed in the form of a rule by Polya "The verification of a new consequence counts more or less according as the new consequence differs more or less from the formerly verified consequence" (page 7, vol 2 of [26]). Unfortunately much of the testing methodology results in exactly this type of testing. It was this state of affairs in testing (and the resulting unfortunate implementation) that led Dijkstra to his famous aphorism that testing can "show the presence of bugs not their absence!" [11]. Like many aphorisms it is only half true. Our thesis is that the testing, if approached properly, can demonstrate the absence of bugs (where absence of bugs does not necessarily imply system correctness). This is especially true when a Popperian approach is taken. In that case since NO system is correct even Dijkstra's mathematical approach to software development is prone to error.

II. Outline of a Theory of Testing

What then do we mean by the absence of bugs? Absence of bugs becomes a relative term. That is, we may be able to assert for a properly tested system that if the system represents a function from a specified class of functions, then the system free of of errors. That is, absence of bugs depends upon a functional property of the system. In the worst case this means that the system does not err on the inputs for which it has been tested. A better situation is when the class of programs under test are known to have certain functional properties and that the test cases are known to be representative of that class of functions in that they characterize the functions under test. In such circumstances, if a program passes its tests positive information is gained. Either the program is correct or the program does not have the desired functional properties. The best that we can hope for is that the program is guarenteed to compute a function within the desired class of functions and the test cases are sufficient to refute the hypothesis that the program computes the its specified function. In other words, the test cases satisfy the criterion of being "good experiments". Testing techniques such as Cause-Effect Graphing [24] attempt to find such test cases.

We are generally unable to assert that a system is free of all errors. How are classes of errors typically described? A common approach is to split the problem of testing into classes determined by the information used in the testing process. If only functional information is used in the choice of tests, the testing is called *functional* testing or *black box* testing. If syntactic information is the primary information used in the choice of tests (i.e. that is information arising from the representation of the software system as a program), the method is called *white box* testing. Syntactic descriptions are the easiest to implement (after all, there is always an explicit syntactic description of the program under test). Thus mutation analysis [12], [16], path testing methods [20], and all other white box testing methods are methods that show the absence of bugs within syntactically describable error classes. The difficulty with this testing is that there is no clear understanding of the class of errors that will be uncovered by test data that meets the most common white box testing metrics. For example, how well tested is a program that has been tested on data that ensures that every statement in the program has been executed? Nobody knows. Some more recent white box metrics are more reasonable [29] and some analysis of the meaning of programs that pass tests based on those metrics can be made. However, we wish to concentrate on a functional testing approach. We will also argue that some white box testing methods can be incorporated into a functional testing approach.

Our notion of functional testing is similar, in some respects, to that described by Howden in [19]. The idea is to identify a class of functions $\mathcal{F}$ and a testing method to generate test data that will identify or distinguish functions within the class $\mathcal{F}$. The functions are usually described at the requirements or specification stage and the method to generate the required test data should be computationally feasible. An early example, due to Howden [19], is the class of real valued polynomials in one variable of fixed degree, say k. Then the values of the polynomials on $k+1$ test points is sufficient to distinguish the polynomials and hence constitute good test data. Howden's general approach actually combined white box and functional testing approaches. For each syntactic program element (statement type, procedure call, etc.) he identified a mathematical interpretation.

He then developed an algebra to combine program elements according to the allowable program construction rules and associated meanings with these combinations. Then by identifying likely errors at both the program element and construction stages he could construct a class of functions $\mathcal{F}$ from a plausible implementation of a system requirement. Data that would distinguish amongst the functions in $\mathcal{F}$ would be considered to be good test data.

Our theory borrows from Howden's theory, from notions of circuit testing (its original genesis), and from work in inductive inference. The basic idea is similar to Howden's fundamental notions. We wish to assess the quality of the testing of a software system. We first identify the class of functions in which we believe the function computed by our system lies and in which functions computed by erroreous implementations of our system lie. We then identify, if possible, a method for generating test data to distinguish amongst these functions. Finally we test the system on the the generated test data. Assuming that we can accomplish the above, we obtain theorems of the type:

> The program was correct on the test data intended to distinguish the program from functions in $\mathcal{F}$. Therefore the program is correct or the the function that it computes is outside the class $\mathcal{F}$.

A full proof of correctness would involve running correctly on the tests and then a proof that the function the program computed was in the class $\mathcal{F}$. This latter proof is generally the more difficult; in some cases, however, the proof can be trivial. This is so when the programming language is a restricted language and programs in the programming language allow only functions in $\mathcal{F}$ to be computed. For an example of this see [7]. We can also obtain partial results. For example, we may wish to test state transition activities of a communications protocol ignoring other tasks of the protocol. The abstracted system becomes a finite state transducer and test cases can be generated that will distinguish that transducer from a wide class of similar, but not identical, transducers.

Note that the above statement is a correctness statement and is as mathematically justified as programs that have been subjected to proofs of correctness. It is in this fashion that we assert that testing does more than just "show the presence of bugs, not their absence!".

The other two motivations for our work come from inductive inference and circuit testing. Circuit testing motivated our notion of distinguishability as a good criterion for test set construction. A suite of tests designed for a circuit is characterized as good provided that it distinguishes certain faults. If the circuit passes such tests, it is known not to contain any of those faults. Note this does not say that the circuit is correct, all that is asserted is that the circuit is not one of a set of boolean functions each of which exhibits faulty (for the circuit) behavior. Most white box testing methods [1] are based on similar principles. If, for example, a test suite is designed so that all statements in a program are executed for some data in the test suite, then the test suite distinguishes the correct program from incorrect programs where the error can be uncovered by a faulty statement execution. Unfortunately in many cases it is not at all clear what is being shown when a program passes test suites based upon such syntactic criteria. Finally, mutation analysis ([5], [16]) and metric based testing [29] use more sophisticated testing metrics based upon distinguishability criteria.

The inductive inference connection arose from an observation that the formal inductive inference process (as described in [2] and [14]) consists of the determination of objects from a class of objects (where each object is an arbitrary set) from finite data. Identifying functions with their graphs (which are sets of domain–range pairs) we noted that the determination of such objects is essentially the same process as distinguishing functions from a class of functions (note that our notion of inductive inference is to be viewed as distinct from the philosophical notion expoused by the inductivists described above). Pursuit of this observation resulted in a recursion theoretic theory of testing [8], connections between testing and inference from positive data [6], the development of the foundations of both a theory of testing and a theory of inductive inference [10], and a new notion of "pairwise testing" [9] akin to the notion of inference in the limit. The latter result completed our initial intuition that testing and inductive inference are essentially the same process. For each class of inferable functions there corresponds an identical class of testable functions and for each class of testable functions there corresponds an identical class of inferable functions. We will have more to say about these results once we establish the necessary definitions.

III. Definitions and Examples

As mentioned above, we view testing as a theory of distinguishability. In its most abstract form, we are interested in distinguishing amongst sets using queries about membership. Functions are represented by their graphs and the membership queries are simply queries regarding the domain–range values of the function. For further information on this abstract approach see [10]. We will restrict ourselves to functions in the rest of this paper. Without loss of generality, we also identify the domain with the range. Let D be a countable domain. Let $\mathcal{F}$ be a countable class $\{F_1, F_2, \ldots\}$ of functions (not necessarily total) mapping elements of D to elements of D. Let $\mathcal{F}_\sigma = \{\sigma_1, \sigma_2, \ldots\}$ be the set of finite segments of functions in $\mathcal{F}$ where σ is a finite segment of F provided that σ's graph is a finite subset of F's graph. We write $\sigma \prec F$ if σ is a finite segment of F. Let T map $\mathcal{F}$ to $\mathcal{F}_\sigma$ with the property that $T(F) \prec F$. We call T a testing agent and it generates the test cases from the functions under test. The functions under test may be presented abstractly as a countable class with certain properties or concretely as syntactic entities such as programs (examples are given below).

Definition 1: $\mathcal{F}$ is *testable in the limit* provided that $\forall i \forall j (j < i \Rightarrow T(F_i) \not\prec F_j)$.

This definition is actually too abstract for practical purposes. We normally require that that the class $\mathcal{F}$ have an effective presentation and that the function T be recursive and easy to compute. The presentation of $\mathcal{F}$ can be as programs (the functions are the functions computed by the programs), as mathematical descriptions (e.g. polynomials), or as expressions in some specification language. The function T should be very easy to compute given the description of a function F from F. We now present some examples of testable (and untestable) classes of functions.

Example 1: The class of polynomials of fixed degree k, $k \geq 0$, is testable. The test set consists of any fixed set of $k+1$ points. This set is testable in a strong sense in that the choice of domain elements in the test set is independent of the function under test.

Example 2: The class of editing programs introduced by Nix [25] is testable in the limit. This was proven in [7] by showing that corresponding to any such program there was a set of test data that uniquely identified that program from all other editing programs. The test data increased in size as the number of variables in the editing program increased. The computational complexity of T is at most linear in the size of the editing program. The testing is independent of the ordering of the class. This class is a simple class of editing functions that are representative of transformational programs known as filters (the finite state transducers below are another such example). We are working on a more general characterization of filter programs.

Example 3: Let F be an arbitrary total function mapping the positive integers to the positive integers. Let $\mathcal{F}$ be the set of all finite segments of F. $\mathcal{F}$ is testable in the limit using the identity function as the function T.

Example 4: Consider the class $\mathcal{F}$ from example 3 along with the function F from example 3. This class is not testable. Consider any enumeration of the above class. Let j be the index of F in that enumeration. Any finite segment with index higher than j is mapped by T to some finite segment of F. This does not satisfy the testability criterion.

Example 5: The class of deterministic finite state transducers is testable in the limit. This was shown in [6] and we can argue informally as follows. The test function T on transducer F_j produces enough inputs to distinguish F_j from earlier transducers. The decidability of equivalence of finite state transducers makes this process effective and reasonably efficient. More restricted classes of deterministic finite state transducers lead to more efficient test function algorithms. Note that in this example structural information (number of states, type of transducer, etc.) could be very useful in the definition of the testing function. The definition of T given above would not be of much practical use. The class of finite state transducers is also interesting for such transducers form the basis for a number of design languages ([18], [23]) and are also frequently used as informal descriptions for the actions of interactive and real-time systems.

Example 6: Any class of sets that is inferable (learnable) in the limit [2] is testable in the limit. The notion of inferable classes has been well-studied in the mathematics community and, in a more *ad hoc* fashion, the artificial intelligence community. A result in [9] shows that the notions are essentially the same. Thus the classes of inferable sets discussed in [10] are all also testable classes of sets. For the most part these sets are not of practical interest but they provide a foundation upon which a theory that will apply to practical test sets can be built.

IV. Games and Complexity issues

As the above examples illustrate, there are a number of well defined classes of testable sets and functions. The complexity results for most of these classes are scattered. In some cases, very quick (even constant time) algorithms can be given to generate distinguishing test data. In other cases the problem is computationally infeasible. We initiated a study of the complexity of testing a simple class of functions. These functions were functions from a finite set of n elements to a set of 2 elements. The number of distinct functions in this class is at most 2^n and each function can be represented by a boolean vector of length n. A set of m such functions can be represented by an m row table of n columns.

The problem of determining whether for any such finite function f in some class of size m there are k domain–range pairs that uniquely identify that function from all the rest of the functions is $\mathcal{NP}$-complete in m and n. Recalling that the best know algorithms for $\mathcal{NP}$-complete problems are exponential, we see that even very simple testing situations may be unexpectedly complex. For more details see [10].

The notion of testing used in the above complexity result is a stronger notion than testing in the limit. In the stronger notion we insist that the testing function distinguish the function under test from all the functions in the class. This stronger notion also has an interesting game formulation originally discussed in [10]. The game is a two player game. Player 1 initially picks a function F from a class of functions $\mathcal{F}$. Player 1 and Player 2 then alternate turns. Player 1 plays a domain–range pair. Player 2 plays a function from $\mathcal{F}$ that differs from F but is still consistent with Player 1's choices. Player 1 wins the game provided that he forces Player 2 to play F. The class $\mathcal{F}$ is testable provided that Player 1 has a winning strategy for every function in $\mathcal{F}$. That is, no matter what function F chosen by Player 1 and no matter what functions are played by Player 2, Player 1 forces Player 2 to play F.

While the notion of a testing game is a bit frivilous, it does provide a convenient language with which to talk about testing. The complexity result also has some potential applications. In testing combinatorial circuits for particular faults (stuck-at faults for example) we are essentially testing a finite function trying to distinguish it from all finite functions exhibiting a fault. The testing complexity result says that this problem may be very difficult. Also investigated were some average cost complexity results and some minimum and maximum complexity results.

V. Software Safety and a Protocol Experiment

The issue of software system safety is one that may be separate from functional testing [22] but our approach to functional testing can be adapted to discuss certain software system safety issues. Software safety is roughly defined as the prevention and discovery and correction of any errors in a system controlled or monitored by software that could lead to serious injury, death, or substantial loss of property. As defined, software safety may have nothing to do with the functional correctness of software. For example, if an engine control program for an automobile never works (thereby causing the automobile never to run) the program might be viewed as safe, though it certainly is not functionally correct.

One technique frequently used in software safety is the technique of fault tree analysis [21]. Fault tree analysis consists of identifying unsafe states and working backward from those states in order to identify software or hardware "locks" that could be incorporated into the system to ensure that the unsafe states cannot be reached. If this is not possible, then a redesign is in order or a warning should be issued when the system is released that unsafe states could occur under certain circumstances.

The essentials of the fault tree analysis are in identifying the unsafe states and blocking access to those states. The blocking of access is essentially restricting the program to compute a function lying outside some specified class of functions. Recalling our testing paradigm, we search for test data that will show that a program is correct (if it lies

within a specified class of functions) or that will demonstrate that the program computes a functions that is outside the class. Software safety analysis is the process of ensuring that a program cannot compute a function lying outside a particular class of functions — that is a function that reaches an unsafe state. Our testing paradigm involves both the generation of test data for classes of functions and restrictions on programs to ensure that they lie within specified function classes; software safety issues reside in the latter category.

As a final example regarding the utility of our testing theory, we report on a method of generating test data for communications protocol testing. The National Bureau of Standards (NBS) has set up a protocol testing laboratory for testing designs of ISO communications protocols. As part of this effort NBS has built a compiler for Estelle, a specification and simulation language based upon finite state transducers and Pascal, and an environment to execute test sequences for communication protocols. As reported in [13] there are several test generation methods that are used for finite state transducers in the context of communications. They range from simple statement coverage techniques (ensure that every state is reached) to branch coverage techniques (ensure that every transition is taken) to behavior uncovering techniques (identical to test sequences required for finite state transducer testing). The latter technique includes the first two as subcases. This latter technique was first used for circuits [15]. Our independent analysis of finite state transducers, however, has led to potentially stronger results. For subclasses of these transducers (which may be quite practical) we can substantially reduce the number of required test cases and still retain the testability condition. We can also state, for general finite state transducers, that this method is "best" in that fewer test cases will lead to the non-distinguishability of some functions [6]. Finally, our method leads to a natural technique for test case generation for finite state transducers.

VI. Conclusions

We began this paper by comparing the process of testing a software system to the process of the development of a scientific theory. We noted some similarities in the two processes and then outlined the development of a positive theory of testing. The theory is positive in the sense that for carefully chosen test data, after the testing is completed successfully, either the software is provably correct or the software computes a function outside a specified class of functions. This is in contrast to the more prevalent negative view of testing which, because of the way the test sets are chosen, is never ending and never yields a statement about the mathematical correctness of the software.

We illustrated our theory with a number of examples of both a mathematical nature and of a somewhat more pragmatic nature. These examples were intended to impart some of the flavor of the theory and to illustrate some of its potential applications. Our research has been mostly mathematical until recently and thus our collection of pragmatic examples is small. We expect it to expand considerably. There are a large number of interesting areas left to explore. Some of these are discussed below.

We have preliminary results that indicate testing, as defined above, and proof of correctness can be viewed as the same process. The theorem states, roughly, that we can define a proof system whose test cases are essentially all the steps necessary in the formal correctness proof of the program. Thus passing the tests validates individual proof steps

and the tests are chosen so that if all individual proof steps are satisfied, the correctness proof is checked. The opposite direction follows by definition — if the program is proven correct, then the program operates correctly on all data satisfying the program's precondition. While this result is mathematically satisfying, it does not really aid in determining the correctness of software. Towards that end, we are investigating connections with automated debugging [27] that arise from the inference of provably correct Prolog programs. The work of Bouge is also relevent in this context ([3], [4]).

The second major area of interest is the connection between probablistic inference (or learnablity theory) and testing. Hamlet [17] has suggested connections between learnablity theory and software reliability. His very interesting results give an estimate of the probability of a program being correct (no undiscovered errors) after a given number of correct test executions. His probabilistic analysis rests heavily on Valiant's [28] probabilistic theory of learnability. We believe that his insight is sound and believe that our connections of inference and testing will carry over to probabilistic inference and some form of probabilistic testing or reliability. We expect to get somewhat more positive results than Hamlet since our underlying probablistic assumptions will be more optimistic. However, we have not had time to explore this topic in detail.

VII. Acknowledgements

Much of the work discussed in this paper was done in collaboration with others. I would particularly like to thank Carl Smith and Rick Statman. Carl taught me most of the inductive inference I know and collaborated with me on the first paper connecting inductive inference and testing. Rick made major contributions to the foundational theories of testing and inductive inference.

References

1. Adrion, W. R., Branstad, M. A., and Cherniavsky, J. C. Validation, verification, and testing of computer software. *Computing Surveys 14* (1982), 159–192.
2. Angluin, D. and Smith, C. H. Inductive inference: theory and methods. *Computing Surveys 15* (1983), 237–269.
3. Bouge, L. A contribution to the theory of program testing. *Theoretical Computer Science 37* (1985), 151–181.
4. Bouge, L., Choquet, N., Fribourg, L., and Gaudel, M. C. Application of PROLOG to test sets generation from algebraic specifications. *Lecture Notes in Computer Science 186* (1985), 261–275.
5. Budd, T. A., Lipton, R. J., Demillo, R. A., and Sayward, F. G. Mutation analysis. *Papers on Program Testing* (1979), 29–57.
6. Cherniavsky, J. C. and Smith, C. H. Using telltales in developing program test sets. Technical Report 4, Department of Computer Science, Georgetown University, Washington D.C., 1986.
7. Cherniavsky, J. C. and Smith, C. H. A theory of program testing with applications. In *Proceedings of the IEEE Workshop on Software Testing*, Banff, Canada, 1986.

8. Cherniavsky, J. C. and Smith, C. H. A recursion theoretic approach to program testing. *IEEE Transactions on Software Engineering 13,7* (1987), 777–784.

9. Cherniavsky, J. C., Smith, C. H., and Statman, R. Testing in the limit. Technical Report 6, Department of Computer Science, Georgetown University, Washington D.C., 1987.

10. Cherniavsky, J. C., Statman, R., and Velauthapillai, M. Testing and inductive inference: abstract approaches. Technical Report 5, Department of Computer Science, Georgetown University, Washington D.C., 1987.

11. Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R. *Structured Programming.* Academic Press, New York, 1972.

12. Demillo, R. A., Lipton, R. J., and Sayward, F. G. Hints on test data selection: help for the practicing programmer. *Computer 11* (1978), 34–41.

13. Favreau, J-P. and Linn, R. J. Automatic generation of test scenario skeletons from protocol specifications written in Estelle. In *Protocol Specification, Testing, and Verification*, G. V. Bochmann, Ed., 6, Elsevier Science Publishers B.V. (North Holland), Netherlands, 1987.

14. Gold, E. M. Language identification in the limit. *Information and Control 10* (1967), 447–474.

15. Gonec, G. A method for the design of fault detection experiments. *IEEE Transactions on Computers* (1970).

16. Hamlet, R. G. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engeneering SE3* (1977), 279–290.

17. Hamlet, R. G. Probable correctness theory. *Information Processing Letters 25* (1987), 17–25.

18. Harel, D. Statecharts: a visual formalism for complex systems. *Science of Computer Programming 8* (1987).

19. Howden, W. E. The theory and practice of functional testing. *IEEE Software 2,5* (1985), 6–17.

20. Huang, J. C. An approach to program testing. *ACM Computing Surverys 7* (1974), 113–128.

21. Leveson, N. C. and Stolzy, J. L. Safety analysis using Petri nets. *IEEE Transactions on Software Engineering 13,3* (1987), 386–397.

22. Leveson, N. G. Software safety: why, what, and how. *ACM Computing Surveys 18,2* (1986), 125–163.

23. Linn, R. J. The features and facilities of Estelle. In *Protocol Specification, Testing, and Verification*, M. Diaz, Ed., 5, Elsevier Science Publishers B.V. (North Holland), Netherlands, 1986.

24. Myers, G. J. *The Art of Software Testing.* John Wiley and Sons, New York, 1979.

25. Nix, R. Editing by example. *ACM Transactions on Programming Languages 7,4* (1985), 600–621.

26. Polya, G. *Mathematics and Plausible Reasoning, Vol 1 and Vol 2.* Princeton University Press, Princeton, 1954.

27. Shapiro, E. Y. *Algorithmic Program Debugging.* MIT Press, Cambridge, Massachusetts, 1983.

28. Valiant, L. A Theory of the Learnable. *Communications of the ACM 27,* 11 (1984), 1134–1142.

29. Weyuker, E. and Davis, M. A formal notion of program-based test data adequacy. *Information and Control 56* (1983), 52–71.

A Model for Assessing Code-based Testing Techniques

Larry J. Morell
Department of Computer Science
College of William and Mary
Williamsburg, Virginia 23185

A theory of fault-based program testing is defined and explained. Testing is *fault-based* when it seeks to demonstrate that prescribed faults are not in a program. It is assumed here that a program can only be incorrect in a limited fashion specified by associating *alternate expressions* with program expressions. Classes of alternate expressions can be infinite. Substitution of an alternate expression for a program expression yields an alternate program that is potentially correct. The goal of fault-based testing is to produce a test set that differentiates the program from each of its alternates.

A particular form of fault-based testing based on symbolic execution is presented. In *symbolic testing* program expressions are replaced by symbolic alernatives that represent classes of alternate expressions. The output from the system is an expression in terms of the input and the symbolic alternative. Equating this with the output from the original program yields a *propagation equation* whose solutions determine those alternatives which are not differentiated by this test.

Larry Morell received his Ph.D. from the University of Maryland in 1983 in the area of program testing. His current research efforts are in program testing, reliability, validation of expert systems, and specification of software tools.

1. Introduction. Despite its many deficiencies, program testing has remained the premier method of program verification to date and is likely to remain so for some time to come. What makes this so puzzling is that testing seems to work when many of the fundamental theoretical results concerning it are negative. It is well-known, for instance, that it is impossible to prove an arbitrary program correct by testing it over a finite subset of its input space. Why then are we so inclined to believe our program correct after so little testing?

One possible explanation is that we believe that the information content in a simple execution is much more than the trivial input-output pair that the test provides us. People speak of warm feelings and the like when their program finally succeeds on some trivial tests. We are willing to extropolate well beyond the justified limits when it comes to believing that the next test case will succeed. Yet we know that theory tells us that a program can always be wrong on the very next input and that results from past testing are very difficult to generalize. We rapidly become schizophrenic if we dwell too long on the subject.

This schizophrenia can be somewhat relieved if it can be shown that some meaningful information content can be associated with each execution of the program, beyond the mere fact of its success (or failure). Ideally such information content should range from zero to one, where zero indicates that no information has yet been gained about the correctness of the program and one indicates that the program is correct. Intermediate values indicate that certain features of the program are correct, while others are yet to be determined correct.

This paper provides such a measure of the information content of an execution, namely the set of all faults that it demonstrates are not present in the program. Note that a test set with no data points (the null test) has zero information content, and the test set which includes the entire input domain (the exhaustive test) has full information content. What is interesting is that each successful execution demonstrates that many faults could not have been made in the program, i.e., exactly those faults whose presence would have caused a failure on the given input. The set of faults thus *eliminated* provide a measure of the power of a given test. If it can be assumed that the program is erroneous in a limited way (as suggested in [DeM78]) then a test demonstrates correctness when it shows the potential is not realized. Hence, under this assumption, it is possible for a non-exhaustive test to eliminate all potential fault, and thus prove the program correct.

A theory of testing is presented here based upon the notion of demonstrating that certain faults are not in the program. In the next section the role of a testing theory is explored and the particular theory called fault-based testing is developed in the following two sections. The use of the theory as a meta-theory is explored in the last section.

2. The role of a testing theory. A theory of testing is a systematic attempt to provide a basis for investigating the phenomena of program testing. It should be both descriptive and prescriptive. As description, theory explains results achieved; as prescription, theory indicates results that can be achieved and directions in which program testing should move. In this section several questions are discussed that a theory of program testing should address. For each of the questions a brief answer is given according to the direction taken in this paper, namely one of error-based testing.

There are three fundamental, intertwined questions that must be addressed by any testing theory:

(1) When do I stop testing?

(2) What is adequate test data?

(3) How do I select a testing strategy?

It is difficult to answer one without at least partially answering the others. An all-too-frequent criterion for stopping testing is the exhaustion of time, money, or other resources. To see the ludicrous nature of this answer, consider using this answer for a different question: When do I stop coding? No one would consider coding complete just because resources have vanished![1] This is because there are recognizable criteria for code completeness independent of the environment in which it occurs.

What are recognizable criteria for completeness of testing? One can imagine many answers: when the program is correct, when the program has attained a certain level of reliability, when a given coverage of the program or specification has been accomplished, etc. Note that these answers approach the notion of completeness from many different angles. Correctness focuses on the relationship between the program and its specification, reliability focuses on estimating the failure frequency of the program, and coverage focuses on breadth of the test. Translating these criteria into assertions about the test data yields answers to the second question above. One role of theory at this point is to precisely define these criteria and to determine when they can be achieved. But theory can and should do more than this. It can provide a framework into which all such answers can be understood, evaluated, and compared. By providing such a framework, it acts in the role of a meta-theory: a theory about theory. This paper pursues one such meta-theory that is applicable to code-based testing schemes.

Establishment of test data adequacy criteria leads to the third question: how does one implement a scheme to achieve such adequacy? The interwining of the questions now becomes obvious, because the selection of the testing scheme always raises the first two questions and is driven by the answer to the first two questions. Should a testing scheme be selected without considering termination criteria and test data adequacy, testing will be done, but its quality will be very difficult to judge. Testing theory should guide the selection of strategies with provable connection to attaining adequate test data.

There are, of course, many question that cannot be fully answered by a theory, especially those which overlap with pragmatic implementation issues dealing with human factors, available resources, training, etc. What is the theoretically best possible testing, may not be practical in a particular environment, and thus cost/benefit tradeoffs must be considered. For example, a particular testing technique may theoretically isolate many program failures, but if the identification of these failures requires a person to spend hours inspecting output, the value of such a technique is reduced. Empirical research is therefore needed to evaluate the utility of of proposed techniques. But such research is distinct from the theory which strives to determine what can be accomplished, given adequate resources. This paper provides a framework for understanding and extending current testing practice in code-based testing.

2.1. An error-based approach to testing theory. The means taken here of assessing the quality of tests is to compare them on the basis of how many errors they demonstrate are not in the program, i.e., on how many errors each test *eliminates*. This choice is intuitively satisfying since the highest quality test by this approach is one which eliminates all errors, implying the program is correct. The quality of a testing strategy is therefore determined solely by what errors it is guaranteed to eliminate.

Testing is *error-based* when it is performed with the aim of eliminating errors. This notion was introduced by Weyuker and Ostrand [Wey80]. In their approach, a program's input is partitioned into a set of disjoint classes called *subdomains*. These subdomains are then shown to be *revealing* for certain errors; i.e., if any member of a subdomain reveal a designated error, then all members of the subdomain reveal the error. In this manner an error is eliminated if the program is correct for an input from a subdomain that is revealing for that error. Only specified errors are eliminated.

[1] You may be bankrupt, but not complete!

The difficulty with Weyuker's and Ostrand's approach is that it does not specify the means of designating errors and hence it is impossible to systematically apply their ideas or to prove any properties of what they envision as error-based testing. The means of eliminating errors is therefore explained by example rather than by providing a method to follow independent of the error categories. They do indicate, however, that for every supposed error in the program, test data must be developed that eliminates the error. In order to do this a proof must be given that certain inputs eliminate particular errors. What is unclear is whether there is any general technique for doing such proofs or what exactly constitutes a proof. There is also no comment on whether such proofs are always possible, and if not, when they are possible. It is therefore impossible to determine for what categories of errors the proofs can be automated.

Testing is *fault-based* when it seeks to demonstrate that certain faults have not occurred in the program text. Fault-based testing is thus a proper subset of error-based testing. This paper develops a fault-based testing meta-theory that

(1) defines fault-based testing,

(2) provides strategies by which fault-based testing can be applied, and

(3) proves properties about the strategies concerning what faults can and cannot be eliminated.

The theory defines the mechanism whereby the faults can be defined, and once defined, how they are eliminated. It is a meta-theory in the sense that it applies once the fault categories have been chosen, and is therefore independent of any particular class of faults.

This approach to testing may be considered as a "fault sieve." Figure 1 illustrates the concept. A program is "poured" through a testing strategy, which sifts out certain faults from the program. This is accomplished by using the strategy to generate a test set which is then applied to the program. Those points for which the program is unsuccessful are identified, and the program is corrected. Eventually the program produces the correct output for each member of the test set. At this point, all that is known is that the program can successfully execute the test data; but how adequate is this data?

To judge the quality of the test data requires a paradigm shift: the program and its test data become the sieve and a set of fault classes are "poured" through this filter. The straining that occurs at this point is the identification of faults that could *not* be in the program, rather than the removal of faults that are in the program. This straining process is called *potential fault elimination* or *elimination* for short. Test data is considered more adequate when it eliminates more faults. Thus, *ideal* test data eliminates all faults and proves the program correct.

The difference between this approach and conventional testing can best be understood by contrasting it with more "traditional" goals of testing. Meyers, for instance, states that "Testing is the process of executing a program with the intent of finding errors." [Mey79]. Further, he says "since a test case that does not find an error is largely a waste of time and money, the descriptor 'successful' seems inappropriate"[Mey79]. To reply to the first quotation: attempting to "find errors" by testing is futile. If this could in fact be done one could systematically find each error and correct it, and ultimately produce a correct program. Since it well known that this is not possible ([Goo75] and [How76]) this advice is non-effective: it establishes goals that are not achievable in practice.

The second quotation can be interpreted as saying that there is no information content in a successful test. One of the goals here is to show that this assertion is false; each successful execution contains in it a wealth of information for deducing what faults could not be in a program.

To see the fallacy in the no-information-content perspective, consider the following analogy. Suppose an insurance company requires a physical examination of potential clients. The participating physician runs a battery of tests to ensure that certain dangerous complications are not present. Analysis of the test results allows the physician to certify that none of the most dangerous

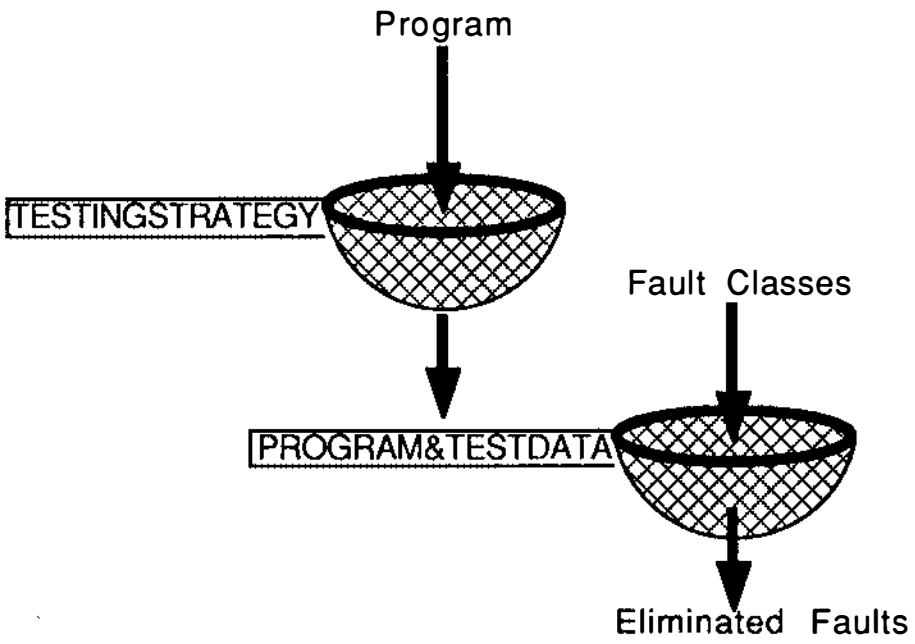


Figure 1

complications are present. If the physician were to follow traditional testing lore and disregard every test that did not indicate the client was sick, the healthy client would be subjected to more and more tests. There is no guideline as to when to stop testing, because the accumulated information content in successful tests is zero (according to the traditional view).

The implicit assumption made in the analogy, of course, is that the test methods serve double duty: they are both reliable for detecting certain diseases in people and for demonstrating that certain diseases are not present. Thus, passing the test is indicative of the absence of certain diseases. For the same results to apply to program testing, a method must be developed for deducing the absence of certain faults in computer programs. The model proposed below treats successful tests as indicative of the non-presence of prescribed faults. The model thus recognizes the information content in a successful execution.

3. Fault-based testing. The appeal of fault-based testing is that it is closely related to the alternate way of showing program correctness given in the previous section: show that the program is not incorrect. If a program has limited potential for being incorrect, then a test set demonstrates correctness when it shows the potential is not realized.

It is first necessary to distinguish among the various forms of incorrectness. The IEEE conventions [IEE83] are adopted here. An *error* is a mental mistake by a programmer or designer. It may result in textual problem with the code called a *fault*. A *failure* occurs when a program computes an incorrect output for an input in the domain of the specification. For example, a programmer may

make an error in analyzing an algorithm. This may result in writing a code fault, say a reference to an incorrect variable. When the program is executed it may fail for some inputs.

The means of specifying incorrectness taken here is to define potential faults for program constructs. Thus the subject is *fault-based testing.*

3.1. Basic definitions. The fundamental components in this testing theory are programs, program functions, specifications, and test sets. In order to understand the relationships among these, the following definitions are given. A *relation* is a set of ordered pairs. If R is a relation then its *domain* is $\{x \mid (x,y) \in R\}$ and is written as dom(R). Every program P defines a relation called its *program relation* defined as {(x,y)| program P on input x halts with output y}.

Since programs and their corresponding relations are frequently discussed together, it is important to adopt a notation that distinguishes the two. [Lin79] suggests the following: If P is a program, then its program relation is denoted by [P]. If $(x,y) \in [P]$ and $(x,z) \in [P] \Rightarrow y = z$ then the relation is a function and the program is called *deterministic*. The theory developed below is for deterministic programs, but can be extended for non-deterministic programs. Since [P] is a function, [P](x) is the output P computes on input x. If no such output exists, then [P] is *undefined* at x and this is written as $[P](x)\uparrow$; otherwise [P] is defined on x which is written as $[P](x)\downarrow$.

A *specification* is a string that denotes a recursive relation with a recursive domain. The denoted relation is called the *specified relation*. For specification S, the specified relation is also written as [S]. Requiring the domain of the specified relation to be recursive avoids the awkward circumstance of not knowing whether a program is supposed to halt on a given input. Requiring the specified relation to be recursive, enables the specification to act as an *oracle,* a decision procedure which can decide whether or not any proposed input-output pair is in the specified relation. Both properties are necessary since neither one implies the other [Mor84].

An *arena* is a 3-tuple <P,S,D> where P is a program, S is a specification, and $D \subseteq dom([S])$. By default the third component is the entire domain of [S]. A *test set* T, for an arena G = <P,S,D> is a subset of D. T is *halting* if and only if T subset dom([P]). T is finite unless otherwise stated. D therefore indicates the set of inputs from which test sets can be chosen.

The central concept relating programs to specifications is that of correctness. A program P is *correct* with respect to a specification S if and only if $dom([P] \cap [S]) = dom([S])$ [Mil80]. Thus, a program is correct with respect to a specification if and only if it computes results for the entire domain of interest, and all the results it computes are as specified.

Definition A *location* in a program denotes some expression of the program. "Expression" is necessarily vague since it is language dependent. An *alternate expression* (an *alternative*) is an expression f which can be legally substituted for the expression designated by *l*. The resulting program is called an *alternate program* and is denoted by P_f^l. $P_{f,g}^{l_i,l_j}$ denotes the *multiple* alternate program formed by substituting expressions f and g in locations l_i and l_j, respectively. A set of alternatives for an expression is called an *alternative set*. An alternative set always contains the original expression unless otherwise stated.

An alternative set is used to represent potential faults in a program. It may be defined by enumeration, or by giving an algorithm by which the members may be generated. The method of description will usually be an English phrase. For instance, rather than giving an algorithm for generating all prime numbers, the phrase "all prime numbers" would be used. However, if any doubt may arise, the algorithm is given.

Definition A *fault-based arena* is a 5-tuple, E = <P,S,D,L,A>, where <P,S,D> is an arena, $L = (l_1, l_2,$

... , l_n) is an n-tuple of locations in P, and A = (A_1, A_2,...,A_n) where each A_i is an alternative set associated with location l_i, $1 \le i \le n$. If each alternative set is finite then E is called *bounded* else it is *unbounded.* The set of all alternate programs for E comprises those programs generated by substituting any alternative in its respective location in the program P. It is denoted by P_E. The program P is called the *original* program. When discussing an expression at a particular location, the expression is called the *designated expression.* By convention P_E contains the original program P.

Definition A fault-based arena E = <P,S,D,L,A> is *alternate-sufficient* if and only if $\exists$ R ε P_E such that R is correct with respect to S.

Alternate-sufficient captures the idea that at least one alternate program is correct. With it testing can sometimes demonstrate correctness. Clearly, there are fault-based arenas which are not alternate sufficient. It is more interesting to ask whether this is a decidable property.

Theorem It is undecidable whether an arbitrary fault-based arena is alternate sufficient.
Proof Consider the fault-based arena E = <P,S,D,ϕ,ϕ>. Deciding whether or not this arena is alternate-sufficient is equivalent to deciding whether P is correct with respect to S. This is, of course, undecidable.

The theorem shows that fault-based testing must restrict the alternative expressions so that testing can do something "useful". Testing is useful when it shows that the specified alternatives could not have been substituted without producing either an equivalent program or one which fails. This concept of usefulness is embodied in the formal notion of differentiation:

Definition For a program P and x ε dom([P]), x *differentiates* P from a program R if and only if

[P](x) $\neq$ [R](x), or
[R] is not defined for x, or

For a program P and a set T $\subseteq$ dom([P]), T *differentiates* P from a program R if and only if there is an x in T such that x differentiates P from R. For an arbitrary set of programs **R**, program P, and set T, T *differentiates* P from **R** if and only if for each program R in **R** not equivalent to P, T differentiates P from R.

The goal of fault-based testing is to determine for each alternate program whether it is equivalent to the original or different. This is achieved by a series of intermediate stages. In the first stage a successful test set is produced for P and in subsequent stages the alternate programs are separated into two classes, programs which have been eliminated and those which have not. An alternate program is *eliminated* if and only if an input has been produced that differentiates the original program from it or it is proven equivalent to the original program. In either case, the eliminated programs are no longer of any interest since they are either incorrect or equivalent to the original. The concept of elimination provides a convenient means of describing those alternate programs which deserve further attention, i.e., those programs that agree with the original program on the test set yet are not equivalent to the original. Fault-based testing is completed when all alternate programs are eliminated.

3.2. Differentiating alternate programs. Fault-based testing for an fault-based arena E = <P,S,D,L,A> involves two steps. First, E must be alternate-sufficient. For example, it may be known that a certain library program exists and is correct (shown, perhaps, by formal proof), but its name has been forgotten. A test set can be developed which selects the desired program from the library. Another example is when it it known that the specification can be implemented by a program from a given class, as in the case of algebraic program testing[How78]. In general, however, E is merely

assumed to be alternate-sufficient. Second, a test set must be constructed that either fails or differentiates P from P_E. This aspect of fault-based testing is discussed below.

Definition A fault-based arena E = <P,S,D,L,A> is *finite* if and only if there is a finite test set T for E such that T differentiates P from P_E. If no such test set exists, then E is *infinite*

Theorem Every bounded fault-based arena is finite.
Proof The test set need only include one point to differentiate each non-equivalent alternate program. In a bounded fault-based arena, there are a finitely many locations and alternatives, hence there are a finite number of alternate programs. Therefore, the arena is finite.

Fault-based testing for a bounded fault-based arena is called *mutation testing* [DeM78]. The assumption that the arena is alternate-sufficient is called the *competent programmer hypothesis.* This states that a competent programmer will write a program that is syntacticly close to the correct program, i.e. that the program will fall in a restricted class of programs, one of which is correct. This is similar to the assumption of alternate-sufficiency. Since the mutation testing arena is bounded, by the above theorem it is finite. Thus, all alternate programs can be generated. The set of alternate programs is unfortunately huge, and the undecidable program-equivalence problem remains.

The limitation of mutation testing to bounded fault-based arenas is quite restrictive. It is certainly desirable to extend fault-based testing to unbounded fault-based arenas. This, however, cannot be done in general since an unbounded fault-based arena can require an infinite test set to differentiate all the alternate programs [Mor84]. Fortunately, this is not always the case.

Theorem There are unbounded fault-based arenas which are finite.
Proof An example will suffice. Consider:

```
proc constant
print 5
endproc
```

Suppose the alternative expressions are "any constant" and the location is that of the constant 5. Then the infinite set of alternate programs is the set of all constant programs. Any non-empty finite test set differentiates the original program from all alternate programs.

A second problem with mutation testing relates to the phenomenal number of mutants that can be generated for even a simple program. [DeM78] assumes that simple faults are linked to complex faults in such a way that a test set which differentiates a program with simple faults, differentiates programs containing complex faults. This assumption (dubbed the "coupling effect") reduces the number of alternate programs that must be generated. It is not verifiable since "simple" and "complex" are not precisely defined. A precise, albeit restricted, definition of coupling is discussed later in this paper.

5. Symbolic testing. This section presents a fault-based testing strategy called *symbolic testing.* This strategy underlies the fault sieve style of verification suggested earlier. It is first assumed that the program is incorrect in some prescribed way. This assumption is then contradicted by evidence derived from symbolic execution of the program. Since the arena is assumed to be alternate-sufficient, the original program must be correct.

The model described below formalizes this process of deducing the absence of potential faults. Each successful test may add to the body of already eliminated faults, thus increasing the information content of the test. Termination of testing can then be defined as the point when all "reasonable" faults

have been eliminated. Symbolic testing is now formalized and illustrated.

4.1. Motivation and basic definitions. Symbolic execution[Cla77] underlies the idea of symbolic testing. The forte of symbolic execution is its ability to model infinitely many executions by a single symbolic execution. Historically this capability has been used to model the parallel execution of one program on an infinite set of inputs. This is accomplished by using a symbolic input to represent all inputs which follow a given path. In such a system, execution proceeds as usual until an input statement is reached. A symbolic input is then obtained and stored as the value of the appropriate variable. Whenever the variable is referenced, its symbolic value is introduced into the computation. This can result in a symbolic expression being stored in another variable. For example, in

```
read x
y := x + 1
print y
```

if the symbolic input **x** is read, then the value of y on output is **x + 1**. A symbolic input represents all inputs that follow a particular path. A symbolic execution system therefore determines the function computed by any given path by expressing the output in terms of the symbolic input.

Definition A *symbolic arena* S = <P,S,D,L,A> is a fault-based arena <P′,S,D,L,A> augmented in the following ways:

(1) dom([P]) = dom([P′]) $\cup$ an infinite set of *symbolic inputs.*

(2) P = P′, except for the replacement of one expression in P′ with an undeclared variable. Referencing such a variable introduces a unique *symbolic alternative* into the computation.

Definition A *symbolic state* of a program is an ordered pair (n,v) where n is a statement number of P and v is a mapping from variables to values

v: var --> value

where value may be an arbitrary expression involving numeric and symbolic values.

Symbolic testing allows representation of another infinite class of executions: that of an infinite set of alternate programs. This is achieved by creating a program called a *transformation* which represents all alternate programs induced by an alternative set. This transformation is created by replacing a designated expression with a symbolic alternative. The symbolic alternative "stands in" for members of the alternative set. The transformation is then symbolically executed along a chosen path. For example, in

```
read x
y := x + F
print y
```

the symbolic alternative F may represent all possible constant substitutions. For input **5** the output is **5 + F**. Thus, for a given (non-symbolic) input a program containing a symbolic alternative determines the output computed by all represented alternate programs.

Definition Let S = <P,S,D,L,A> be a symbolic arena. For location l in L and variable F not in P, P_F^l is called a *transformation* of P.

A transformation represents a class of alternate programs; it is not an alternate program itself. The transformation P_F^l represents all alternate programs P_e^l, for all e in the alternative set.

If the symbolic form of the computation for a class of alternate programs can be determined for an input x, then this form can be equated to the computation of the actual program, yielding a *propagation equation.* A propagation equation involves both the symbolic fault and actual or symbolic values.

Definition For x ε dom([P]), a *propagation equation* for transformation P_F^I is

$[P](x) = [P_F^I](x)$

If x is a symbolic input then the equation is *general* otherwise it is *specific.*

4.2. Examples. For example, consider the following program to swap two integers:

```
proc name
int x,y
read x read y
x := x + y
y := x - y
x := x - y
print x
print y
endproc
```

which produces on input 2,3:

```
x: 3
y: 2
```

Suppose that the reference to y in the first assignment statement is incorrect. Replacing this expression with symbolic alternative F yields the program

```
proc name
int x,y
read x,y
x := x + F
y := x - y
x := x - y
print x
print y
endproc
```

Re-executing the program for the same input (2,3) produces the symbolic output:

```
x: 2 + F - (2 + F - 3)
y: 2 + F - 3
```

The output encodes the functions computed by all the alternate programs represented by the (unspecified) alternative set. One propagation equation is produced by equating the two values computed for y:

$2 + F - 3 = 2$

This simplifies to F = 3. Thus, {3} is the solution set to the propagation equation.

The symbolic alternate in the propagation equation serves a dual role, as a free variable over the value space of the designated and as a free variable over the alternate set. As a free variable over the value space, the symbolic alternate specifies those values which, if substituted for the designated expression, would yield alternate programs that are not differentiated by the input value. In this case

substituting 3 for y produces an alternate program that is not differentiated by the input (2,3).

Once this solution set has been found the symbolic fault can function as a free variable over the alternate set in equations known as *fault equations*. The solution set of a fault equation comprises those alternatives which, if substituted for the designated expression would produce programs that are not differentiated by the input. In this case no alternate that evaluates to a constant other than 3 in the data environment defined by the original execution could be substituted in the original program and not be differentiated by this test. For example, substituting x + 1 for F would go undetected for the input (2,3).

Consider then some potential alternative sets for the designated expression. If F represents the class of constant substitutions, then no constant other than 3 could be substituted and the output be correct for this input. Thus, all constants other than 3 are eliminated by this test. An additional input is therefore necessary to eliminate the alternate program formed by substituting 3 for y.

If F represents the class of variable substitutions, then an inspection of the state at the designated expression reveals that no variable other than y has the value 3. Thus the test eliminates this alternative set. If F represents the class of "variable + constant" faults, then the only possible substitutions are x + 1 and y + 0 since the alternative must evaluate to 3 in the current state, which has x = 2 and y = 3. The second alternative is equivalent to the original expression. An additional input is required to eliminate the first alternative.

Another propagation equation can be generated by equating the two computed values for x:

$$2 + F - (2 + F - 3) = 3$$

This however simplifies to 3 = 3 and thus no alternate programs can be eliminated based on this test.

Executing the program for input (8,5) yields the following set of equations:

$$8 + F - 5 = 8$$
$$8 + F - (8 + F - 5) = 5$$

Again the second equation simplifies to 5 = 5. The first equation simplifies to F = 5. This eliminates the class of constant substitutions because no constant can evaluate to 3 on one execution and 5 on another. It also eliminates the class of variable + constant substitutions because the only possible substitution this time is x + (-3) which is not equivalent to the previously determined substitution. Therefore the test set {(2,3), (8,5)}, is sufficient to eliminate all the aforementioned alternatives.

The following is an example of a symbolic execution using symbolic inputs (B,C,T).

```
proc name
int b, c, t
read b,c,t
t := b + c*b
b := b*t -c
c := c + b*t
print t, b, c
endproc
```

The resulting output is:

```
t: B + (C * B)
b: B * (B + (C * B)) - C
c: C + ((B * (B + (C * B)) - C) * (B + (C * B)))
```

Introducing a symbolic alternative into the first assignment statement

```
t := b + c*F
```

and re-executing yields:

t: B + (C * F)
b: B * (B + (C * F)) - C
c: C + ((B * (B + (C * F)) - C) * (B + (C * F)))

Note that the output is expressed in terms of the symbolic inputs and the symbolic alternative.

Generating the propagation equation for t yields:

B + (C * F) = B + (C * B)

thus F = B (provided c ≠ 0). Symbolic alternative F must therefore have as its value B. Thus, only alternatives equivalent to B in the data environment will not be differentiated.

The propagation equation for b yields the same conclusion:

B * (B + (C * F)) - C = B * (B + (C * B)) - C
thus
F = B.

If c were the only output, then evaluation would be harder. After forming the propagation equation and simplifying there are two solutions:

F = B
AND
2*B*B*C - C*C + B*C*C*F + B*B*C*C = 0

For this propagation condition there are two separate solutions. In the latter case F must evaluate so that

2*B*B*C - C*C + B*C*C*F + B*B*C*C = 0.

The solution to this is the identity

F = (-1*B*B*C +C - 2* B*B)/(B*C).

The division operator in this solution is algebraic and not integer division. If F must be an integer, the solution holds only for those inputs which produce an even division. Examples of this are when b = 1, and c a member of {-3,1,3}. Thus, for these inputs the second solution above would not be differentiated.

5. Use of the model as a meta-theory. The fault-based model developed in the preceding sections can serve not only as method of program testing, but also in the role of a meta-theory for evaluating different code-based testing strategies. In this section an examples is given of the use of the fault-based model of program testing to investigate an area of particular concern to code based strategies, namely that of fault coupling. Other areas of concern such as domain independence and coincidental correctness are dealt with in [Mor84].

Recall that the coupling effect is the assumption that simple faults are linked to complex faults in such a way that a test set which differentiates a program with simple faults, differentiates programs containing complex faults. It was observed that this effect is not provable in practice since the terms "simple" and "complex" are not precisely defined. Such a definition is given below and explored in depth.

Definition For the fault-based arena **E** = <P,S,D,L,A>, two expressions at the distinct locations l_i and l_j, *couple* on a test set T for **E** if and only if ($\exists$ f ε A_i) ($\exists$ g ε A_j) such that

(1) T is successful for P with respect to S.

(2) T differentiates P from $P_f^{l_i}$ and $P_g^{l_j}$

(3) T is successful for $P_{f,g}^{l_i,l_j}$ with respect to S.

Conditions (1) and (2) indicate that the original program is successful, yet its single alternates have been eliminated, since each of the separate substitutions are incorrect by themselves. Condition (3) says that the substitutions in combination compensate for each other in such a way that the double alternate has not been differentiated from the original. This definition can be generalized (at a great loss of clarity) to allow for n coupled expressions.

An example of coupled expressions is given by this program to compute the average of three numbers:

```
proc average
int i, j, k, ave
read i, j, k
ave := 1 + j + k / 3
print ave
endproc
```

The program illustrates two mistakes by a novice programmer. First is the logical error of the misparenthesization. Second is the typographical error of 1 for i. For input i = 6, j = 2, and k = 7 the program correctly computes 5, and differentiates both of the alternate programs with single substitutions:

```
ave := (1 + j + k) / 3
ave := i + j + k / 3
```

which compute, respectively, 3 and 10. From the data collected, it would be possible to erroneously conclude that the test has eliminated all alternate programs. Thus, the two expressions are coupled.

The general results about coupling are not encouraging.

Theorem It is undecidable whether or not coupled expressions occur in an arbitrary fault-based arena.
Proof Take a straight-line program with coupled expressions. Guard the statements by an arbitrary condition. The expressions are still coupled if and only if the path will be executed. This implies the ability to decide whether an arbitrary condition is ever true, which is undecidable.

Corollary It is undecidable whether or not coupled expressions occur in a bounded fault-based arena.

However, there is some hope.

Theorem There are unbounded fault-based arenas for which the presence of coupled expressions is decidable.
Proof The example just given illustrates this point.

Coupling is a pernicious problem that can compromise all fault-based testing strategies. In revealing subdomains [Wey80], the problem is cited and then ignored. In domain testing[Whi80] and perturbation testing [Zei83] the problem is presumed not to occur as a "simplifying assumption." As mentioned above, mutation testing identifies the problem, and tries to argue on the basis of a few experiments that coupled expressions rarely occur. In section the next section formal arguments are given which indicate that expressions indeed couple infrequently for certain classes of programs.

When they do, however, program faults may go undetected.

5.1. The likelihood of coupling. The previous section introduced coupled expressions as the primary pitfall of fault-based testing. This section discusses why this is so and describes approaches to analyzing how likely coupling is.

In the simplest case coupling relates four programs determined by fault-based testing: P, $P_A^{l_1}$, $P_B^{l_2}$, $P_{A,B}^{l_1,l_2}$. For notational convenience these alternates are denoted by P_A, P_B, and P_{AB}. When locations l_1 and l_2 differ, the possibility of coupling arises.

Definition For alternate program P_e^l and alternative e at l in A the *associated failure set* is $\{x \mid [P](x) \neq [P_e^l](x)\}$.

Let D_A, D_B, and D_{AB} be the associated failure sets of P_A, P_B, and P_{AB}, respectively. The potential intersections of these sets are shown in figure 2. Conditions for coupling can be expressed by assertions about regions of the diagram. For example, it is necessary that

(1) Region 2 be non-empty or regions 1 and 3 must be non-empty.

(2) The test set T be a subset of regions 1-3.

If the existence (or non-existence) of points in one group of regions implies the existence (or non-existence) of points in another region, then perhaps something positive can be said about the absence

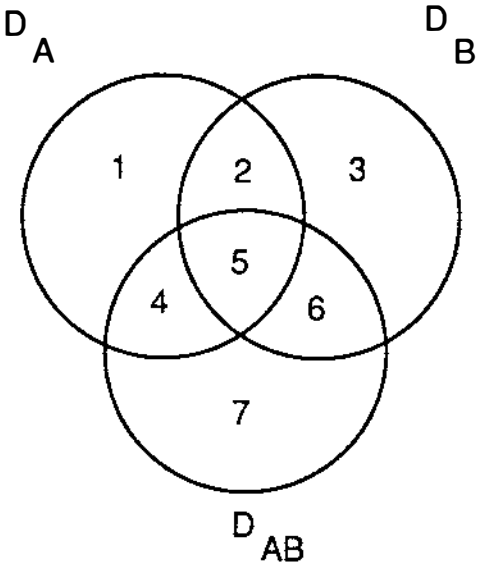


Figure 2

of coupling. Unfortunately,

Theorem Any combination of regions can be empty.

Proof Let P be a table-lookup program that chooses which table to search depending on whether the variables x and y are assigned 0 or 1. (Thus, table 1 is chosen if x = 0 and y = 0, table 2 is chosen if x = 0 and y = 1, etc.) Let x and y be initialized to 0 in P, and let the alternate programs have the initializations

P_A
x = 0, y = 1
P_B
x = 1, y = 0
P_{AB}
x = 1, y = 1

The effect stated by the theorem can be confirmed by appropriately filling in the tables.

If nothing can be said about the emptiness or non-emptiness of any regions, perhaps something can be said of relative sizes of the regions. In particular for coupling to occur on a test set T it must be that T is a subset of regions 1-3. If therefore it can be shown that these regions are "small" relative to D_{AB}, then the expressions couple on "few" test sets. Hence, if the test sets are chosen randomly, there is little chance of selecting a bogus test set.

To investigate the relative sizes of regions, consider an example of coupling in linear equations. Suppose the original program computes

$$f(x)=3x+4$$

and the correct program computes

$$g(x)=4x+3$$

Let the two alternate programs compute

$$f_1(x)=4x+4$$

and

$$f_2(x)=3x+3.$$

For what values of x will coupling occur? To satisfy the first condition, inputs must solve the equation

$$f(x)=g(x)$$

i.e., the original must compute the correct answer on the test set. The only solution is x = 1. Thus, coupling can only occur on the point x = 1. Coupling does occur for this point because the third condition is satisfied, namely,

$$f(1)\neq f_1(1)$$

and

$$f(1)\neq f_2(1)$$

Thus the expressions couple, but only on input 1. Such a point is called a *coupling point*.

For this example then the set of coupling points is small compared to the points for which coupling does not occur. There is therefore little chance of randomly choosing 1 out of all possible integers as the point to differentiate the single alternate programs.

In the argument above the exact values of the constants are immaterial. What is relevant is the size of the solution set, not the particular solutions. In fact, the entire argument above can be maintained for the general class of programs which compute multinomials.

Definition A *multinomial* in x_1 , ... , x_n is any expression in x_1 , ... , x_n and integer constants using only the operators +, -, or *.

Let P be a incorrect program which computes a multinomial in n variables. Suppose the correct program is P′, an alternate program for P produced by substituting several alternatives for expressions in P. Suppose further that P′ computes a multinomial in n variables. Then the solution set to the equation

$$[P](x) = [P'](x)$$

is at most an n-1 dimensional subset of the input space.

Since the class of multinomials includes the classes of linear forms and polynomials, the following theorem is established.

Theorem For any program that computes a linear form, a polynomial, or a multinomial its coupling points are a zero-volume subset of the input space.

The proof of the above theorem is invalid if functions more general than multinomials are allowed. This is because the solution set may not be an n-1 dimensional subset of the input space. For example, for the program that computes (by integer division)

$$f(x)=\frac{2}{x}$$

$$g(x)=\frac{4}{3x},$$

every x > 2 is in the solution set.

Another possible extension to the above theorem is to allow the introduction of conditional statements to compute selected multinomials. This fails, however, because a condition can arbitrarily restrict a multinomial to only be computed on its coupling points. Of course, the opposite can occur and the path conditions make it impossible for any coupling to occur.

6. Summary. A new theory of program testing has been presented which is based on the idea of demonstrating that certain faults are not present in a program. Significant features of this fault-based theory include

(1) A program is presumed incorrect in a limited fashion.

(2) Incorrectness is modeled by associating program expressions with potentially infinite alternative sets.

(3) The information content of a test set can be characterized by the set of alternates it eliminates from a program.

Symbolic testing was introduced as a means of implementing and exploring the ramifications of the theory. Symbolic faults introduced into a program can act as stand-ins for alternatives. Output from a program containing a symbolic fault expresses the effect any of the alternatives would have had on the program, had they been substituted for the symbolic fault. Equating this symbolic output with the actual output yields a propagation equation whose solution is the set of all values which could have been substituted for symbolic fault and gone unnoticed. This solution set in turn induces fault

equations whose solution is the set of non-differentiated alternatives.

The problem of fault coupling was defined and shown to pose few problems for certain path functions. This illustrates the use of the theory as a meta-theory for code-based testing, since coupling is a problem faced in most code-based testing schemes.

References

[DeM78] R. A. DeMillo, R. J. Lipton, and F. G. Sawyer, Hints on Test Data Selection: Help for the Practicing Programmer, *Computer 11*, p. 34 41 (April 1978).

[Wey80] Elaine J. Weyuker and Thomas J. Ostrand, Theories of Program Testing and the Application of Revealing Subdomains, *IEEE TSE SE-6*, 3, pp. 236-246 (May 1980).

[Mey79] Glenford J. Meyers, *The Art of Software Testing*, John Wiley & Sons(1979).

[Goo75] John B. Goodenough and Susan L. Gerhart, Toward a Theory of Test Data Selection , *IEEE Trans. Soft. Eng. TSE-SE1*, 2, pp. 156-173 (June, 1975).

[How76] William E. Howden, Reliability of the Path Analysis Testing Strategy, *IEEE TSE SE-2*, 3, (Sept. 1976).

[IEE83] IEEE, *IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 729-1983*, IEEE(1983).

[Lin79] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming Theory and Practice*, Addison-Wesley(1979).

[Mor84] Larry J. Morell, A Theory of Error-Based Testing, University of Maryland TR-1395, Department of Computer Science(August, 1984). PhD Thesis

[Mil80] Harlan D. Mills, Function Semantics for Sequential Programs, *Information Processing 80*, (1980).

[How78] William E. Howden, Algebraic Program Testing, *Acta Informatica 10* , (1978).

[Cla77] L. A. Clarke, A System to Generate Test Data and Symbolically Execute Programs, *IEEE TSE SE-2*, p. 215 222 (Sept. 1977).

[Whi80] Lee J. White and Edward I Cohen, A Domain Strategy for Computer Program Testing, *IEEE TSE SE-6*, 3, pp. 247-257 (May 1980).

[Zei83] Steven J Zeil, Testing for Perturbation of Program Statements, *IEEE TSE SE-9*, p. 335 346 (May 1983).

Test Data Selection Using the RELAY Model of Error Detection

Debra J. Richardson†
Margaret C. Thompson‡

†Information and Computer Science Department
University of California
Irvine, California 92717

‡Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

RELAY, a model for error detection, defines *revealing conditions* that guarantee that a fault *originates* an error during execution and that the error *transfers* through computations and data flow until it is *revealed*. This model of error detection provides a criterion for test data selection. The model is applied by choosing a fault classification, instantiating the revealing conditions for the classes of faults, and applying them to the program being tested. Such an application guarantees the detection of errors caused by any fault of the chosen classes. In this paper, we summarize the RELAY model of error detection and its instantiation. RELAY is then applied to a sample program to illustrate its use as a test data selection criterion.

Biographical Sketches

Debra J. Richardson received the B.A. degree in Mathematics from Revelle College of the University of California at San Diego in 1976 and the M.S. and Ph.D. degrees in Computer and Information Science from the University of Massachusetts at Amherst in 1978 and 1981, respectively.

Dr. Richardson is currently an Assistant Professor in the Department of Information and Computer Science at the University of California at Irvine. Her research interests include program testing and verification, specification languages, and software development environments. Her previous work includes the conception of the Partition Analysis Method, a testing method based on both the specification and the implementation of a program. The major thrust of her current work is the development of framework within which various testing methods can be described and the formulation of the RELAY model for testing, so named because of its view of error detection as analogous to a relay race.

Margaret C. Thompson received the B.A. degree in Biological Sciences from Smith College in 1982 and the M.S. degree in Computer and Information Sciences from the University of Massachusetts at Amherst in 1986. She is currently a doctoral candidate in Computer Science at the University of Massachusetts at Amherst. In her dissertation work, she will continue investigating and developing the RELAY model of error detection.

This research was supported in part by the National Science Foundation under grants DCR-83-18776 and DCR-84-04217 and by the Rome Air Development Center grant F30602-86-C-0006.

1 Introduction

The goal of testing a program is the detection of errors. This is typically done by attempting to select test data for which execution of the program produces erroneous results. Many testing techniques [Bud81,Bud83,Ham77,How82,How85,Mor84,Zei83,Wey81] are directed toward the detection of errors that might result from commonly occurring faults in software. These "fault-based" testing techniques assume that the program being tested is "almost correct", which might be determined by successfully passing some high-level functional testing phase. If a "comprehensive" set of test data is selected by a fault-based technique and the program executes correctly, then the tester gains confidence that the program does not contain specific types of faults.

This paper reports on a new model of error detection called RELAY, which provides a fault-based criterion for test data selection. The RELAY model builds upon the testing theory introduced by Morrell [MH81,Mor84], where an error is "created" when an incorrect state is introduced at some fault location, and it is "propagated" if it persists to the output. We refine this theory by more precisely defining the notion of when an error is introduced and by differentiating between the persistence of an error through computations and its persistence through data flow operations. We introduce similar concepts, *origination* and *transfer*[1], as the first erroneous evaluation and the persistence of that erroneous evaluation, respectively. The RELAY model defines *revealing conditions* that guarantee that a fault *originates* an error during execution and that the error *transfers* through all affected computations until it is *revealed*.

The next section summarizes the RELAY model and briefly describes its instantiation to develop revealing conditions for a class of faults; more detail is provided in a related paper[RT86b]. In the third section, we present an example of the application of RELAY as a test data selection criteria.

2 RELAY: A Model of Error Detection

The primary use for the RELAY model of error detection is as a test data selection criterion. RELAY is capable of guaranteeing the detection of errors that result from some chosen class or classes of faults. In addition, given test data that has been selected by another criterion, RELAY can be used as a measurement technique for determining whether that test data detects such errors. Before describing the RELAY model of error detection, we first introduce a terminology within which we define the RELAY model.

2.1 Terminology

We consider the testing of a *module* M that implements some function $F_M : X_M \rightarrow Z_M$. A module M can be represented by a *control flow graph* $G_M = (N, E)$, where N is a (finite) set of

[1] We have chosen the term "originate" rather than "create" or "introduce", because we feel it better connotes the first location at which an erroneous evaluation occurs and does not imply the mistake a programmer makes while coding. We have chosen the term "transfer" over "propagate" so as to avoid the connotation of an "increase in numbers" and instead of "persist" so as not to conflict with Glass's notion [Gla81], where an error is persistent if it escapes detection until late in development.

nodes and $E \subseteq N \times N$ is the set of edges. N includes a start node n_{start} and a final node n_{final}; each other node in N represents a simple statement[2] or the predicate of a conditional statement in M. A *subpath* in a control flow graph $G_M = (N, E)$ is a finite, possibly empty, sequence of edges $p = [(n_1, n_2), ..., (n_{|p|}, n_{|p|+1})]$ in E; note that the last node $n_{|p|+1}$ has been selected by virtue of its inclusion in the last edge but is not visited in the subpath traversal. An *initial subpath* p is a subpath whose first node is n_{start}. A *path* P[3] is an initial subpath whose last node is n_{final}.

A *test datum* t for a module M is a sequence of values input along some initial subpath. For any node n in G_M, $DOMAIN(n)$ is the set of all test data t for which n can be executed. A test datum t may be an incomplete sequence of input values in the sense that it cannot execute a complete path. This may be because: 1) additional input is needed to complete execution of some path; or 2) the initial input t may cause the module to terminate abnormally (before n_{final}) or possibly never to terminate. The possibility of incomplete input sequences allows for testing criteria that consider invalid test data, which are not in the domain of M but for which M may initiate execution. The *test data domain* D_M for $G_M = (N, E)$ is the domain of inputs from which test data can be selected. [4] $D_M = \{t \mid \exists\, n \in N, t \in DOMAIN(n)\}$.

A testing method typically specifies some subset of the test data domain for execution. A *test data set* T_M for a module M is a finite subset of the test data domain, $T_M \subseteq D_M$. A *test data selection criterion* S, is a set of rules for determining whether a test data set T_M satisfies selection requirements for module M.

To reveal errors by testing, there is usually some test oracle that specifies correct execution of the module [How78,Wey82]. A test oracle might be a functional representation, formal specification, or correct version of the module or simply a tester who knows the module's correct output. In any case, an *oracle* $O(X_O, Z_O)$ is a relation, $O = \{(x, z)\} \subset X_O \times Z_O$, where X_O and Z_O are the domain and range, respectively, of the oracle. When $(x, z) \in O$, z is an acceptable output for x. Execution of a module M on input x reveals an *output error* when $(x, M(x)) \notin O$.

A "standard" oracle judges the correctness of the module's output for valid input data. Testers often have a concept of the "correct" behavior of a module, however, and not just its correct output. Rather than waiting until output is produced to find errors, the tester might check the computation of the module at some intermediate point, as one does when using a run-time debugger. This approach to testing can be performed with an oracle that includes information about intermediate values that should be computed by the module; this information might be derived from some correct module, an axiomatic specification, run-time traces [How78], or monitoring of assertions. Let us associate with the execution of an initial subpath p on some test datum t a *context* $C_{p(t)}$, which contains the values of all variables after execution of $p(t)$. A *context oracle* O_C is a relation $O_C = \{((t, p), C_{p(t)})\}$, that relates a test datum and an initial subpath (t, p) to one or more contexts

[2] Single statements are considered here for the clarity of the presentation; a simple extension allows nodes to represent a group of simple statements.

[3] Where the distinction between a subpath and a path is important, we will use an upper case letter (P) to signify a path and a lower case letter (p) for a subpath (or initial subpath).

[4] D_M does not include data that would result in an invalid call of the module.

$C_{p(t)}$ that are acceptable after execution of p on t[5]. Execution of a path p on test datum t reveals a *context error* when $((t,p), C_{p(t)}) \notin O_C$.

2.2 RELAY

The errors considered within the RELAY model are those caused by some chosen class or classes of *faults* in the module's source code. The fault-based approach to testing relies on an assumption that the module being tested bears a strong resemblance to some hypothetically-correct module. Such a module need not actually exist, but we assume that the tester is capable of producing a correct module from the given module and knowledge of the errors detected. As currently formulated, RELAY is limited to the detection of errors resulting from a single fault.

A node containing a fault may be executed yet not reveal an error; the module appears correct, but just by coincidence of the test data selected. It is also possible that the tested module produces correct output for all input despite a discrepancy between it and the hypothetically-correct module. In this case, the module is not merely coincidentally correct, it is correct. Thus, a discrepancy is only "potentially" a fault. Likewise, incorrect evaluation of an expression is only "potentially" an error since the erroneous value may be masked out by later computations before an erroneous value is output. A **potential fault**, denoted f_n, is a discrepancy between a node n in the tested module M and the corresponding node $n^\star$ in the hypothetically-correct module $M^\star$ — that is, $n \neq n^\star$. The evaluation of some expression EXP[6] in M, which contains a potential fault, and the corresponding expression $EXP^\star$ in $M^\star$ results in a **potential error** when $exp \neq exp^\star$. To discover a potential fault, erroneous results must appear for some test datum as a context error or as an output error, depending on the type of oracle used.

RELAY is a model that describes the ways in which a potential fault manifests itself as an error. Given some potential fault, a potential error **originates** if the smallest subexpression of the node containing the potential fault evaluates incorrectly. Consider the module in Figure 1 for example. Suppose that the statement $X := U*V$ at node 1 contains a variable reference fault and should be $X := B*V$. A potential error originates in the smallest expression containing the potential fault, which is the reference to U, whenever the value of U differs from the value of B.

It is not only important to originate an error but also to ensure that it is not masked out by later computations. A potential error in some expression **transfers** to a "super"–expression that references the erroneous expression if the evaluation of the "super"–expression is also incorrect. Take another look at Figure 1; if V holds the value zero, the potential error in U that originates in node 1 does not transfer to affect the assignment to X; the potential error transfers, on the other hand, whenever V is nonzero. To reveal a context error, a potential fault must originate a potential error that transfers through all computations in the node thereby causing an incorrect context. This is termed **computational transfer**. To reveal an output error, a potential fault

[5]For simplicity, the granularity of the context oracle is assumed to be the same as that of the control flow graph, although this is not necessary.

[6]Upper case $[EXP]$ is used here to denote the source-code expression, while lower case $[exp]$ denotes the evaluated expression.

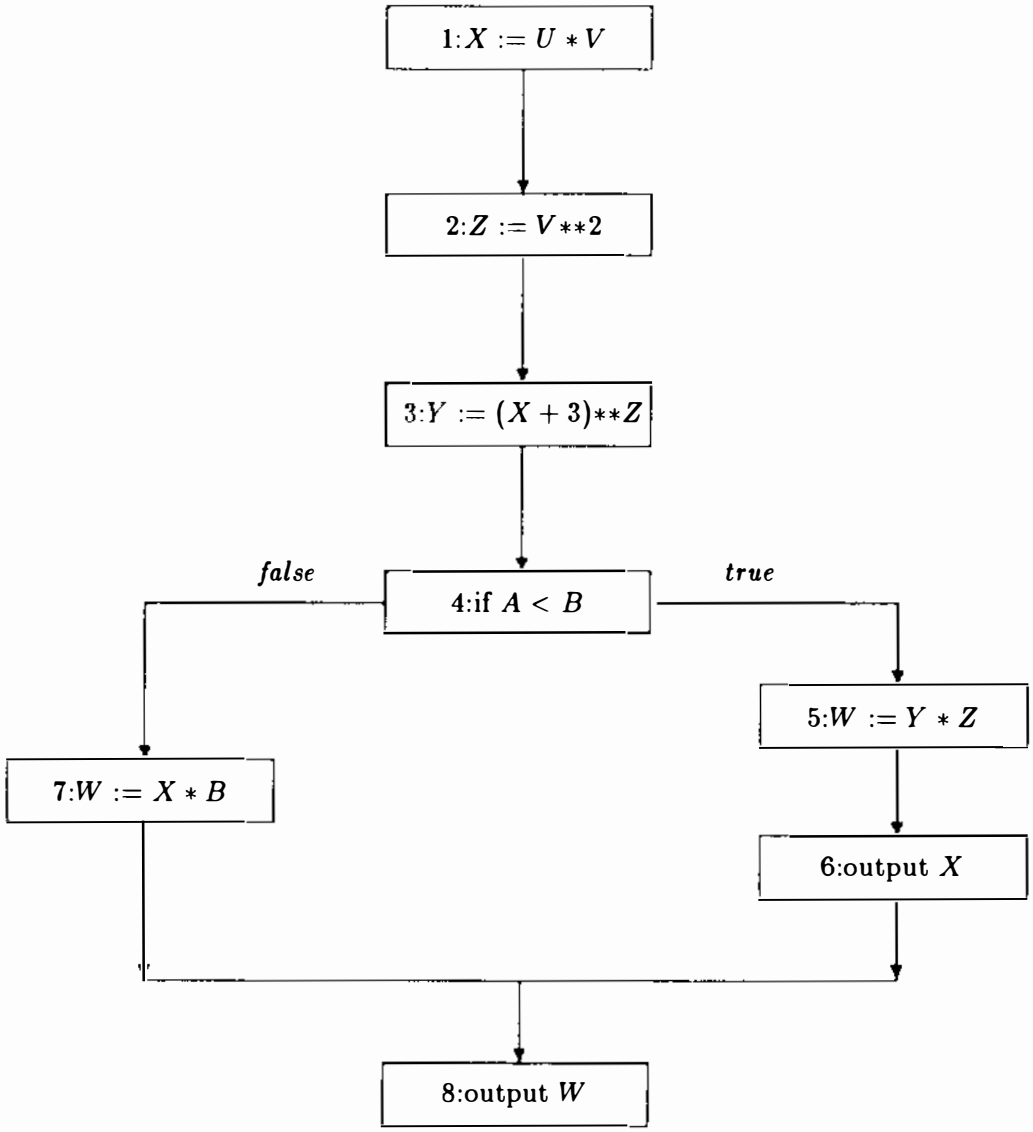


Figure 1: Module for Test Data Selection

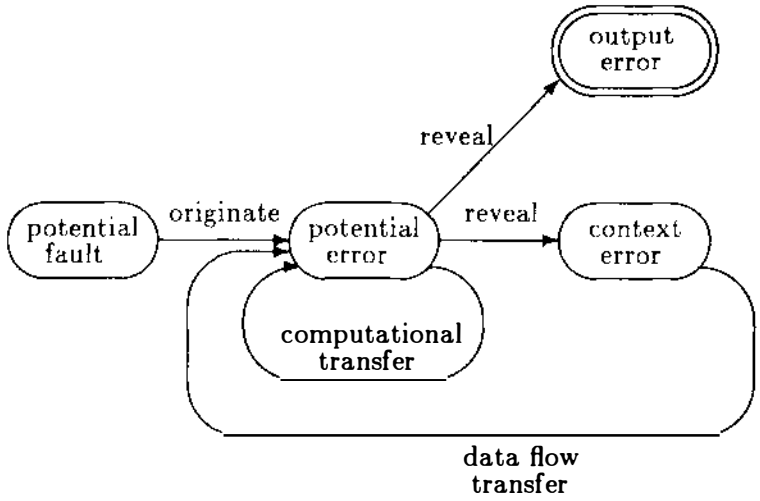


Figure 2: RELAY Model of Error Detection

must cause a context error that transfers from node to node until an incorrect output results. This transfer includes **data flow transfer**, whereby a potential error reaches another node — that is, the potential error is reflected in the value of some variable that is referenced at another node — as well as computational transfer within the nodes that an erroneous value reaches. Using the example of Figure 1 again, the potential error in X must transfer through data flow to a use, say at node 7, transfer through the computations at node 7 to produce an error in W, and then transfer to the output of W at node 8. We know unequivocally that the module is incorrect only if an output error is revealed. Thus, a potential fault is a **fault** only if it produces incorrect output for some test datum.

Figure 2 illustrates the RELAY model of error detection and how this model provides for the discovery of a fault. The conditions under which a fault is detected are 1) origination of a potential error in the smallest containing subexpression; 2) computational transfer of that potential error through each operator in the node, thereby revealing a context error; 3) data flow transfer of that context error to another node on the path that references the incorrect context; 4) cycle through (2) and (3) until a potential error is output. If there is no single test datum for which a potential error originates and this "total" transfer occurs, then the potential fault is not a fault, and the module containing the potential fault is equivalent to some hypothetically-correct module.

As shown in Figure 3, the RELAY view of error detection has an illustrative analogy in a relay race, hence the name of our model. The starting blocks correspond to the fault location. The take off of the first runner, as the gun sounds the beginning of the race, is analogous to the origination of a potential error. The runner carrying the baton through the first leg of the race is the computational transfer of the error through that first statement. The successful completion of a leg of the race has a parallel in revealing a context error, and the passing of the baton from one runner to the next is analogous to the data flow transfer of the error from one statement to another. Each succeeding leg of the race corresponds to the computational transfer through another statement. The race goes on until the finish line is crossed, which is analogous to the test oracle revealing an output error.

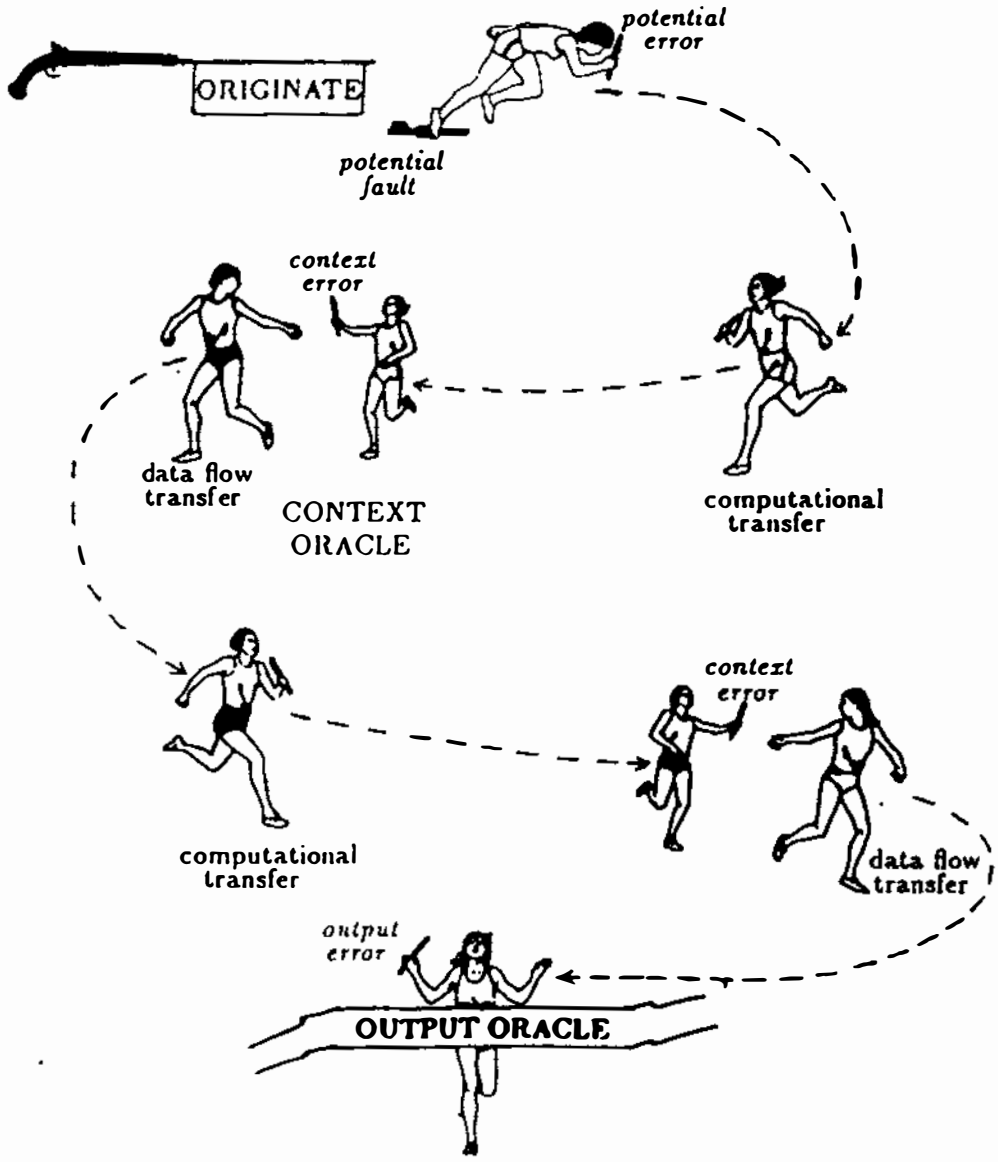


Figure 3: The Testing Relay

Our goal, of course, is to complete the relay race -- that is, to detect errors. To this end, the RELAY model proposes the selection of test data that originates an error for any potential fault of some type and transfers that error until it is revealed. RELAY uses the concepts of origination and transfer to define conditions that are necessary and sufficient to guarantee error detection. When these conditions are instantiated for a particular type of fault, they provide a criterion by which test data can be selected for a program so as to guarantee the detection of an error caused by any fault of that type.

Given an oracle and a module M with $G_M = (N, E)$ that contains a potential fault f_n at node $n \in N$, a test data selection criterion S is said to **guarantee detection** of a fault f_n if for all test data sets T_M that satisfy S, there exists $t \in T_M$ such that f_n originates an error for $M(t)$ that transfers until it is detected by the oracle. If a context oracle exists, the potential fault must reveal a context error for some test datum in every test data set. Note that guaranteeing detection of a context error does not necessarily mean that an output error will result for this execution, since it is possible that the context error is masked out by later statements and not transferred to the output. If error detection is done by a standard output oracle, then a context error revealed by the fault must also transfer to the output for some test datum in every test data set.

Here, we define *origination*, *transfer*, and *revealing conditions* that are necessary and sufficient to guarantee that an error is revealed. Sufficient means that if the module is executed on data that satisfies the conditions and the node is faulty, then an error is revealed. Necessary, on the other hand, means that if an error is revealed then the module must have been executed on data that satisfies the condition and the node is faulty.

These conditions are defined for a potential fault independent of where the node occurs in the module. The test data selected, however, must execute the node within the context of the entire module. Thus, for a potential fault at node n, such test data are restricted to $DOMAIN(n)$. Because these conditions are both necessary and sufficient, if the conditions are *infeasible* within $DOMAIN(n)$, then no error can be revealed and the potential fault is not a fault. Although, in general, the feasibility problem is undecidable, in practice, it can usually be solved.

First, suppose that we are attempting to detect a particular fault f_n in a node n. This is somewhat unrealistic, since if one explicitly knew the location of a fault, one would fix it. We will address this issue in a moment, after some groundwork is laid.

To reveal an output error, we must first generate a context error at the node containing the fault; thus, let us first consider the conditions required to guarantee the detection of a context error. By requiring test data to distinguish the faulty subexpression from the correct one, the **origination condition** for a potential fault f_n guarantees that the smallest subexpression containing f_n originates a potential error. A potential error originating at the smallest subexpression containing a potential fault must transfer to affect evaluation of the entire node. By requiring test data that distinguishes the parent expression referencing a potential error from the parent expression referencing the correct subexpression, the **computational transfer condition** guarantees that a potential error transfers through a parent operator. To affect the evaluation of a node, test data must satisfy the computational transfer condition for each operator that is an ancestor of the subexpression in which the potential error originates thereby producing a context error. The **node transfer condition** is the conjunction of all such computational transfer conditions. To guarantee

a fault's detection through revealing a context error, a single test datum must satisfy both the origination and node transfer conditions. The **revealing condition** for a context error resulting from a potential fault f_n occurring in node n is the conjunction of the origination condition and the node transfer condition for f_n and n.

As an example of these conditions for error detection, consider again the module in Figure 1. If the statement $X := U * V$ at node 1 should be $X := B * V$, then the origination condition is $[u \neq b]$. This originated potential error must transfer through the multiplication by V; the corresponding computational transfer condition is $(u * v \neq b * v)$, which simplifies to $(v \neq 0)$. This value must then transfer through the assignment to X, which is trivial. Thus, the revealing condition for a context error resulting from this potential fault is $[(u \neq b)$ and $(v \neq 0)]$.

Testing is primarily concerned with the generation of an output error as the manifestation of a fault and not only with incorrect values at intermediate points in the module. Thus, we must guarantee that a context error transfers to affect execution of the module as a whole. A context error is evidenced through a potential error in at least one variable. By requiring test data that causes the execution of a statement referencing a variable that contains a potential error and that causes the smallest subexpression containing that reference to result in a potential error, a **data flow transfer condition** describes the requirements for transfer of a context error from one statement to another. To reveal an output error, we must execute a def-use chain that begins with the node containing the potential fault and ends with the output of a variable. A *def-use chain* is a chain of alternating definitions and uses of variables, where each definition reaches the next use in the chain and that use defines the next variable in the chain. Satisfaction of the data flow transfer conditions will force execution of such a chain. In addition, the subsequent node transfer conditions for the references forced by data flow as well as the context error revealing condition at the location of the fault must be satisfied. A **chain transfer condition** for a def-use chain is the conjunction of the data flow transfer conditions for all pairs in the def-use chain and the node transfer conditions for all uses in the def-use chain. The **revealing condition** for an output error is the conjunction of the context error revealing condition and the chain transfer condition for the def-use chain from the fault location to the output.

Consider again the potential variable reference fault at node 1 in Figure 1. One def-use chain from the fault location to an output consists of the definition of X at node 1, followed by a use of X at node 7, where W is defined, followed by a use of W in the output statement at node 8. The potential error in X transfers through data flow to node 7 whenever the false branch of the conditional at node 4 is taken; thus, the data flow transfer condition is $(a \geq b)$. Reference to the potential error in X must transfer through the multiplication by B to the assignment of W at node 7, which entails a node transfer condition of $(b \neq 0)$. Thus, for this def-use chain, the chain transfer condition is $[(a \geq b)$ and $(b \neq 0)]$. Recall that the context error revealing condition is $[(u \neq b)$ and $(v \neq 0)]$, creating an output error revealing condition for this chosen def-use chain of $[(u \neq b)$ and $(v \neq 0)$ and $(a \geq b)$ and $(b \neq 0)]$.

As currently defined, derivation of revealing conditions is dependent on knowledge of the correct node. Since this is unlikely, an alternative approach is to assume that any node, in fact any subexpression of any node, might be incorrect and consider the potential ways in which that expression might be faulty. By grouping these potential faults into classes based on some common

characteristic of the transformation, we define conditions that guarantee origination of a potential error for any potential fault of that class. A class of potential faults determines a set of alternative expressions, which must contain the correct expression if the original expression indeed contains a fault of that class. To guarantee origination of a potential error for a class, the potentially faulty expression must be distinguished from each expression in this alternate set. For each alternative expression, then, our model defines an origination condition, which guarantees origination of a potential error if the corresponding alternate were indeed the correct expression. For an expression and fault class, we define the **origination condition set**, which guarantees that a potential error originates in that expression if the expression contains a fault of this class. The origination condition set contains the origination condition for each alternative expression.

For each alternative expression, a potential error that originates must also transfer through each operator in the node to reveal a context error and through data flow and subsequent computations to reveal an output error. The transfer conditions, which are determined by these subsequent manipulations of the data, are independent of the particular alternate. Thus, for a fault class, each alternate defines a revealing condition, which is the conjunction of the origination condition and the transfer conditions. The **revealing condition set** contains a revealing condition for each alternate in the alternate set and is necessary and sufficient to guarantee that a potential fault of a particular class reveals an output error.

Once again, consider the module in Figure 1 and the statement $X := U * V$, but now suppose that the reference to U might be faulty but we do not know what variable should be referenced. To guarantee origination of a potential error for an incorrect reference to U, we must select test data such that for each alternative variable, $\overline{U}$ [7], T contains a test datum where the value of U is different from the value of $\overline{U}$ at node 1. The possible alternates depend on what other variables may be substituted for U without violating the language syntax. If we assume that all variables referenced in this module are of the same type, then there are seven alternates and hence seven origination conditions. The origination condition set is $\{[u \neq \overline{u}] \mid \overline{U} \in \{A, B, V, W, X, Y, Z\}\}$. The node transfer condition for node 1 is $[v \neq 0]$. The chain transfer condition for the use of X to define W at node 7 and the output of W at node 8 is $[(a \geq b)$ and $(b \neq 0)]$. Thus, the set $\{[(u \neq \overline{u}$ and $(v \neq 0)$ and $(a \geq b)$ and $(b \neq 0)] \mid \overline{U} \in \{A, B, V, W, X, Y, Z\}\}$ is a sufficient revealing condition set for this potential fault. This set is sufficient but not necessary because all def-use chains are not considered.

The RELAY model of error detection is based on the generic revealing condition sets just defined. The model is applied by first selecting a fault classification. Given a particular class of faults, the generic origination and transfer conditions are instantiated to provide conditions specific to that class. We have instantiated the model for six classes of faults: constant reference fault, variable reference fault, variable definition fault, boolean operator fault, relational operator fault, and arithmetic operator fault [RT86b]. For a module being tested, the instantiated origination and transfer conditions are evaluated for the nodes in a module's control flow graph to determine the actual revealing condition sets. Satisfaction of these sets guarantees the detection of an error for any fault in the chosen classification. The actual revealing conditions for a module can be used to

[7] We use the bar notation to denote an alternate.

variable referenced	origination condition set
V	$\{[\overline{v} \neq v \mid \overline{V}$ is a variable other than V that is type-compatible with $V]\}$

Table 1: Origination Condition Set for Variable Reference Fault

measure the effectiveness of a pre-selected set of test data and/or to select a set of test data. A simple example of RELAY as a test data selection criterion is presented in the next section.

3 Use of RELAY for Test Data Selection

As an example of how RELAY can be used to select test data, consider the potential faults at node 1 in the example shown in Figure 1. If we assume that the module is "almost correct", then the statement at node 1 might have a variable reference fault, a variable definition fault, or an arithmetic operator fault. These classes are three of six for which RELAY has been instantiated thus far. Both the origination conditions for these three classes as well as the applicable computational transfer conditions are reported in this paper. Let us first consider the conditions that must be satisfied to guarantee that a context error is revealed at node 1 for these three classes of faults. Then, we construct the chain transfer conditions necessary to reveal an output error. Finally, we append each transfer condition to each origination condition to provide the output error revealing condition.

3.1 Context Error Revealing Conditions

Consider first a potential variable reference fault at node 1; either the reference to U or the reference to V could be incorrect. The origination condition set for this class of faults is shown in Table 1. When evaluated for the reference to U at node 1 [8] , the origination condition set is $\{[u \neq u] \mid u \in \{A, B, V, W, X, Y, Z\}\}$. The transfer condition for multiplication is shown in Table 2; when evaluated for node 1, this requires $(v \neq 0)$.

Thus, the context error revealing condition set is $\{[(u \neq u \text{ and } (v \neq 0)] \mid \overline{U} \in \{A, B, V, W, X, Y, Z\}\}$ (as constructed in the previous section). A similar condition set must be satisfied to reveal a context error for a potentially incorrect reference to V; it is $\{[(v \neq \overline{v} \text{ and } (u \neq 0)] \mid \overline{V} \in \{A, B, U, W, X, Y, Z\}\}$.

Next, consider a potential arithmetic operator fault. The multiplication operator at node 1 could potentially be any other arithmetic operator. The origination condition set for an incorrect multiplication operator, where the alternative operators are $+, -, /, **$, is shown in Table 3. Thus, to guarantee origination of a potential error for a potential arithmetic operator fault in this node, a

[8] These conditions refer to values before execution of the node since they are requirements on test data selected for the node.

operator	expression	transfer conditions
$+$	$exp_1 + exp_2 \neq \overline{exp_1} + exp_2$	*true*
$*$	$exp_1 * exp_2 \neq \overline{exp_1} * exp_2$	$exp_2 \neq 0$
$**$	$exp_1 ** exp_2 \neq exp_1 ** exp_2$	$(exp_2 \neq 0)$ and $(exp_1 \neq -\overline{exp_1}$ or $exp_2 \bmod 2 \neq 0)$

Table 2: Transfer Conditions through Arithmetic Operators

expression	origination condition set
$(exp_1 * exp_2)$	$\{[(exp_1 * exp_2) \neq (exp_1 + exp_2)]$, $[(exp_1 * exp_2) \neq (exp_1 - exp_2)]$, $[(exp_1 * exp_2) \neq (exp_1 / exp_2)$ or $[(exp_1 * exp_2) \neq (exp_1 ** exp_2)]\}$.

Table 3: Origination Condition Set for $*$ Operator Fault

test data set must satisfy the origination condition set $\{[(u*v) \neq (u+v)], [(u*v) \neq (u-v)], [(u*v) \neq (u/v)], [(u*v) \neq (u**v)]\}$. The potential error originated by a potential arithmetic operator fault in this statement must transfer through the assignment operator in order to reveal a context error; this requires no additional conditions. The context error revealing condition set is, therefore, the same as the origination condition set.

Now, consider a potential variable definition fault. The origination condition set for this class of faults is shown in Table 4. Again assuming all eight variables are of the same type, the origination condition set is $\{[(x \neq \bar{x}$ **or** $(u*v \neq x)] \mid X \in \{A, B, U, V, W, Y, Z\}\}$. There are no transfer conditions required to reveal a context error for this potential fault once a potential error is originated. Thus, the context error revealing condition set for this potential fault is the same as the origination condition set.

assignment	origination condition set
$V := EXP$	$\{[(\bar{v} \neq v)$ or $(exp \neq v) \mid V$ is a variable other than V that is type-compatible with $V]\}$.

Table 4: Origination Condition Set for Variable Definition Fault

3.2 Chain Transfer Conditions

We are now in a position to consider the additional requirements necessary to guarantee revelation of an output error for the potential faults discussed above. There are two paths in this module, and for each path, we must construct the chain transfer condition for each def-use chain, where the definition at node 1 reaches an output.

Let us first consider the path $[(1,2),(2,3),(3,4),(4,7),(7,8)]$, where the false branch of the condition at node 4 is taken. Along this path, the only def-use chain consists of the definition of X at node 1, the use of X at node 7 to define W, and the output of W at node 8. The corresponding chain transfer condition (as described in the previous section) is $[(a \geq b)$ and $(b \neq 0)]$.

Now, consider the second path $[(1,2)(2,3)(3,4),(4,5),(5,6),(6,8)]$, along which there are two def-use chains beginning with the fault location and ending with an output statement. One such def-use pair consists of the definition of X at node 1 followed by the use of X in the output statement at node 6. The data flow transfer condition to force this def-use pair is $(a < b)$ and no other node transfer conditions are required. Thus, the chain transfer condition for this def-use chain is $[(a < b)]$.

The other def-use chain along this path consists of the definition of X at node 1, followed by the use of X at node 3 where Y is defined, followed by the use of Y at node 5 to define W, followed by the use of W in the output statement of node 8. The transfer from node 1 to node 3 is sequential, so there is no data flow transfer condition required. At node 3, the potential error in X must transfer through the addition and the exponentiation to the assignment of Y; this is defined by the node transfer condition for node 3. The transfer conditions for addition and exponentiation are shown in Table 2. The transfer condition for + is trivial — that is, (true). There are several possible ways, however, in which exponentiation could mask out a potential error in the expression $(X + 3)$. The most obvious way is if the value of Z is zero; thus one transfer condition is $(z \neq 0)$. Another possibility is if Z is even and the potential error expression containing X is the negation of the potentially correct expression. It is sufficient, but not necessary, to simply constrain Z to be odd. Thus, a sufficient node transfer condition for node 3 is $((z \neq 0)$ and $(\mathbf{odd}(z)))$ [9] . The potential error must also transfer from this definition of Y at node 3 to the use of Y at node 6, which is defined by the data flow transfer condition $(a < b)$. Within node 6, the potential error in Y must transfer through the multiplication by Z to the definition of W, which requires satisfaction of the node transfer condition $(z \neq 0)$. Thus, revelation of an output error for this def-use chain is achieved by satisfying the following chain transfer condition — $[(z \neq 0)$ and $(\mathbf{odd}(z))$ and $(a < b)]$.

3.3 Output Error Revealing Conditions

We can now construct the output error revealing condition sets for the potential faults of node 1. A context error at node 1 may transfer to an output error through any of the def-use chains discussed above. This is described by the disjunction of the chain transfer conditions for each def-use chain — $[((a \geq b)$ and $(b \neq 0))]$ or $[(a < b)]$ or $[((z \neq 0)$ and $\mathbf{odd}(z))$ and $(a < b)]$. To construct the revealing condition set for a particular fault class, we conjoin each condition in the

[9] For simplicity, we will continue to use this sufficient constraint.

context error revealing condition set with this disjunction of the chain transfer conditions. For the fault classes considered above, this results in the following output error revealing condition sets.

incorrect reference to U

$$\{\,[\,(u \neq \bar{u}) \text{ and } (v \neq 0) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,] \mid U \in \{A, B, V, W, X, Y, Z\}\,\}$$

incorrect reference to V

$$\{\,[\,(v \neq \bar{v}) \text{ and } (u \neq 0) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,] \mid V \in \{A, B, U, W, X, Y, Z\}\,\}$$

incorrect arithmetic operator

$$\{\,[\,((u*v) \neq (u+v)) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,],\ [\,((u*v) \neq (u-v)) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,],\ [\,((u*v) \neq (u/v)) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,],\ [\,((u*v) \neq (u**v)) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,]\,\}$$

incorrect variable definition to X

$$\{\,[\,(x \neq \bar{x}) \text{ or } (u*v \neq x) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,] \mid \overline{X} \in \{A, B, U, V, W, Y, Z\}\,\}$$

3.4 Test Data Selection

If we blindly select test data to satisfy these revealing condition sets, then we might potentially select 25 test data. There is much overlap, however, between these condition sets. In fact, selection of a single test datum such that

$$\forall\ VAR \in \{A, B, W, Y, Z\} \mid [\,(u \neq v \neq x \neq var) \text{ and } (u \neq v \neq 0) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))]$$[10]

[10]Because this is the first node in the module, no additional conditions restrict the domain from which the test data are selected.

satisfies the revealing condition sets for both potential variable reference faults and the variable definition fault. The single test datum ($a = 2, b = 3, u = 4, v = 5, w = 6, x = 7, y = 8, z = 9$) satisfies this sufficient output error revealing condition for these two fault classes and transfers an error along the chain through the definitions of X at node 1, Y at node 3, and W at node 5. Upon examination of the revealing condition set for an arithmetic operator fault, we see that this test datum satisfies each condition in that set as well. This single test datum, therefore, is sufficient to guarantee detection of any fault in node 1 of these classes. This means that if execution of the module on this test datum provides correct results, we know that node 1 does not contain any fault of these classes. While we could have selected test data that executes all def-use chains, execution of such data would not provide any additional information about the presence or absence of faults at node 1 for the chosen fault classes. Under the assumption that the module is "almost correct", we have guaranteed that node 1 is correct with respect to the fault classes considered. This testing process must, of course, be undergone for all nodes in the module to guarantee detection of any such fault in the module.

4 Conclusion

In this paper, we present the RELAY model of error detection and demonstrate its use as a criterion for test data selection. The model itself defines generic origination and transfer conditions that must be satisfied to guarantee the detection of an error. To use RELAY as a test data selection criterion, a fault classification is chosen and the origination condition sets and the applicable transfer conditions are instantiated for that fault classification. For each node in a module, these origination condition sets are then evaluated for the potential faults. To develop the revealing condition sets, the applicable computational and data flow transfer conditions are added to each origination condition in the origination condition sets. Test data selected to satisfy these revealing condition sets is capable of guaranteeing detection of an error for any fault in the chosen fault classification.

This paper provides an example of the selection of test data for a chosen fault classification. Test data is selected for one node, which potentially has faults in three classes. The application of the origination condition sets is straightforward, and it is relatively easy to determine the data flow and computational transfer conditions for all def-use chains from the potential fault to output. In general, finding a sufficient output error revealing condition set is not difficult since it requires determining only a single def-use chain from the potential fault to an output. When test data is selected to satisfy a sufficient revealing condition set for some fault class, correct execution guarantees that the module does not contain a fault of the class. If that set is not satisfiable, however, absence of a fault in the class is not guaranteed; the complete revealing condition set must be considered. Determining the complete, necessary condition set, however, may be more difficult since it requires determining all possible def-use chains from the potential fault to output. This is particularly complex when a potential error transfers through a looping construct.

RELAY has also been used to evaluate the error detection capabilities of other testing techniques[RT86a]. This analysis demonstrated how the rules of a test data selection criterion must be carefully designed and tightly integrated to reveal an error for any potential fault by showing how other techniques have failed to accomplish this precision. Without this precise analysis,

it is easy to arrive at test data selection rules that do not guarantee the detection of a fault and may not even be sufficient to do so. Using RELAY, we have evaluated where previous criteria have failed in this regard.

We continue to extend the RELAY model of error detection, to evaluate its capabilities by instantiating it for other classes of faults, and to analyze other testing criteria using RELAY. In addition, we are investigating the implementation of a tool to automate the selection of test data based on the RELAY model.

REFERENCES

[Bud81] Timothy A. Budd. Mutation analysis: ideas, examples, problems and prospects. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 129–148, North-Holland, 1981.

[Bud83] Timothy A. Budd. *The Portable Mutation Testing Suite.* Technical Report TR 83-8, University of Arizona, March 1983.

[Gla81] Robert L. Glass. Persistent software errors. *IEEE Transactions on Software Engineering*, SE-7(2):162–168, March 1981.

[Ham77] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, July 1977.

[How78] William E. Howden. Introduction to the theory of testing. In Edward Miller and William E. Howden, editors, *Tutorial: Software Testing and Validation Techniques*, pages 16–19, IEEE, New York, 1978.

[How82] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(2):371–379, July 1982.

[How85] William E. Howden. The theory and practice of functional testing. *IEEE Software*, 2(5):6–17, September 1985.

[MH81] Larry J. Morrell and Richard G. Hamlet. *Error Propagation and Elimination in Computer Programs.* Technical Report 1065, University of Maryland, July 1981.

[Mor84] Larry J. Morrell. *A Theory of Error-Based Testing.* PhD thesis, University of Maryland, April 1984.

[RT86a] Debra J. Richardson and Margaret C. Thompson. *An Analysis of Test Data Selection Criteria Using the RELAY Model of Error Detection.* Technical Report 86-65, Computer and Information Science, University of Massachusetts, Amherst, December 1986.

[RT86b] Debra J. Richardson and Margaret C. Thompson. *A New Model of Error Detection.* Technical Report 86-64, Computer and Information Science, University of Massachusetts, Amherst, December 1986.

[Wey81] Elaine J. Weyuker. *An Error-based Testing Strategy.* Technical Report 027, Computer Science, Institute of Mathematical Sciences, New York University, January 1981.

[Wey82] Elaine J. Weyuker. On testing nontestable programs. *The Computer Journal*, 25(4), 1982.

[Zei83] Steven J. Zeil. Testing for perturbations of program statements. *IEEE Transactions on Software Engineering*, SE-9(3):335–346, May 1983.

Session 9

SOFTWARE CONCEPTS

"Turing: A Trustworthy Programming Language"
S. G. Perelgut, Holt Software Associates, Inc., and R. C. Holt, University of Toronto

"Machine-Aided Production of Software Tests"
Thomas J. Ostrand and Marc J. Balcer, Siemens Research Laboratories

"Bang Metrics: An Ongoing Experiment at Hewlett-Packard"
Robert W. Dea, Hewlett-Packard

Turing: A Trustworthy Programming Language

S.G. Perelgut
Holt Software Associates, Inc.
203 College Street, Ste. 305
Toronto, Ontario M5T 1P9

Prof. R.C. Holt,
M. Funkenhauser
Computer Systems Research Institute
University of Toronto
10 King's College Road, SF2001D
Toronto, Ontario M5S 1A4

ABSTRACT

Turing is a general purpose programming language designed to produce reliable, efficient programs. Developed in 1982-3 at the University of Toronto, Turing has a complete, formal semantic specification that is suitable for developing mathematical proofs of correctness for programs.

This paper provides an overview of the language and the specific features that make reliable programming feasible. A description of the formal semantic definition is provided with references to follow up with the complete definition. Some simple examples are provided.

An extension to Turing, Turing Plus, is also briefly described. Turing Plus is 100% compatible with Turing and adds features suitable for systems implementation projects. The resulting language is defined in terms of *clean*, *dirty* and *dangerous*. features.

INTRODUCTION

What is Turing?

Turing is a general purpose programming language designed for convenient development of reliable, efficient programs. The language was designed in 1982 by Prof. R. C. Holt and J.R. Cordy at the University of Toronto's Computer Systems Research Institute. The design includes an informal definition [1], a formal definition [2], a textbook [3] and various portable compilers and interpreters running on machines such as IBM compatible pc's, SUN Microsystems workstations, DEC VAX minicomputers, and IBM mainframes.

Turing is an evolutionary development within the Algol family of languages. Its most immediate ancestor is the Concurrent Euclid programming language, designed in 1979 from the Department of Defence specifications of the verifiable language Euclid [4]. Turing looks like Pascal with the unnecessary syntax removed and a set of useful enhancements added.

Why was it Developed?

Turing resulted from an effort to find an acceptable language for teaching introductory computing science courses at the University of Toronto. It became obvious in the early 1980's that the university was going to move from a batch environment running on an IBM /370 to some form of interactive programming environment for introductory courses. Since a change of environment involved a change in compiler, a number of questions were raised regarding the effectiveness of the current language.

The language used in the batch systems was a variant of PL/I. There were no comparable PL/I compilers for the interactive environments being considered. Although PL/I has the advantages of powerful language features and good support material in the form of texts and manuals, PL/I is a large, clumsy language with a number of ill-defined features that made introductory courses unnecessarily difficult. Thus, there was a movement within the department to switch languages.

One alternative considered was Pascal since it is small and elegant and there are a number of good texts and manuals describing Pascal. Pascal's international acceptance made it an obvious candidate but the poor string manipulation, limited numerical facilities, lack of local scopes, and missing dynamic

arrays made Pascal unsuitable for the intended courses. Finally, Pascal has some syntax problems such as the use of semi-colons and procedure headers that make it more difficult to teach in an introductory course.

BASIC was also considered as the introductory language but its limitations and inelegance ruled it out in spite of the simpler syntax. Concurrent Euclid was also ruled out for a complex syntax without convenient input/output mechanisms or numerical facilities.

The Main Design Goals

Designing a programming language is more art than engineering. The discussions mentioned above identified some major goals for the new language including: general purpose; easy to learn (and teach); reliable; syntactically clean; implementable; and enjoyable. The last is the most difficult to quantify although it is important if the language is to be useful.

A general language is one appropriate for a wide class of applications. Turing approached it by applying insight to maximize the breadth of applications similar to those supported by PL/I or Pascal. Another goal was to keep the language from being unwieldy or complex. Thus came about the first design rule:

RULE 1: *Maximize utility, minimize complexity*

Pascal was chosen as a basis for comparison due to its wide acceptance. In deference to the local perception of Pascal:

RULE 2: *More powerful than Pascal*

This rule arose from the various demands of the computing science professors to support algorithms that Pascal does poorly. This includes: text manipulation, graph algorithms, numeric algorithms and software engineering.

Generality led to a carefully selected set of features to overcome these problems without reducing the functionality from that of Pascal.

Examining features that are confusing to students led to some conclusions about how to make a language that is easy to learn and to teach. The most obvious problem is Pascal's unnatural syntax. This is particularly evident in the use of semi-colon which trips students frequently in the introductory classes. The

program headers of PL/I and Pascal are examples of verbiage that is only for the convenience for the compiler writer.

RULE 3: *No frills syntax*

Students learn faster when the notation looks familiar, leading to another rule:

RULE 4: *Cultural recognizability*

A person familiar with standard algebraic notation will expect similar notations to behave the same. For example:

x---y

has different meanings for different programming languages. A C-programmer will usually take this to mean the value of ***x-y*** and then decrement ***x*** but some will expect it to mean decrement ***y*** followed by the value of ***x-y***. Ada programmers take this to mean the value of ***x*** since the -- is take to represent an end of line comment. Turing applies the same meaning a junior high school student would expect, that of ***x-y*** without additional changes. Similarly, Pascal's definition rejects

if x **>** 0 **and** x **<** 100 **then**

in spite of the fact it is clear what is intended. (Pascal requires the sub-expressions to be bracketed [5]).

A third feature that improves learnability is consistency, expressed by:

RULE 5: *Language features with similar purposes should have similar syntax*

For example, **if** statements are terminated **end if**, **for** statements with **end for**, etc.

A final feature that improves learnability is captured by the *Levels of Mastery* rule:

RULE 6: *Learners should be able to start with a language subset and then learn increasingly larger subsets.*

By allowing students to learn small increments of language features, the inundation of information with many languages can be avoided.

The third design goal, reliability, means that Turing should help the programmer in developing software that accomplishes its specified task. Turing was designed to take into account the fact that some bugs are inevitable in any large program; it attempts to minimize the frequency and severity of these bugs. In particular, certain classes of bugs have been acknowledged in the design.

The first class of bugs stems from the cultural framework that all programmers bring with them. Turing was designed to meet the following rules:

RULE 7: *Easily overlooked errors should not lead to disaster.*

RULE 8: *No unpleasant surprises.*

When a person uses cultural background, education and common sense to conclude that a program has a particular meaning, then the language should not provide any surprises such as a different meaning. For example, the notational problems noted above lead to obvious intended meanings that some languages do not apply.

Early detection and notification of problems helps as well. Bugs that make it to production are expensive or disastrous. Turing is designed to warn of possible errors as early as possible, including checks for uninitialized variables, dangling pointers, and multiple references to a single storage location. Also, Turing has strong static type checking to determine if a variable is suitable for its stated use.

Finally, reliability exists only when the compiler produces a run time object module that behaves according to the programmers expectations and the language specification. The formal definition of Turing makes this more likely for Turing programs than any other language.

TURING LANGUAGE FEATURES

This section provides a general description of the Turing programming language. Complete details are available in [1]. Some features, designed specifically for reliable programming, are highlighted. This is followed by a brief discussion of the language enhancements collectively known as Turing Plus, providing a language as rich as Ada within a framework that is as simple as BASIC.

The Turing Programming Language

A Turing program consists of a mixture of declarations and statements that look similar to Pascal with some of the frills removed. Declarations and statements can be interspersed within the program, allowing variables with limited use to be declared textually near the place where they are used.

A Turing variable declaration looks like:

```
        var idList : Type [:= Expn]
or      var idList := Expn

(ie)    var i : int                 % uninitialized
        var s,t : string := ""      % both set to empty string
        var j := 0.0                % real value, initially zero
```

[Anything on a line following a **%** is treated as a comment and ignored by the compiler.] Declarations without types are only legal when the type is unambiguous. For example, **0** is an integer value whereas **0.0** is a real value. By shortening obvious cases and making variable initialization simple, programmers are more likely to enjoy using the language and are more likely to use the features that improve reliability.

Variables represent items that can have changing values. Turing acknowledges another class of values that cannot be changed. These are called constants and are declared similar to variables with the keyword **var** replaced by the keyword **const** as in:

```
const c := 42   % c is an integer, value 42
c := 1  % illegal!! c is a constant and cannot change.
```

Attempting a statement like the one shown above will result in a compilation time error message indicating the line and file where an attempt has been made to change the value of a constant variable.

Turing allows user defined types as well as providing the full range of predefined types listed below:

int	Integer. Defined to have at least 32-bits of representation ($-2^{31}-1$ to $2^{31}-1$).
real	Real numbers. The definition recommends 8 byte representation allowing large values with small relative roundoff errors.

boolean	Boolean values **true** or **false**.
string[(*Expn*)]	Character string values. A string has a set maximum number of characters with a recommended default of 255. The maximum can be user set as shown.
enum (*idList*)	Enumerated range over the list of identifiers provided.
Expn1 .. *Expn2*	Subrange. May be a subrange of either integer values or enumerated values. The value of *Expn1* must be less than or equal to *Expn2*
array	Multi-dimensional sequential list of similar items. The upper bounds of the array may be compile time (static array) or run time (dynamic array) values.
record	A non-scalar collection of items with varying types.
union	A form of record providing variant record capability in a verifiable fashion as described later.
set *Range*	A set represented as a bit-vector indicating the presence or absence of an element in the set. The Report recommends a minimum limit of 31 bits.
collection	A reliable form of dynamic storage reference that is explained later in this paper.
pointer	A reliable form of memory reference, used in conjunction with the **collection** type.

Statements and declarations can be mixed to allow declaration of variables to occur near their first use. This has been shown to make programs easier to write and read. Statements and declarations can appear in the main global scope or can be grouped into subprograms as procedures (capable of changing global and parametric values) and functions (result returning subprograms that behave as mathematical functions, not changing global values directly or indirectly). Subprograms may not be

nested within other subprograms, but data, statements and subprograms can be nested within a module (which may be part of a larger module). The concept of modules allows for data encapsulation and provides a simple, clean interface behind which implementation details are hidden.

Modules and subprograms have *closed scopes* (must explicitly import global variables and export entry points), much like Concurrent Euclid . The design of subprograms was improved by allowing the compiler to produce the list of imports, thus making subprograms appear to the programmer to be open scopes. In fact, the design and implementation treat all modules and subprograms as closed scopes requiring explicit instructions from the programmer on what data may cross the interface.

Subprograms may optionally have parameter lists. A parameter may be marked **var** to indicate that the value may be changed by the subprogram. Parameters not marked **var** are treated as constants by the procedure with the same restrictions given above. Parameters are passed by reference unless they are not marked **var** and they are one of the scalar types (**int, real, boolean, set, enum, pointer,** or *subrange*).

Functions, by definition, are restricted from changing global variables directly or indirectly. An example of a function is:

```
function Sum( source : array 1 .. * of real) : real
   var total := 0.                % real number, initially 0
   for i : 1 .. upper(source)
      total := total + source(i)
   end for
   result total
end Sum
```

This example declares a function named *Sum* that has a parameter named *source* that is an **array** with a run time upper bound. The array elements are all type **real** and the function returns a **real** result. The local variable *total* is declared and initialized to the **real** value 0. A loop counting from 1 to the upper bound of the parametric array is declared and *total* is increased by the indexed value within the array.

The function is terminated with a **result** statement to indicate that a value is returned. A corresponding procedure would not name a resulting value type and would terminate at the end of the procedure or with an explicit **return** statement.

Turing statements include:

assignment: := (also +=, -=, *=). This is simple assignment of a type compatible value to a variable. The indicated short forms are notations designed to shorten common forms of usage. Experience is showing that one of the most common uses is of the form:

v **+=** value

input/output **get** and **put**. These statements provide a simple and convenient form of I/O that is easy to use and more complete that Pascal. The simplest format is the **put** or **get** any numeric or character type. With slightly more sophistication, the user can add field widths, decimal display limits, etc.

subprogram Calling a subprogram invokes its name followed by an optional parameter list surrounded by parentheses. **return** and **result** are used to terminate subprograms as previously described.

selection There are two forms of selection statement in Turing, provided for convenience of notation. The first is the **if** statement which processes a boolean expression and chooses which alternative to execute. An **if** can include any number of **elsif** clauses which are checked in turn for a **true** result. The default case is indicated by **else**. The second form is a **case** where an integer value is checked against labeled statement groups. If the index is not matched, it defaults to either the **label** : case or, if there is no such case, it aborts at run time.

repetition There are two forms of repetition in Turing. The first is an infinite **loop** terminated by an **exit** statement. Turing also provides a counted **for** loop (which can also have an **exit** for premature termination). A counted loop operates over a *Subrange* type and declares the loop index, an unchangeable local variable for the scope of the loop.

Features for Reliability

A number of the language features are designed to enhance program reliability. For example, Turing allows dynamic storage with the **collection** and **pointer** types. A **collection** can be thought of as an array with elements created and deleted at run time. For example,

```
var c : collection of
   record
      value : int
      next, prev : pointer to c
   end record
var root : pointer to c := nil(c)
var p,q : pointer to c

new c, p
c(p).value := 42  % access to storage is identical to array
q := p      % pointers are assignable scalar values
free c, p  % NOTE: q now points to freed storage
```

The value of **nil(*c*)** is a null pointer that the compiler considers unique to the **collection** *c*. Elements are dynamically created and removed with **new** and **free**.

Faithful execution refers to the property of a program to behave at runtime exactly as the source specifies. A faithful execution involving dynamic storage allocation can be implemented by associating a unique timestamp with each new pointer and storage allocation. This resolves the *dangling pointer* problem of most other languages.

Turing also prevents two names from referring to the same storage location. This multiple naming is known as *aliasing* and can be the source of many subtle bugs in large programs. One simple way that this can occur is through **var** parameters in any language as

```
procedure p ( var x, y : int)
      . . .
var a := 42
p(a,a)
```

In this example, procedure *p* will refer to the global variable *a*'s storage with either the name *x* or *y*.

```
x := 1
y := 2
% x nows has the value 2 in spite of what common
% sense leads you to believe.
```

The change to *y* incidently changes the value of *x*. Turing will issue a warning at compilation time that aliasing is possible and a faithful execution will abort at run time if this does happen.

Turing implements a **module** as a closed scope that must *explicitly* **import** any global variables, subprograms or modules that it uses. Subprograms are also closed scopes that must import externally defined items although the **import** can be *implicit* (calculated by the compiler). The following is a simple example that shows how modules and subprograms can be used.

```
const increment := 2
module Sample
   import (increment)     % must explicitly import globals
   export (Count, Display) % must explicitly export entries

   var total := 0      % local to module

   procedure Count(numberOfIncrements : int)

      total += increment * numberOfIncrements
   end Count

   function Display : int      % parameterless function
      result total     % total implicitly imported
   end Display
end Sample
   . . .
Sample.Count(3)
put Sample.Display     % will output 6 in this example
```

An import list represents the *direct imports* to the closed scope. *Indirect* (*transitive*) imports refer to all items directly imported and, recursively, all items imported by closed scopes that are directly imported. The transitively closed import list is used to calculate potential aliasing conflicts. The following rules (enforced at compilation and run time) will prevent aliasing problems.

a) (a part of) a variable may not be passed to a **var** parameter of a subprogram if the subprogram directly or indirectly imports (a part of) the same variable or if (a part of) the same variable is passed to another parameter of the subprogram.

b) (a part of) a variable cannot be passed by reference if the called subprogram directly or indirectly imports the variable **var** or if the variable is passed to a **var** parameter of the subprogram.

Another source of potential aliasing is Turing's **bind** statement, used to rename a variable or part of a variable. **bind** is typically used to assign a short name to an element of an array or record as in:

```
var listOfNames : array 1 .. 100 of string
. . .
bind var x to listOfNames(i), y to listOfNames(j)
```

A **bind** may be **var** if the item named is going to be changed. The above example would be an alias if i=j. The following rules are used to avoid aliasing due to binding.

c) a **var bind** of name *y* to (a part of) variable *x* makes *x* inaccessible for the scope of *y*.

d) a **var bind** to *x* disallows calls to subprograms or modules that directly or indirectly import *x*.

e) a constant **bind** to *x* makes *x* read-only for the scope of the **bind**.

Reliability is also enhanced by statements that act as run time enforceable comments. All of these statements evaluate a boolean expression at run time in a faithful execution and abort if the result is **false**.

The simplest form is the **assert** statement that can appear wherever any statement can legally appear. **assert** is used to denote a critical assumption on the programmer's part that can be represented as a boolean expression that is evaluated at run time. For example,

```
assert x=0 or 1/x < 1
```

An **invariant** is used to denote assumptions about things that don't change during loop execution or through the call to a module from an external routine. An **invariant** in a loop might look something like:

```
loop
    invariant numberOfNames = 20
  . . .
end loop
```

pre and **post** are used to state assumptions that must be true on entry to and exit from a subprogram. These are very useful to describe exactly what is assumed by a subprogram, acting as a guarantee by the programmer that the following will be true throughout the subprogram. For example,

```
procedure(var a : array 1.. * of int,
          index : int)
    pre index > 1 and index < upper(a)
    post a(index) = 42
    . . .
```

A final reliability feature to be described is the type-safe form of variant records known as **union**. **union** is a record with multiple bodies, selected by the value of the **tag** field in the **union** header. Faithful execution checks the **tag** whenever an element of a **union** body is referenced. Whenever the **tag** is changed (using a **tag** statement), faithful execution will re-initialize the data storage.

Implementation Details

Faithful execution requires a number of space and time consuming checks. For example, all variable references are checked before the value is used to make certain that the variable has been initialized. All the tests a good checking compiler emits are standard for Turing (subscript bounds, subrange out of value, division by zero, overflow, underflow, etc.). Tests unique to Turing are also emitted, such as run time anti-aliasing checks, invalid **union** references, dangling or freed pointer, etc.

For efficiency, the implementation of the Turing compiler allows these tests to be optionally omitted. Thus, a programmer can check software with the aid of the verification tools and then remove the run time checking with the increased level of trust that the software performs as expected.

Turing Plus Extensions

Turing Plus is the name of a compatible extension to Turing. It adds features such as separate compilation, random access input and output, concurrency, exception handling, bit manipulation and assembly language inserts. Turing Plus recognizes the benefit of faithful execution and preserves this as much as possible in the rich set of added features. It does so by defining three levels of access (*visibility*).

The first level is *data visibility*. This allows the programmer to consider the bits assigned to a variable of one data type as if they belonged to a variable with a different data type. For example, the 32 bits representing an integer can be considered as 4 characters of a **string**. This may affect the portability of the code since some bit patterns are unique to an implementation of a data type and may vary from implementation to implementation. Data visibility is considered *dirty* but it will not cause remote corruption of code or data.

The second level is *address visibility*. This allows explicit access to machine addresses through the use of non-type safe pointers as in

```
p := addr(x)      % addr returns the address of a variable
int4@(p+2) := 5 % store the 4 byte integer representation
                  % of the value 5 into a machine address
                  % two storage unit past the start of x
```

This requires knowledge of the specific implementation and can corrupt both data and code. In general, any type followed by an **@** followed by a parenthesized integer expression means that the value of the expression will be treated as a machine address and the bits starting at that address will be treated as the given type. This is considered *dangerous* since it can corrupt remote elements.

Code visibility refers to the **asm** statement, allowing the direct inclusion of machine code as part of the program text. This is a means of inserting highly efficient machine code into a time or space critical section of the program. It is *dangerous* since it can corrupt remote code and data, and it is implementation specific.

Turing Plus refers to all features as either *clean*, *dirty* or *dangerous*. Clean features can be either *checkable* (when a checking compiler is used, execution becomes faithful) or

uncheckable (even when certain constraints have been exceeded, execution does not corrupt remote code or data). Dirty features are at least partially implementation dependant and as such cannot be mathematically defined. Dangerous features are dirty features that can cause corruption of remote code or data.

A complete specification of the Turing Plus extensions is available in [2] This paper will deal with only a couple of relevant and interesting features.

The first feature is separate compilation which imposes a form of directed acyclic graph structure on the overall program. Turing Plus divides each separately compiled source into an interface **stub** and an implementation **body** (with any number of programmer defined alternative bodies for various implementations of the interface). Separate compilation performs full type checking across the interface. Any data referenced across the separate compilation must be in the **grant** list of the module. As an example, consider the following stack manager.

```
stub module Stack % Interface for stack
   export (Push, Pop)
   procedure Push (el: int)
   procedure Pop (var el: int)
end Stack

body module Stack % This can be separately compiled
   const numElements := 50      % limit stack size
   var list : array 1 .. numElements of int
   var top : 0 .. numElements := 0

   body procedure Push          % (el: int)
      top += 1
      list(top) := el
   end Push

   body procedure Pop           % (var el: int)
      el := list(top)
      top -= 1
   end Push
end Stack
```

An alternate implementation of *Stack* can be created and separately compiled. This allows multiple implementations that can be useful when testing code before completing all modules. It is also useful when one version uses statically declared arrays versus dynamically allocated collections, as shown below.

```
body "stack.st" module Stack   % Alternate Stack
   var c : collection of
      record
         value : int
         prev : pointer to c
      end record
   var top := nil(c)

   body procedure Push
      var oldTop := top
      new c, top
      c(top).value := el
      c(top).prev := oldTop
   end Push

   body procedure Pop
      const oldTop := top
      el := c(top).value
      top := c(top).prev
      free c,oldTop
   end Push
end Stack
```

The next feature of Turing Plus is the in-line assembly language facility. This is a *dangerous* but useful tool for critical code sections. Code sequences are emitted by the code generator with variable references and expressions converted. For example,

```
var vlong, v2 : int4     % int4 is a 4 byte integer in Plus
asm "movl", 5, vlong     % equivalent to vlong := 5
```

Full Turing Plus expressions and variable references can appear as operands of the **asm** instruction where appropriate. The compiler will resolve them to addressable references of the target machine language and produce the correct assembly language output. The programmer must make certain that the code emitted can be handled since circumstances (such as an expression that consumes too many temporary registers) may have unexpected side effect or may fail altogether. It is the user's responsibility to make certain a dangerous feature does what is expected.

Finally, the concurrency features are worth brief mention. Concurrency is provided by means of statically defined **process** statements that can be thought of as subprograms that execute concurrently when invoked by a new statement, **fork**. An example of a simple process and its invocation is:

```
process Speak ( phrase : string)
   loop
      put phrase
   end loop
end Speak
   . . .
fork Speak("Hi")
fork Speak("Ho")
```

This will start two concurrently active versions of the *Speak* process each intermittently outputing the given phrase.

Shared data is protected by use of a **monitor** which is very similar to a **module** except that only one version of one entry point of the **monitor** can be active at any time. When more than one **process** attempts concurrent access to the **monitor**, all but one are blocked on a queue until the one active **process** exits the **monitor** scope or becomes inactive by executing a **wait** statement which puts it on a queue waiting for a **signal** of the queue's **condition**. An excellent description of how this form of concurrency increases program trustworthiness and how it works is available in [6].

FORMAL DEFINITION

The complete formal definition of the Turing programming language is beyond the scope of this paper. For a complete definition read [2]. This provides a brief overview concentrating on how the definition was developed followed by a look at some elements of the definition, how they influence reliability, and some implementation issues.

Why have a formal definition?

A language definition should meet the goals of completeness, clarity and consistency. To be complete, the definition should discuss all relevant aspects of the language. Clarity requires a definition that is easy to read and understand without being subject to interpretations. It should also be easy to implement for practical languages. Having a single interpretation for every aspect of the definition is the key to consistency.

Turing has a complete definition with the following minor exceptions to make the definition practical to use. First, Turing provides a number of predefined subprograms that are omitted from the definition in the interest of brevity. Completeness would be achieved at the expense of bulky definitions that would obscure the overall document. The formal definition also requires explicit imports for subprograms ignoring the practical decision to allow implicit imports. This does not detract from the verifiability of a program since import lists can be made explicit through a simple mechanical process. Related to this is the use of **put** and **get** I/O statements. These are assumed to require the explicit import of an input/output module. Again, this can be added to the program text in a mechanical fashion and is thus only a minor matter. Finally, the convenience of the **init** construct for initializing non scalar data elements is omitted for simplicity's sake.

The clarity of the Turing language definition has been helped by the previously mentioned omissions and by the overall design goal of simplicity. The definition (as presented in [Ref. D&D]) is reasonably understandable although it does point out the inherent complexity of a general purpose programming language. It is possible that further research in language formalization will yield better techniques in the future.

The third goal is consistency. Turing's definition is consistent except for the resource exhaustion issues which are not completely expressed mathematically. The formal definition generally assumes execution on "God's Own Machine" which never runs out of memory, etc. with the provision of a basis for handling real machine limitations.

The formal definition does not deal with the Turing Plus extensions to the language. This is viewed as a shortcoming that requires more research to address.

The Formal Syntax

Most of the formal syntax follows the well known rules for defining a context free syntax to define the phrase structure of the language. The formal definition extends this with the definition of a function *syn* that acts on strings. *Syn* returns a boolean result indicating whether the string is legal Turing.

Given Σ, the set of all characters (Σ^* is the set of all finite strings)

syn: Σ^* -> (**true**, **false**)

if S is an element of Σ^* then S is a Turing program iff syn(S) = **true**

Syn is defined in terms of three subfunctions, *lex*, *cfs* and *cc* defined as follows.

lex(*S*) is **true** iff *S* is a lexically legal Turing program. Practically, we think of *lex* as the lexical analyzer that takes a string of characters, *S*, as input and produces a string of lexemes (tokens). Each token is one of: keyword (**procedure**), identifier (*x*), explicit constant (42, "Hello", etc.), or a special symbol (**:=**, **+**, etc.). If *lex*(*S*) is true, then we consider *S'* to be the tokenized representation of *S*.

cfs(*S'*) is **true** iff *S'* satisfies the context free syntax of Turing. Turing has an LR(k) [Ref. Aho 77] formal context free syntax that can be used to automatically generate a parser and is guaranteed to be unambiguous. When *cfs*(*S'*) is true, the strings can be considered to be mapped into *S"*, a parsed structure representing *S*.

cc(*S"*) maps the context conditions (sometimes referred to as the context sensitive syntax). Context conditions impose further restraints such as type compatibility that are awkward to include in the context free syntax. Turing has a large *cc* definition, a result of the inherent complexity of a general purpose programming language.

Together, these form *syn* as follows

syn(*S*) = (*lex*(*S*) & *cfs*(*S'*) & *cc*(*S"*))

Thus the formal syntax is managed much like a compiler is written, with a lexical analyzer transforming characters into tokens that are verified by a parser as it transforms the tokens into a parsed representation that is then used by a semantic content analyzer. This helps provide a constructive basis for the Turing language translators.

Informal Semantics

The informal semantic definition of the semantics of Turing programs is a non-mathematical presentation designed to be easily read by most programmers. Each construct is explained with examples where it is defined.

While the informal definition is intended to be complete, it relies on the reader's common sense in place of a rigid, formal definition. As a result, some parts of the Report ([Ref. TLR] defines the informal semantics) may be open to misinterpretation.

The benefit of an informal definition is more than simple readability. It also provides an operation point of view: which actions occur as a result of executing each construct. This allows the informal report to easily handle resource exhaustion constraints and to discuss implementation concerns such as the size of integers. For example, the Report recommends strings be allowed to have at least 255 characters. The informal report also recognizes the possibility of various standards such as ASCII versus EBCDIC.

Finally, an informal definition allows for some practical extensions that are difficult or impossible to formalize. For example, the informal definition allows for **external** subprograms to be defined in Turing and implemented in another language.

Formal Semantics

The formal semantics of Turing maps the set of all syntactically legal Turing programs into the set of all program meanings. Thus we want a mapping of the form:

$$sem : T \rightarrow M \text{ where } T=\{S \text{ element of } \Sigma^*: syn(S) = \textbf{true}\}$$

The approach chosen is to provide *proof rules* for each statement. Taken together, these rules determine how to prove the correctness of a program.

The formalization of Turing uses ten *basis* statements. All Turing statements and declarations are defined in terms of these ten which are in turn defined by both a formal axiomatic definition and a formal operational semantics.

Informally, the ten basis statements are:

CONTINUE	the null statement
PICK(x,s)	Assign any non-empty set s to variable x
IF e THEN S1 ELSE S2 END IF	Choose either of S1 or S2 based on the boolean value of e
S1; S2	S1 followed by S2
ABORT	Program failure followed by immediate halt
CHAOS	Program failure with arbitrary continuation
BLOCK S END BLOCK	Bracketing to support exiting from S
SUBPROG S END SUBPROG	Bracketing to support returning from S
EXIT	Terminate (jump to end of) enclosing BLOCK
RETURN	Terminate (jump to end of) enclosing SUBPROG

A formal axiomatic semantics of these basis statements is given in [Ref. D&D] using weakest preconditions [Ref. Hoare, Hehner]. Simplified, a weakest precondition is the predicate required such that execution of statement S will result in the established goal R. For example, if we take a program that calculates area and volume of a cube, the goal can be represented as:

R: A=L*W & V=H*A

Considering the statement S:

S: V := H * A

the weakest precondition would be:

wp(S,R) = (A=L*W)

And if S was preceded by statement U:

U: A := L * W

then we would state the weakest precondition for U as:

wp(U, (A=L*W)) = **true**

Notice that the established goal of statement U is the precondition for establishing the goal R of the following statement. This example ignores some possibilities that a complete definition must keep account of such as the problem of uninitialized values for L, W, or H. To handle such cases, validity predicates are defined such as *EXPN*, the validity predicate which is true iff the expression is valid.

EXPN(H*A) = (INITIALIZED(H) & INITIALIZED(A))

INITIALIZED is another validity predicate that is true iff the variable has a value. Thus, the first example of weakest precondition more properly becomes:

wp(V := H*A, V=H*A & A=L*W) =
(EXPN(H*A) & A=L*W)

This formal approach has been expanded to provide a complete axiomatic semantic definition of Turing. An appendix to [Ref. D&D] also provides a formal operational definition of the basis statements. The approach is to define a state by state progression that models program execution. This operational model is called *next state semantics*.

The operational approach assumes that there is a set of variables and that the combined values of these variables defines the state of the computation. Each variable can assume a value from a set of possible values which include checking conditions such as the uninitialized value.

x,y, etc. are variables and v is the combination of all variables
T_x is all legal values for variable x given its type
$\Sigma_x = T_x + \{UNINIT(T_x)\}$

The overall state space, Σ is defined
$\Sigma = \Sigma_x + \Sigma_y + \ldots$

Thus each statement is a mapping from the overall state space to a pair formed from all possible sequences of states and a status to indicate the operational result in terms of the set {CONTINUE, EXIT, RETURN, ABORT}. *trace* is used to represent the sequence of states and *status* denotes which of the four values above. The notation becomes

v S (*trace*, *status*)

which is read as (*trace*, *status*) **in** S(v).

For example, the CONT and PICK basis statements can be defined by:

v CONT (*trace*, *status*) $=_{df}$ trace=<v> & status=CONT

v PICK(x,s) (*trace*, *status*) $=_{df}$
There is some e in s such that
trace=<v(x:e)> & status=CONT

The PICK definition means that trace is set to the variable space with the variable x set to the value e taken from set s.

EXPERIENCE AND CONCLUSION

Turing follows in the footsteps of Concurrent Euclid, a previous "verifiable" programming language that offered some tools for proving programs without a full, formal definition of the programming language. Concurrent Euclid has been used to write a complete, UNIX file-system compatible operating system. This is documented in a textbook [Ref. CE, U & T] and [Ref. TUNIS Tech Rep.]. Parts of the operating system have been verified by hand although the overall program would be a daunting prospect.

Turing's formal definition allows for a more reasonable project although the language lacks some features that large, time critical systems require. To date, there have been no large projects verified using the Turing programming language although there are a number of large software tools that might be suitable.

Turing has been used to write a number of large pieces of software including a visual editor and an interpreter for the Turing programming language. The speed with which these projects came about and the few problems encountered have been taken as a sign that programming in Turing benefits from the tight language definition even when the verifiability is not being used.

We surmise that the concise, general language and the benefits construed by a formal definition written in conjunction with the typical informal specification have made Turing a superior language. It is easy to learn, being used to teach all introductory computing science courses at the University of Toronto. (The first term Turing was used, the standard 26 lecture hour course was completed in 22-24 lecture hours.) The existing large software projects have shown that it is suitably general purpose and is efficient enough for most applications.

Turing Plus has added *dirty* and *dangerous* features with carefully defined informal semantics. The result is a very attractive and enjoyable language that has been used to write an efficient compiler for itself. Turing Plus programs benefit from the improved trustworthiness from Turing with the programmer having powerful tools for discretionary use.

A project is also underway to produce a secure, updated version of the TUNIS operating systems written in Turing Plus. It is expected that parts or all of this operating system may be verified to meet some level of Orange Book [Ref. ???] specifications.

REFERENCES

[1] *The Turing Language Report*, J.R. Cordy and R.C. Holt, CSRI Technical Report 153, Computer Systems Research Institute, December, 1983

[2] *The Turing Programming Language: Design and Definition*, R.C. Holt, P.A. Matthews, J.A. Rosselet and J.R. Cordy, Prentice-Hall, 1987

[3] *Introduction to Computer Science Using the Turing Programming Language*, R.C. Holt and J.N.P. Hume, Prentice-Hall Publishing Company, Reston, Virginia, 1984

[4] *Specification of Concurrent Euclid*, J.R. Cordy and R.C. Holt, CSRI Technical Report 133, Computer Systems Research Instutute, August, 1981

[5] *Pascal User Manual and Report*, K. Jensen and N. Wirth, Springer-Verlag, New York, 1974

[6] *Concurrent Euclid, Unix and Tunis*, R.C. Holt, Addison-Wesley, 1982

Machine-Aided Production of Software Tests

Thomas J. Ostrand
Marc J. Balcer

Software Technology Department
Siemens Research and Technology Laboratories
105 College Road East
Princeton, NJ 08540

Abstract

Effective and thorough functional tests for large software systems can be designed despite limitations on testing time and resources. This paper describes a method for creating functional test suites in which a test engineer analyzes a software system's specification and writes a series of formal test specifications. A tool generates test descriptions from the test specifications, and the engineer then transforms the descriptions into executable test scripts. A particular advantage of the method is that the tester can easily control the complexity and number of the tests by modifying the test specifications.

Authors' Biographies

Thomas J. Ostrand is a Senior Research Scientist at Siemens Research and Technology Laboratories. He is currently designing and building a prototype interactive test development environment for the definition and administration of specification-based tests for large, evolving software systems. His interests also include software test thoroughness criteria and formal requirements and specification tools. Dr. Ostrand received his S.B. in mathematics from M.I.T., and M.S. and Ph.D. in computer and information sciences from the University of Pennsylvania. He was formerly a member of the computer science faculty at Rutgers University, and a research scientist with Sperry Univac.

Marc J. Balcer is a Software Engineer at Siemens Research and Technology Laboratories. He is part of a team developing an innovative integrated programming support environment. Mr. Balcer received degrees in Computer Science from the College of William and Mary (B.S., 1982) and Villanova University (M.S., 1986). He is currently also a member of the faculty of Beaver College in Glenside, Pennsylvania.

1. Methods of Functional Testing

The purpose of functional testing of a software system is to find discrepancies between the behavior of the implemented system's functions and the behavior described in the system's functional specification. This is usually carried out by test cases that attempt to exercise all the system's externally defined functions. A commonly used method for defining such test cases for a function is to find an appropriate partition of the function's input domain, and then select test data from each class of the partition. Various methods of carrying out this partitioning have been described in the testing literature [1, 4, 5, 6].

Finding the appropriate partition is frequently difficult, especially in situations where there are many different parameters and environment conditions that can affect the way a function behaves. In this paper we describe a systematic specification-based method that uses partitioning to generate functional tests for complex software systems. The method is supported by the concept of a formal *test specification* and by a *generator tool* that produces test case descriptions from test specifications.

The method begins with the decomposition of the functional specification into *subcomponents* that can be executed independently (programs). The next step is the decomposition of each subcomponent into *functional units* that can be tested independently. A functional unit can be either a top-level user command or an internal function that is called by other functions of the system.

After the functional units of a subcomponent have been identified, the next decomposition step is to identify the *parameters* and *environment conditions* that affect their execution behavior. Parameters are the explicit inputs to a functional unit, supplied by either the user or another program. Environment conditions are characteristics of the system's state at the time of executing a functional unit, i.e., any influence on the unit's behavior that is not a parameter.

The actual test cases for a functional unit consist of values of the parameters and environment conditions, chosen to maximize the chances of finding errors in the unit's implementation. Exhaustive testing of each function on all combinations of each parameter value and environment situation is almost always impossible. Instead, the tester continues the decomposition process by finding *categories* of information that characterize each parameter and environment condition. The final decomposition step is to partition each category into distinct *choices* that include the significant aspects of the category. This last step corresponds to the partitioning discussed in the references mentioned above.

Once the partition classes have been established, a test case for a decomposed function can be specified by choosing a representative from each class. A good set of test cases for a function will include many combinations of elements from the classes,

but not so many that the testing process becomes prohibitively expensive. The determination of what level of testing constitutes "prohibitive expense" is a pragmatic decision that depends on the criticality of the software, the project's budget, and perhaps on the skill levels of the programmers and the testers.

The testing strategy and tool described in this paper were designed to create test cases for an interactive program development environment, whose components are a text editor, a source-language debugger, and a version control/configuration management system. These components contain over 200 different user commands, with an average of four parameters per command. The behavior of a command is affected by an average of five environment conditions. Suppose we make the very conservative assumption that each parameter and environment condition has only a single important category to consider. If we also assume that each of the resulting nine categories (four parameters and five environment conditions) can be partitioned into 3 choices, there are 3^9 (19,683) potential combinations to test for each command. Note that these combinations by no means constitute exhaustive testing, but merely represent complete coverage of the combinations of three representative choices from each category.

However, most testers would probably agree that 19,683 tests for a typical program development environment command exceeds the "prohibitively expensive" limit. The tester must find a way to reduce the number of tests, while still retaining confidence in their ability to expose most errors in the code. The method and tool described in this paper give the tester a way to express all combinations of choices from the categories of parameters and environment conditions, and then to restrict the generation of tests to those considered most significant and most likely to expose errors.

2. The Category-Partition Method for Test Case Generation

Despite the growing use of formal requirements and specification languages [2, 3], many software specifications are still written in natural language. These documents are frequently wordy, unstructured, and difficult to use as a basis for functional testing. The tester must transform the written specification into some intermediate representation, from which test cases and test scripts can then be generated.

Our intermediate representation is a *formal test specification* that is written for each individual function to be tested. In this formal specification, the tester can identify and describe the causes that affect the function's behavior. In addition, the test specification is used to describe constraints among the causes. A *generator tool* processes the test specification for each function, producing descriptions of a set of test cases for the function. The constraint information is used to control the potentially astronomic number of input combinations.

Since the key steps of the method are the identification of categories of influences on the functions being tested, and the partitioning of the categories into choices, we call it the **category-partition method.** It proceeds in the following steps:

1. **Analyze the specification -** The tester identifies individual functions or commands of the software system that can be separately tested. For each function, the tester identifies

   - its parameters, and major characteristics of each parameter
   - any objects in the environment whose state could affect the function's operation, and characteristics of these objects

   The tester then classifies these items into *categories*; each category should be a characteristic of a parameter or the environment that has some effect on the behavior of the function.

   This information is found by careful reading of the specification. It is frequently helpful to mark off phrases in the specification that describe behavioral aspects of the function or environmental conditions. Unclear or missing descriptions of a function's behavior are frequently discovered during specification analysis.

2. **Partition the categories into choices -** The tester determines the significantly different cases that can occur within each parameter/environment category. This step requires a reasoned judgement based on the tester's specific knowledge of the software product, as well as his past experience in selecting good test cases.

3. **Determine constraints among the choices -** The tester decides how the choices interact, how the occurrence of one choice can affect the existence of another, and what special restrictions might affect any choice.

   This and the following two steps frequently occur repeatedly as the tester refines the test specification to achieve the desired level of functional tests.

4. **Write and process the test specification -** The category and choice information is written in a formal Test Specification Language (TSL). The written specification is then processed by a generator that produces a set of test frames for the function. A *test frame* is essentially a specification for a test case. It describes general characteristics of the parameters and environment that are to be satisfied by one test case. To create the actual test cases, the tester supplies specific input values and environment conditions that meet the requirements of the frames.

5. **Evaluate the test case generator's output** - The tester examines the output of the generator and determines if any changes to the test specification are necessary. Reasons for changing the test specification include the absence of some obviously necessary test situation, the appearance of impossible test combinations, or a judgement that too many test cases have been produced. If the specification must be changed, Step 4 is repeated.

6. **Transform into test scripts** - When the test specification and the tool's output are stable, the tester converts the test frames produced by the tool into test cases, and organizes the test cases into test scripts. Information from the parameter categories of a test frame determines the choice of specific inputs for the functional unit. The environment categories guide the establishment of the system state when the test is run.

A function's test specification consists of a list of the parameter and environment categories of the function, and the partition choices within each category. The test frames produced by the generator are tuples of choices, with each category in a test specification contributing either zero or one choice. If a specification consists only of categories and choices, with no constraint information, it is said to be *unrestricted.* Each test frame generated from an unrestricted test specification contains exactly one choice from each category. The total number of frames generated from an unrestricted specification is equal to the product of the number of choices in each category.

3. Using the Test Specification Language

Figure 3-1 is the natural language specification for a simple **find** command that searches for occurrences of a pattern in a file. The remainder of this section illustrates the steps of the category-partition method for this command.

Step 1: Analyze the specification. The **find** command is an individual function that can be separately tested. Its parameters are *pattern* and *file.* Based on the description in the specification, we decide that the pattern parameter's major characteristics are

- the length of the pattern
- whether the pattern is enclosed in quotes
- whether the pattern contains embedded blanks
- whether the pattern contains embedded quotes

For the file parameter, we will assume that all files in the system are "text" files that contain the same type of information as the search pattern. This leaves the existence

Command:
Find

Syntax:
find <pattern> <file>

Function:
The find command is used to locate one or more instances of a given pattern in a text file. All lines in the file that contain the pattern are written to standard output.

The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes ("). To include a quotation mark in the pattern, two quotes in a row ("") must be used.

Examples:
find john myfile
displays lines in the file **myfile** which contain **john**

find "john smith" myfile
displays lines in the file **myfile** which contain **john smith**

find "john"" smith" myfile
displays lines in the file **myfile** which contain **john" smith**

Figure 3-1: Natural language specification for the FIND command

or non-existence of the file its only significant characteristic as a parameter.

The object in the environment that affects the **find** function's operation is, of course, the file; the important characteristic is the occurrence of the pattern in the file. Significant categories are

- number of occurrences of the pattern in the file
- number of occurrences of the pattern in a line that contains it

Step 2: Partition the categories. In partitioning the function's categories, it is important to include any situation that might reasonably occur, plus some that might not be expected. The latter type of situations will become the basis for error tests for the function. For the **find** function, the categories are partitioned as shown

```
#  Test specification for the FIND command

Parameters:
    Search string size:
        empty
        single character
        many character
        longer than any line in the file

    Quoting:
        string is quoted
        string is not quoted
        string is improperly quoted

    Embedded blanks:
        no embedded blank
        one embedded blank
        several embedded blanks

    Embedded quotes:
        no embedded quotes
        one embedded quote
        several embedded quotes

    File name:
        good file name
        no file with this name
        omitted


Environments:
    Number of occurrences of pattern in file:
        none
        exactly one
        more than one

    Number of pattern occurrences on line containing pattern:
        one
        more than one
```

Figure 3-2: Category partitions for **find**

in Figure 3-2. Note that the categories are divided into the two broad groups of **Parameters** and **Environments**. Lines beginning with # are comments.

The information in this figure could serve as an unrestricted test specification, but it will generate 1944 test frames, many of which express situations that are impossible to achieve. The frame in Figure 3-3 is an example. According to this frame, the pattern parameter is supposed to be an empty string that contains several embedded

blanks; in addition, there are supposed to be no occurrences of the pattern in the file, but one occurrence on the target line.

```
Search string size : empty
Quoting : string is quoted
Embedded blanks : several embedded blanks
Embedded quotes : no embedded quotes
File name : good file name
Number of occurrences of pattern in file : none
Number of pattern occurrences on line containing pattern : one
```

Figure 3-3: Test frame with impossible requirements

Step 3: Determine constraints among the choices. Constraints on the use of choices are indicated with *properties* that can be assigned to certain choices, and tested for by other choices. Each individual choice in a test specification can be annotated with *property lists* and *selector expressions.* A property list is simply a list of properties that are associated with every test frame that includes the annotated choice. The form of a property list is

`[property A, B,...]`

where A, B,... are individual property names.

A selector expression is a conjunction of properties that were previously assigned to other choices. An expression tells the test frame generator not to include the annotated choice in a test frame unless the properties in the expression are associated with choices that are already in the test frame. A selector expression is assigned to a choice by appending a suffix of the form

`[if A]`

or

`[if A and B and ...]`

to the choice.

Figure 3-4 shows the **find** command test specification annotated with property lists and selector expressions. Since we want to distinguish empty patterns from non-empty ones, the two properties **Empty** and **NonEmpty** are assigned to the appropriate choices of the category **Pattern size.**

When the generator encounters a choice with a selector expression, it omits that choice from any tuple which does not have the specified property. For example, all of the choices in the categories **Embedded blanks** and **Embedded quotes**, as well as the last two in **Quoting** would never be combined with the choice **Search string size: empty.** In addition, the second and third choices in **Embedded blanks** would only be combined with a tuple that contains the choice **Quoting:**

```
# Test Specification for the FIND command
# Modified:  property lists and selector expressions added

Parameters:
    Search string size:
        empty                                   [property Empty]
        single character                        [property NonEmpty]
        many character                          [property NonEmpty]
        longer than any line in the file        [property NonEmpty]

    Quoting:
        string is quoted                [property Quoted]
        string is not quoted            [if NonEmpty]
        string is improperly quoted     [if NonEmpty]

    Embedded blanks:
        no embedded blank               [if NonEmpty]
        one embedded blank              [if NonEmpty and Quoted]
        several embedded blanks         [if NonEmpty and Quoted]

    Embedded quotes:
        no embedded quotes              [if NonEmpty]
        one embedded quote              [if NonEmpty]
        several embedded quotes         [if NonEmpty]

    File name:
        good file name
        no file with this name
        omitted


Environments:
    Number of occurrences of pattern in file:
        none                        [if NonEmpty]
        exactly one                 [if NonEmpty] [property Match]
        more than one               [if NonEmpty] [property Match]

    Number of pattern occurrences on line containing pattern:
        one             [if Match]
        more than one   [if Match]
```

Figure 3-4: Specification with property lists and selector expressions

`string is quoted`.

When a choice is annotated with both a selector expression and a property list (e.g., the choices `exactly one` and `more than one` in the Environments section) both are applied independently; the selector expression is applied to restrict the test frames that will include the choice, and the properties in the list are assigned to any tuples

containing the choice.

The restricted specification of Figure 3-4 reduces the number of resulting test frames to 678; this is one-third of the original number, but still too large to be economically and practically feasible for a fairly simple operation like **find**.

Steps 4 and 5: Write test specification and evaluate generator output. If we examine the test frames produced from the restricted specification, we find that many are redundant; they test the same essential situation. They differ only in varying parameters that have no effect on the command's outcome. This occurs frequently when some parameter or environment condition causes an error. For instance, every **find** command for which the named file does not exist will result in the same error, regardless of the form of the pattern.

TSL allows the special annotation `[error]` to be attached to a category choice. `[error]` tests are designed to test a particular feature which will cause an exception or other error state. It is assumed that if the annotated parameter or environment has this particular value, any call of the function will result in an error. A choice marked with `[error]` is not combined with choices in the other categories to create test frames. The generator creates a test frame that contains only the `[error]` choice. The test case defined from such a frame must fulfill the statement of the marked `[error]` choice, but the tester can set the test's other parameters and environment conditions at will. Adding four `[error]` markings to the restricted specification, as in Figure 3-5, reduces the number of tests to 125.

The number may be further reduced by using `[miscellaneous]` markings. This annotation is intended to describe special, unusual, or redundant choices that do not have to be combined with other choices. As with `[error]` choices, the generator does not combine a `[miscellaneous]` choice with any choices from other categories. A single frame is produced that contains only the marked `[miscellaneous]` choice. This reduces the total number of frames, but assures that one frame will be created with the marked choice. By marking three choices of the **find** specification as `[miscellaneous]`, the total number of test frames drops to 40. The final, completely marked specification is shown in Figure 3-5.

The decision to use a `[miscellaneous]` marking is a judgement by the tester that the marked choice can be adequately tested with only one test case. It represents an attempt to balance the desire for complete testing of all combinations against the pragmatic limits of time and personnel that usually prevail in a software development project.

Step 6: Transform into test scripts. Figure 3-6 shows one test frame generated from the final specification for **find**, together with the operating system command to establish the file environment, the instance of **find** that performs the test, and a

```
# Test Specification for the FIND command
# Modified: property lists and selector expressions added
# Modified: error and miscellaneous annotations added

Parameters:
    Search string size:
        empty                                   [property Empty]
        single character                        [property NonEmpty]
        many character                          [property NonEmpty]
        longer than any line in the file        [error]

    Quoting:
        string is quoted                [property Quoted]
        string is not quoted            [if NonEmpty]
        string is improperly quoted     [error]

    Embedded blanks:
        no embedded blank               [if NonEmpty]
        one embedded blank              [if NonEmpty and Quoted]
        several embedded blanks         [if NonEmpty and Quoted]

    Embedded quotes:
        no embedded quotes              [if NonEmpty]
        one embedded quote              [if NonEmpty]
        several embedded quotes         [if NonEmpty] [miscellaneous]

    File name:
        good file name
        no file with this name          [error]
        omitted                         [error]


Environments:
    Number of occurrences of pattern in file:
        none                        [if NonEmpty] [miscellaneous]
        exactly one                 [if NonEmpty] [property Match]
        more than one               [if NonEmpty] [property Match]

    Number of pattern occurrences on line containing pattern:
        one                         [if Match]
        more than one               [if Match] [miscellaneous]
```

Figure 3-5: Final test specification for **find**

description of the test's expected output.

Note that the test frame contains two numbers. The "Test Case" number sequentially identifies the test. The "Key" number refers to the choices made from each of the frame's categories (choice 3 from the first category, choice 1 from the

second, and so forth). Once the categories and choices in a test specification are fixed, the key number for a given test frame stays the same even if the property lists and selector expressions change.

Test Frame:

```
Test Case 28:  (Key = 3.1.3.2.1.2.1.)
Search string size : many character
Quoting : string is quoted
Embedded blanks : several embedded blanks
Embedded quotes : one embedded quote
File name : good file name
Number of occurrences of pattern in file : exactly one
Number of pattern occurrences on line containing pattern : one
```

Commands to set up the test:

```
copy /testing/sources/case_28 testfile
```

find command to perform the test:

```
find "has "" one quote" testfile
```

Instructions for checking the test:

```
The following line should be displayed:
   This line has " one quote on it
```

Figure 3-6: A test frame and test case

In the example shown in Figure 3-6, the file **case_28** has been created earlier and stored in another directory. It is copied into a testing directory to set up the environment for the test. The output of **find** is either a display of matching lines or a message that no matches were found. Test result checking could be done either manually or by directing the command's output to a log file which is later compared to a prepared output file. In cases where the tested function modifies the system state, additional commands may be necessary to verify that it performed correctly.

When the actual test scripts are created, inconsistencies, redundancies, and ambiguities in the test specifications may become apparent. A combination of parameter values and environment conditions that looked right in the specification may actually be impossible to test. One test case may be identical to another. Such situations require that the tester modify the original test specification, usually adding property lists and selector expressions, to correct the problem.

4. Experience with TSL

The TSL is now being used for the design of functional tests for the version and configuration management (VCM) component of the interactive programming support environment mentioned in Section 1. The entire system has been designed using object-oriented techniques, and is implemented in Ada.

There have been three interim releases of the VCM component. For the first release, a simple form of cause-effect strategy was used to generate tests. The tester prepared a list of all parameters and environment causes affecting the operation of a function, and another list of all the function's effects. A tool was constructed that formed pairs and triples of the causes. The tester then associated with each of these combinations its expected output effects, and created test cases by choosing specific input and environment values for each combination.

Although this method provided some automated help, it did not give enough control over the way that different causes could be combined. Further, it provided almost no help in systematically categorizing the influences on a function. A more systematic method was essential to handle the large number of system commands and options for each command in the VCM component. The TSL method that succeeded the original approach makes it possible both to define thorough tests and to assess the functional coverage of test scripts.

The preliminary tool used for Release 1 was the basis for the design of the TSL, which has been used to create test scripts for the second and third releases. The second release consisted of approximately 35,000 Ada source statements (counted by semicolons), making up over 2200 Ada procedures in 131 packages. Function testing for this release consisted of testing 91 "high-level" procedures, each roughly equivalent to one user-level command.

The VCM design and implementation team consists of five members. When functional testing was started, one team member became primarily responsible for testing. This test engineer used the written specification documents, Ada package specifications, and his own experiential knowledge of the system to write the test specifications. One test specification was written for each of the 91 high-level procedures. All of the test specifications were written in approximately three weeks.

The four other team members were then given a short one-page overview of the TSL format. These team members individually reviewed the test specifications, noting errors and inconsistencies and suggesting changes. Judging by the quality of the reviews, they found the TSL format to be relatively easy to understand.

Most of the review comments simplified the test specifications. Many environment conditions were eliminated as unnecessary and others were marked

`[miscellaneous]`. In the few instances where functionality had changed since the last revisions of the program specifications, the entire test specification had to be rewritten (and the program specification was quickly revised).

The TSL generator was run on each of the test specifications to produce a set of test frames for each procedure. The entire system required 1022 test cases.

Writing the actual test scripts was the most time-consuming part of the process. Each procedure's script consisted of some setup commands, the test cases themselves, and cleanup commands. Common data structures were used as appropriate. Although it would have cut down on the total number of commands, it was decided not to use one test case to set the environment for a following test case (e.g. test case A creates a file, test case B reads the file). A script with such dependencies is difficult to use since failure of any part of the script renders the remaining tests useless. Wherever possible, the test cases included commands to check the results of the test. Script writing was an iterative process. Some implausible test frames only became apparent when they had to be translated into scripts.

Once the test cases were written, it took approximately two weeks to run all of the tests. Many times there were "bugs" in either the test scripts or the original test specifications themselves, despite the fact that the test specifications had been reviewed prior to the time the test cases were generated and written. This required modifying the scripts, and in a few cases, the test specifications themselves. Thirty-nine *bona fide* bugs in the software were found and corrected.

The complete suite of test scripts has been saved as a regression test base for future releases. All of the test specifications and scripts have been version controlled. To facilitate checking the regression tests, a transcript of the last complete run of the test scripts was saved so that it can be compared mechanically against a transcript of a run of the same scripts against a future release.

5. Conclusions

The category-partition method provides the tester with a systematic method for decomposing a functional specification into test specifications for individual functions. The test specification language is a concise, easily modified representation of tests that can be understood by programmers, testers, and managers. A particular benefit is that the tester can control the size of a product's test suite, not by arbitrarily stopping when a given number of tests have been written, but by specifying the interaction of parameters and environments that determine the tests.

TSL and the category-partition method are a promising foundation for further automation of specification-based functional testing, a task that is generally seen as tedious but necessary for the implementation of high-quality software.

References

1. Adrion, W.R., Branstad, M.A., and J.C. Cherniavsky. "Validation, Verification, and Testing of Computer Software". *ACM Computing Surveys 14*, 2 (June 1982), 159-192.

2. Berzins, V. and M. Gray. "Analysis and Design in MSG.84: Formalizing Functional Specifications". *IEEE Transactions on Software Engineering SE-11*, 8 (August 1985), 657-670.

3. Bjorner, Dines and C.B. Jones. *Formal Specification and Software Development.* Prentice-Hall International, Englewood Cliffs, NJ, 1982.

4. Goodenough, J.B. and S.L. Gerhart. "Toward a Theory of Test Data Selection". *IEEE Transactions on Software Engineering SE-2*, 2 (June 1975), 156-173.

5. Howden, W.H. A Survey of Dynamic Analysis Methods. In *Tutorial: Program Testing and Validation Techniques*, Miller, E.F. and W.H. Howden, Eds., IEEE, 1981.

6. Weyuker, E.J. and T.J. Ostrand. "Theories of Program Testing and the Application of Revealing Subdomains". *IEEE Transactions on Software Engineering SE-6*, 3 (May 1980), 236-246.

BANG METRICS: AN ONGOING EXPERIMENT AT HP

Research and Development Engineer
Software Engineering Laboratory
Software Engineering Operation
Corporate Engineering
Hewlett-Packard Company
Robert W. Dea

ABSTRACT

Bang metrics, an early indicator of system size and complexity, will be discussed. The paper contains a short introduction to Bang metrics followed by a description of the tools that were created to help calculate Bang. Then, some of the interesting discoveries that have come about as a by-product of the Bang tools will be discussed.

After studying the work of Tom DeMarco [1], the Software Engineering Operation (SEO) at HP felt that he was on the right track with his work on specification metrics (Bang). With the arrival of automated CASE tools to help create Data Flow Diagrams, we decided to create tools that could analyze the databases of these CASE products and produce the Bang calculations.

BIOGRAPHICAL SKETCH

A native of California, Robert attended the University of California at Berkeley. He received his BS degree in Electrical Engineering and Computer Science in 1973. He has contributed to the development of graphics software and design tools for Hewlett-Packard computers. He is currently involved with designing a Integrated Project Support Environment (IPSE) at HP. He worked in the aerospace industry before joining HP in 1979. Robert lives in Fremont, California, is married, and is interested in meditation and martial arts. He also enjoys fishing, photography, and personal computers. He can be reached at:

Robert W. Dea
Hewlett-Packard Company
3500 Deer Creek Road Bldg. 26U
Palo Alto, California 94304
(415) 857-7095

INTRODUCTION

The Software Engineering Operation (SEO) / Software Engineering Lab (SEL) plays an important role as a catalyst for action among HP divisions and as a vehicle for exchange of information. To accomplish this, SEO/SEL develops and/or acquires software productivity engineering tools and methodologies which are used to control the software development process. SEO/SEL then releases the tools to HP divisions and operations, and provides supporting and/or consulting services as needed.

SEO/SEL saw promise in DeMarco's specification metrics and promoted its use within HP. HP organizations, at that time, had to calculate Bang manually. This was a very time consuming and tedious task. Over the last few years, a number of third-party structured analysis tools have reached the market. SEO/SEL evaluated a number of these tools and recommended a few for use within HP. To help in the collection of Bang metrics, SEO/SEL developed Bang calculation tools that work with each of the third-party tools.

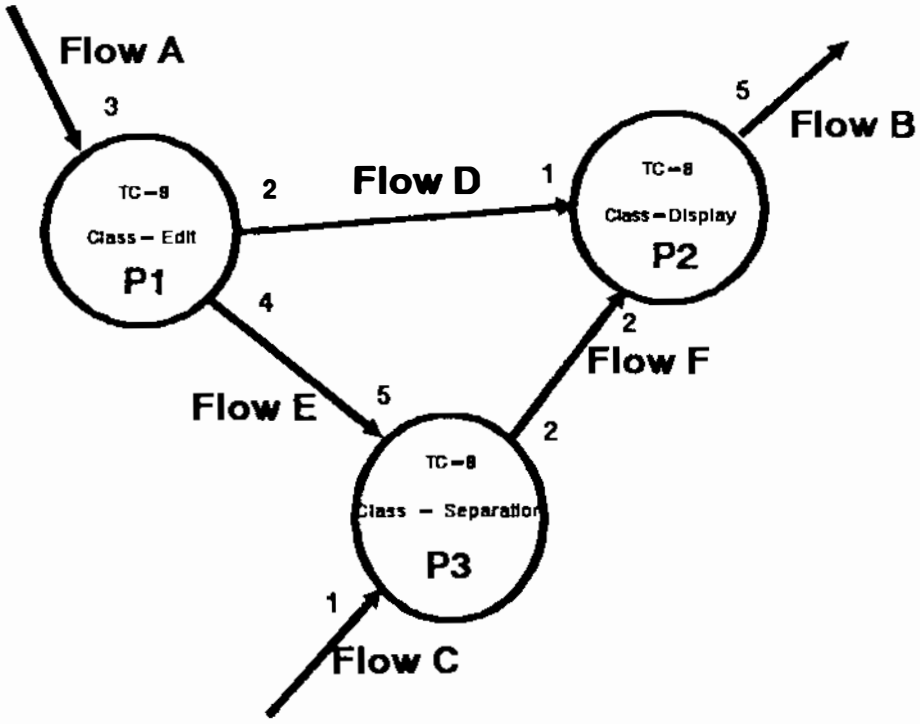
DEMARCO'S BANG METRICS

Tom DeMarco [1], in his book, *Controlling Software Projects*, proposes a model for estimating effort on software projects. DeMarco suggests that the specification model can give an accurate indication of system size.

The basic concept behind DeMarco's [1] algorithm for calculating Bang for function strong systems are:

1. Find all the primitive processes in the specification.
2. Calculate the Token Count of each primitive by summing the number of data elements consumed or produced by the primitive process.
3. Adjust this Token Count for data weight using DeMarco's [1] "Data Weighting for Size Correction of Functional Primitives" table.
4. Assign a Class type for the primitive process and look up its Class Weight in the "Complexity Weighting Factors by Class of Function" table.
5. Multiply the adjusted Token Count by the Class Weight.
6. To get a total Bang value sum up the above for all the primitive processes.

For an example, using the above algorithm, see Figure 1. The above mentioned tables and the detailed description of the Bang algorithm are in Chapter 9 of DeMarco's [1] book, *Controlling Software Projects.*



P1 Token Count = 9, Adjusted Token Count = 14.3
P2 Token Count = 8, Adjusted Token Count = 12.0
P3 Token Count = 8, Adjusted Token Count = 12.0

P1 Class = Edit Class Weight = 0.6
P2 Class = Display Class Weight = 1.8
P3 Class = Separation Class Weight = 0.6

Total Bang = P1(ATC)*P1(CW) + P2(ATC)*P2(CW) + P3(ATC)*P3(CW)

Total Bang = 14.3 * 0.6 + 12.0 * 1.8 + 12.0 * 0.6 = 37.38

Figure 1.

AUTOMATED BANG CALCULATORS

SEO/SEL developed stand-alone Bang calculators for use with existing structured analysis tools. The Bang metric programs traverse the structured analysis tools' database and detect the primitive processes. They look at the dataflows that are connected to these processes and calculate the token counts by reading the data dictionary and summing up each data item in the definition. The Bang programs assign the default value of one for the Class Weight of each primitive process.

Optionally, the user can specify a token count exception file and/or a class weight exception file. Exception files allow the user to refine the Bang calculation by specifying more accurate token counts for individual tokens or by specifying class weights for primitive processes. At the end of this paper is an example specification, data dictionary and Bang output. The Bang output was calculated using one of the automated Bang calculators described above.

Once the Bang metric is calculated, what good is it? Like any metric, the Bang metric is of no use by itself. DeMarco [1] believes that Bang, collected over a number of software projects, can be used for early prediction of effort. Bob Grady (Corporate Engineering) suggests that it is equally important to relate Bang values to parts of a design or code in order to measure complexity. The goal is to identify and correct potential support problems early in the development process.

SEO/SEL will be working with the HP Software Metrics Council to incorporate Bang into their metrics program. Hopefully in the future, HP divisions will access normalized Bang data and will use this information to predict effort and complexity in future HP software projects.

LEVELED BANG

Our goal for creating the Bang programs was to make it easy to calculate Bang so that it can be used as a predictor of effort in the future. We expected long term benefits from collecting Bang and did not expect any short term pay off.

During the development of the Bang programs, we decided to optionally produce a waterfall report containing Relative Bang at each intermediate level from the specification diagrams. This Leveled Bang report can also be formatted so it can be used by a number of outlining programs available on HP's Vectra PC for additional analysis. Immediate benefits can be obtained using an outlining program with this Leveled Bang information.

An outlining program allows the user to look at as much detail of a multi-leveled outline as one wishes. The outlining program starts out by showing the upper most level of the Leveled Bang report. By using the arrow keys to select a process, one can press the "+" key to view lower levels or use the "-" key to hide lower levels.

In Figure 2, we see the outer most level of a Leveled Bang report, as displayed on the screen by an outlining program. We can visually see the Context Diagram level. Starting at the indentation, we have the Level 0 diagram. On the left portion of the display we see "+" and "-" characters. A "+" character specifies that the process can be expanded into additional levels. A "-" character marks a primitive process, one that can not be further expanded.

```
+  1004.14  Calculator Editor
   -   109.83  Calculator Commands (1)
   +   139.58  Tools (2)
   +    16.75  View (3)
   +    12.00  Setting (4)
   +   330.09  Perform Calculation (5)
   +    81.21  Mouse Processor (6)
   +   186.86  File (7)
   +    47.81  System (8)
   +    43.28  Scroll (9)
   +    36.71  Paint (10)
```

Figure 2.

By moving the cursor to Perform Calculation, process (5) in Figure 2, and pressing the "+" key, the process will expand to the following in Figure 3.

```
+  1004.14  Calculator Editor
   -   109.83  Calculator Commands (1)
   +   139.58  Tools (2)
   +    16.75  View (3)
   +    12.00  Setting (4)
   +   330.09  Perform Calculation (5)
       -    34.74  Edit Pulldown Menu (5.1)
       -    18.58  Calculate (5.2)
       +    47.36  Numeric Input (5.3)
       +    29.20  Clear (5.4)
       +    35.12  Subtraction (5.5)
       +    32.36  Addition (5.6)
       +    64.04  Multiply (5.7)
       -     2.38  Redo (5.8)
       +    66.29  Divide (5.9)
   +    81.21  Mouse Processor (6)
   +   186.86  File (7)
   +    47.81  System (8)
   +    43.28  Scroll (9)
   +    36.71  Paint (10)
```

Figure 3.

We now have the ability to look at various levels of detail in the Leveled Bang report. What can this information be used for? One use is to help identify possible flaws in the specification. In Figure 2, we noticed that Perform Calculation, process (5) has a Bang number of 330.9, that is approximately 1/3 of the Total Bang for the project. By expanding into the process as in Figure 3, we discover that it breaks down into many smaller processes of reasonable Bang numbers. If you look at Calculator Commands, process (1), the Bang number is 109.83 which is 10% of the Total Bang yet it is a primitive process. This process stands out as a possible flaw in the specification.

A number of factors can cause an exceptionally high Bang number:

1. it may represent reused code
2. it could mean that the specification for the process is incomplete
3. it may represent a large switch statement
4. it could mean the process is very complex

The first three reasons may not cause problems. But if the process is very complex, it can mean that if any defects are in the finished project, this process is a likely candidate. Detection of defects in the specification gives one the opportunity to correct them early in the project.

Project managers can use this information to help allocate engineering resources. One HP project manager had four engineers working for him. The manager noticed that one of the engineers was taking a lot more time to finish his part in the project. It was later discovered that this engineer had five times the amount of work of the other engineers.

This unequal allocation of labor is very common when a method of evaluating relative effort within a project is not available. If the Relative Bang does imply relative effort then the manager could have allocated 1/4 Total Bang to each of the engineers.

In the future, if this method of allocating engineering resources is practical, we can develop automated ways of moving this information into future project management systems. When we are able to use Bang to get accurate predictions for effort, we can do away with engineers pulling estimates out of a hat, or managers having magic numbers for completion dates.

With Leveled Bang we also have the opportunity to perform "What if?" evaluations. Suppose we predicted that a project would take two years of effort to finish. But the window of opportunity for the product to be successful is one and a half years. With information from Leveled Bang, portions of the specification can be eliminated so that it could be completed in one and a half years and still maintain most of its functionality.

Leveled Bang can be used by engineers to justify estimates to managers. When additional features to a product are requested, the engineers and/or managers now have a tool to estimate the impact upon the schedule of the addition of these features.

UNANSWERED QUESTIONS

Now that we have automated tools to collect Bang, we are discovering that we need answers to a number of questions before we can fully utilize Bang as a predictor.

This is the current list of unanswered questions:

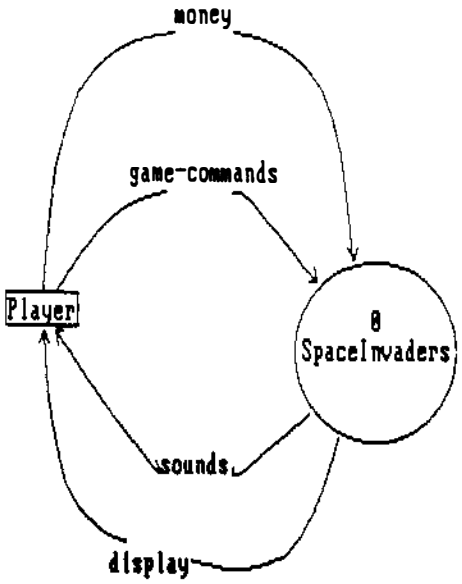
- How does Bang relate to engineering effort?
- How does Bang relate to code size?
- How does Bang relate to development environments?
- How do we normalize Bang across the company?
- What are reasonable Bang numbers for primitive processes?
- Is Relative Bang linear?
- How do we know data flow diagrams are complete before we compute Bang?
- Can we predict final Bang if we only finish two or three levels?
- If we complete one path to completion, can we predict final Bang?
- Is there a correlation between defects and large Bang values?
- Do large Bang values cause designers to redesign?
- What else can we get from Bang?

Now that automated tools for collecting Bang are available, HP must now face the job of answering these questions.

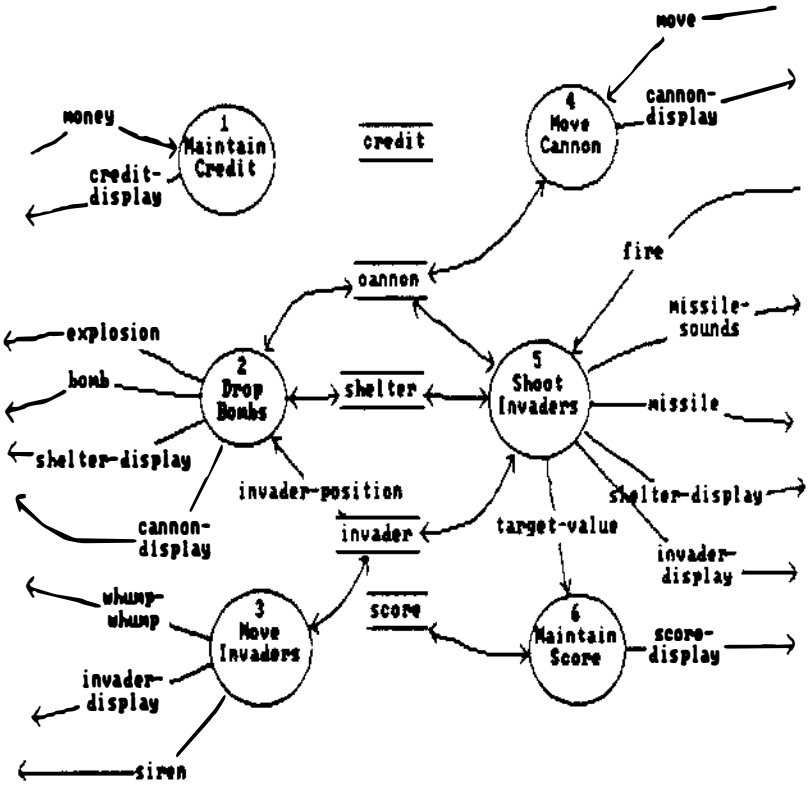
REFERENCES

1. DeMarco, Tom., Controlling Software Projects. New York: Yourdon Press, 1982.

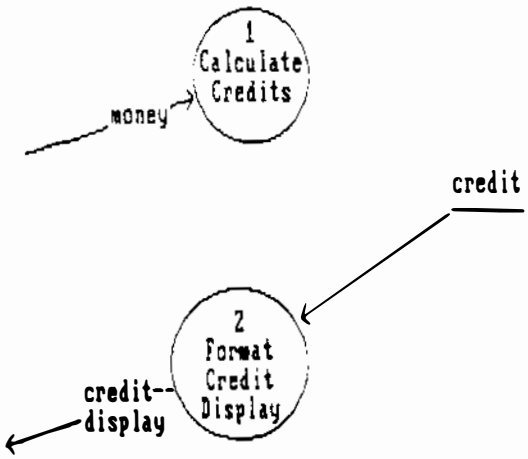
SPACE INVADERS DATAFLOW DIAGRAMS:



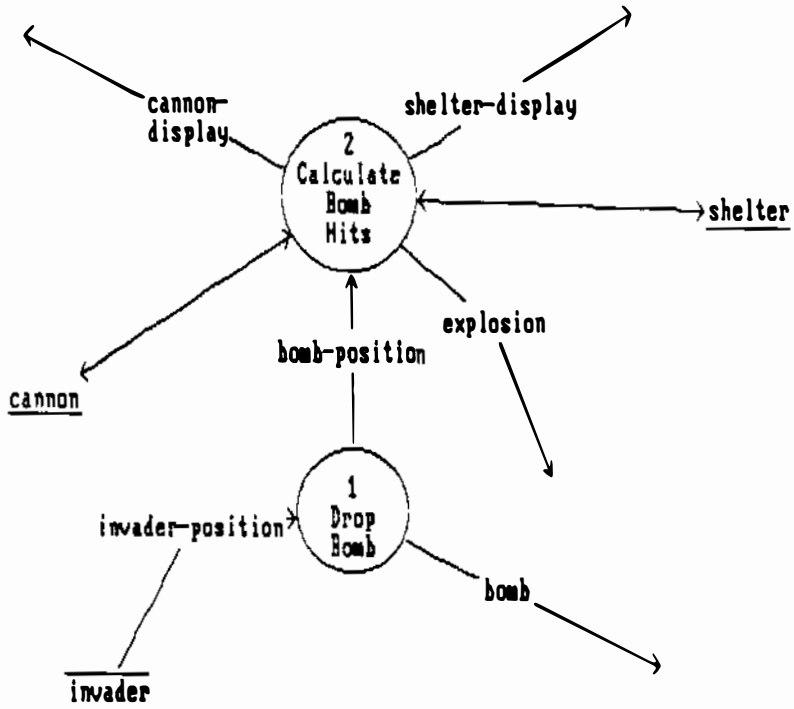
DFD -1 : Context Diagram



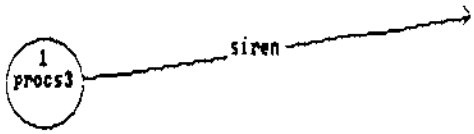
DFD 0 : SpaceInvaders



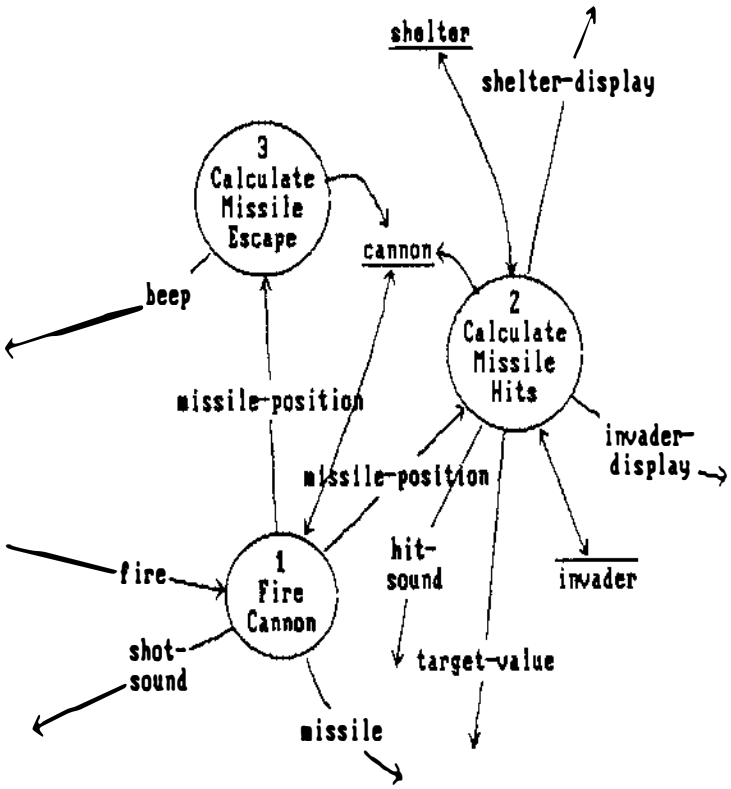
DFD 1 : Maintain Credit



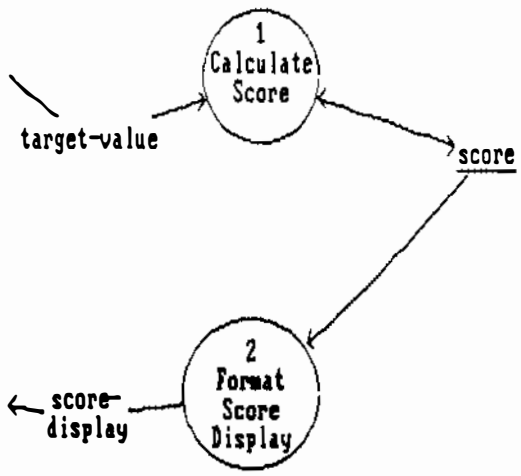
DFD 2 : Drop Bombs



DFD 3 : Move Invaders



DFD 5 : Shoot Invaders



DFD 6 : Maintain Score

SPACE INVADERS DATA DICTIONARY:

beep = *Sound when cannon shot goes off*

bomb = *Portion of screen where bombs drop*

bomb-position = *Position of bomb on screen*

cannon = cannon-position + cannon-image + missile-fired

cannon-display = *Portion of screen where the cannons are moved*

credit = *Integer number of games player can play*

credit-display = *Display of the number of games player can play *

display = credit-display + invader-display + cannon-display + score-display + shelter-display + missile + bomb

explosion = *Sound when a bomb strikes its target *

fire = *Command from fire button to fire cannon*

game-commands = {[move | fire]}

hit-sound = *Sound when a cannon shot strikes its target*

invader = {invader-position + invader-type + invader-image}

invader-display = *Portion of screen where invaders march*

invader-position = *Position of invader on display*

missile = cannon-position *Portion of screen where missile fires*

missile-position = *Integer position of missile on screen*

missile-sounds = shot-sound + hit-sound + beep

money = 1{[nickel | dime | quarter]}

move = *Command from joystick to move cannon*

score = *Integer count of hit-points by player*

score-display = *Portion of screen where score is displayed*

shelter = {shelter-position + shelter-image}

shelter-display = *Portion of screen where shelters are*

shot-sound = *Sound of cannon when fired*

siren = *Sound when invader wins*

sounds = [missile-sounds | explosion | whump-whump | siren]

target-value = *Integer value of invader type*

whump-whump = *Sound of an advancing invader*

BANG METRIC OUTPUT:

Token Count	Data Dictionary Name
1	beep
1	bomb
1	bomb-position
3	cannon
1	cannon-display
1	credit
1	credit-display
7	display
1	explosion
1	fire
2	game-commands
1	hit-sound
3	invader
1	invader-display
1	invader-position
1	missile
1	missile-position
3	missile-sounds
3	money
1	move
1	score
1	score-display
2	shelter
1	shelter-display
1	shot-sound
1	siren
4	sounds
1	target-value
1	whump-whump

Bang	Bubble #	Label
110.14	0	SpaceInvaders
3.38	1	Maintain Credit
2.38 *	1.1	Calculate Credits
1.00 *	1.2	Format Credit Display
29.03	2	Drop Bombs
2.38 *	2.1	Drop Bomb
26.65 *	2.2	Calculate Bomb Hits

0.00 3 ..... Move Invaders

0.00 * 3.1 ..... procs3

12.00 * 4 ..... Move Cannon

62.36 5 ..... Shoot Invaders

19.03 * 5.1 ..... Fire Cannon

37.53 * 5.2 ..... Calculate Missile Hits

5.80 * 5.3 ..... Calculate Missile Escape

3.38 6 ..... Maintain Score

2.38 * 6.1 ..... Calculate Score

1.00 * 6.2 ..... Format Score Display

Notes

Notes

1987 Pacific Northwest Software Quality Conference

Proceedings

ORDER FORM

Name _____

Affiliation _____

Address _____

City, State _____ Zip _____

Telephone _____

For each copy of the *Proceedings*, please enclose payment of $25. Checks should be payable to PNSQC, and mailed with the form to:

PNSQC
c/o Lawrence & Craig, Inc.
P.O. Box 40244
Portland OR 97204