TENTH ANNUAL PACIFIC NORTHWEST

SOFTWARE QUALITY CONFERENCE

October 19 - 21, 1992

Oregon Convention Center
Portland, Oregon

TABLE OF CONTENTS

MANAGEMENT TRACK - DAY TWO

TECHNICAL TRACK - DAY ONE

TECHNICAL TRACK - DAY TWO

PREFACE

Elicia M. Harrell

Welcome to the Tenth Annual Pacific Northwest Software Quality Conference. It is rewarding to know that for our first decade, we have been able to provide successful Software Quality Conferences, which provided a forum and environment for software professionals to meet, share information, learn new ideas, and acquire new skills.

Through these last ten years, we've watched the software industry change the focus from getting products out to providing good quality products on time. The nation has placed more emphasis on quality, examples being the Malcolm Baldrige Award and such institutions as SEI, which strive to help companies become better at predicting and facilitating quality within their own products and organizations. We've watched the Conference grow from a day of meetings held in conference rooms at the Hewlett-Packard facility in Corvallis, Oregon, to a three-day conference at the Oregon Convention Center with two days of technical presentations and one day of workshops. And this growth has taken place while the computer industry itself has experienced quite a shakeout in terms of focus, job availability, and future direction.

Our 1992 keynote speaker is Dr. Bill Hetzel, a well-known software-testing consultant and principal of Software Quality Engineering. His work has focused on better software engineering and quality assurance, and one of his publications is entitled "The Complete Guide to Software Testing." His keynote address is entitled "Current Perspectives and Directions in Software Testing," which will take a broad look at the discipline of software testing.

We are pleased to publish these Proceedings, which contain the papers presented during the technical program. The papers represent current thinking and practice from the United States and Canada. Twenty-three papers were selected from the abstracts received from our Call for Papers. An additional six speakers representing experts in the industry have been invited to share their experience and expertise during our technical sessions. An Integrated CASE Tools presentation and debate with representatives from four companies and a Testing Tutorial have also been added to our program this year.

I would like to thank Hilly Alexander and Mark Johnson, the Program Committee Co-Chairs, for putting in the hard work it takes to pull the program together. Their work is the basis for this Conference. Many thanks also to the members of the Program Committee for their time, energy, and intelligence in refereeing the abstracts and papers. The Program Committee is the foundation of the process for putting together the Conference.

I would like to thank the remaining members of the full committees, whose names are listed in the next section, who contributed their ideas, effort, and attendance to many meetings to make this Conference a success.

Finally, I want to express special thanks to Terri Moore at Pacific Agenda for being able to handle the organizational and administrative tasks while, at the same time, keeping the committees on track and moving forward.

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Elicia Harrell - President/Chair
Intel Corporation

Steve Shellans - Vice President
Martiny & Shellans, Inc.

Debra Lee - Secretary
Intel Corporation

Lowell Billings - Treasurer
Infotec Development

Hilly Alexander - Program Co-Chair
ADP

Mark Johnson - Program Co-Chair
Mentor Graphics Corporation

Sue Bartlett - Workshops
Tektronix, Inc.

James Mater - Exhibits
Technical Solutions, Inc.

G.W. Hicks - Publicity
TSSI, Inc.

Ray Lischner - Software Excellence Award
Mentor Graphics Corporation

CONFERENCE PLANNING COMMITTEE

Chuck Adams
Tektronix, Inc.

Sandhi Bhide
Tektronix, Inc.

Ann Bynum
Intel Corporation

Alan Chumura

Rick Clements
Tektronix, Inc.

Curtis Cook
Oregon State University

David Crockett
Portland State University

Terri Crowley
II Morrow

Margie Davis
ADP

Dave Dickmann
Hewlett-Packard

Cynthia Gens
Solus Systems, Inc.

Michael Green

Dick Hamlet
Portland State University

Gary Hanson
Kentrox Industries

Bill Harris
Hewlett-Packard

Warren Harrison
Portland State University

Karen Herrold
Intel Corporation

Greg Hoffman
Tektronix, Inc.

Connie Ishida
Mentor Graphics Corporation

Rajesh Jhunjhunwala
Technical Solutions, Inc.

Bill Junk
University of Idaho

Karen King
Credence

Ken Maddox
Oregon Software Association

Peter Martin
Apple Computer

Howard Mercier
Intersolv

Hausi Mueller
University of Victoria

Shannon Nelson
Intel Corporation

Dave Patterson
Mentor Graphics Corporation

Misty Pesek
Serviologic

Mark Powell
Phoenix Management, Inc.

Ian Savage
CFI Barker Services Group

Eric Schnellman
Boeing Commercial Aircraft

Katherine Stevens
ADP

George Tice

Jay Van Sant
Management Resources

Donald H. White

Nancy Winston
Mentor Graphics Corporation

Barbara Zanzig
NeoPath

Barbara Zimmer
Hewlett-Packard

PRESENTERS

Tamara Baughman
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070-7777

Dr. Linda Beach
NYMA, Inc.
7501 Greenway Center Drive
Greenbelt, MD 20770

Frank Calliss
Dept. of Computer Science
Arizona State University
Tempe, AZ 85287-5406

Bert de Boer KCS/KS
NV KEMA
PO Box 9035 6800 ET ARNHEM
The Netherlands

Mary Jean Harrold
Dept. of Computer Science
Clemson University
Clemson, SC 29634-1906

Joel Henry
Dept. of Math. and Comp. Science
Dickinson College
Carlisle, PA 17013

Bill Hetzel
Software Quality Engineering
3000-2 Hartley Road
Jacksonville, FL 32257

Dr. William Howden
University of California at San Diego
La Jolla, CA 92093

Rajesh Jhunjhunwala
Technical Solutions, Inc.
16199 NW Joscelyn Street
Beaverton, OR 97006

John Lakos
Mentor Graphics Corporation
15 Independence Boulevard
Warren, NJ 07059

Wayne Lim
Hewlett-Packard Company
Corporate Engineering
1801 Page Mill Road, 18DG
Palo Alto, CA 94303

Ray Lischner
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070-7777

Brian Marick
Testing Foundations
809 Balboa
Champaign, IL 61820

Wayne McDaniel
Intel Corporation
C01-01
15201 NW Greenbrier Parkway
Beaverton, OR 97006

Thomas Ostrand
Siemens Corporate Research, Inc.
755 College Road East
Princeton, NJ 08540

Eugene Spafford
Dept. of Computer Science
Purdue University
West Lafayette, IN 47907-1398

Shari Lawrence Pfleeger
MITRE Corporation
7525 Colshire Drive
McLean, VA 22102

Rukmani Tekchandani
Intel Corporation
5200 NE Elam Young Parkway, JF1-67
Hillsboro, OR 97124

Mary Sakry
The Process Group
PO Box 515531
Dallas, TX 75251-5531

Dr. Jeffrey Voas
Reliable Software Technology Corp.
1001 N. Highland Street
Arlington, VA 22201

Jean Scholtz
Portland State University
41 N. Atlantic, Apt. 2
Cocoa Beach, FL 32931

Steve Whitchurch
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070-7777

Barbara Siefken
Tektronix, Inc.
PO Box 500, M/S 39-750
Beaverton, OR 97077

Scott Whitmire
Boeing Computer Services
PO Box 24346, MS 6C-MA
Seattle, WA 98124-0346

EXHIBITORS

Bill Sundermeier
Cadre Technologies Inc.
19545 NW Von Neumann Drive
Beaverton, OR 97006
503/690-1318

Pacific Northwest
Software Quality Conference
PO Box 970
Beaverton, OR 97075
503/223-8633

Margaret Quigley
Interactive Development Environments
595 Market St., 10th Floor
San Francisco, CA 94105
415/543-0900

Shawn Wall
Powell's Technical Books
33 NW Park Avenue
Portland, OR 97209
503/228-3906

Becky Johnson
KnowledgeWare, Inc.
3340 Peachtree Rd. NE, #1100
Atlanta, GA 30326-1050
404/231-3510

Sue Condon
PSDI
777 108th Ave. NE, Suite 600
Bellevue, WA 98004-5195
206/646-4818

Joan Brown
McCabe & Associates, Inc.
5501 Twin Knolls Rd., #111
Columbia, MD 21045
301/596-3080

Aaron Omid
Scopus Technology, Inc.
1900 Powell St., #900
Emeryville, CA 94608
510/428-0500

Steve Shellans
Martiny & Shellans, Inc.
34351 South Ranch Road
Newberg, OR 97132
503/625-6417

Teresa Harrison
SET Laboratories
PO Box 868
Mulino, OR 97042
503/829-7123

George Lai
Mercury Interactive Corporation
3333 Octavius Drive
Santa Clara, CA 95054
408/987-0109

Linda North
Software Quality Engineering
3000-2 Hartley Road
Jacksonville, FL 32257
904/268-8639

Rita Bral
Software Research, Inc.
625 3rd Street
San Francisco, CA 94107-1997
415/957-1441

Craig Thompson
Texas Instruments
3365 Avenida de Loyola
Oceanside, CA 92056
619/721-9209

James Mater
Technical Solutions, Inc.
1485 NW Lancashine Ct.
Beaverton, OR 97006
503/690-7741

Ellen O'Dell
Tiburon Systems, Inc.
1290 Parkmoor Avenue
San Jose, CA 95126
408/371-9400

Current Perspectives and Directions in Software Testing

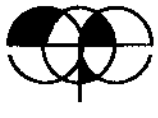
Abstract

This talk takes a broad look at the discipline of software testing—where it has come from, its evolution and development, what the industry practices are today and what we forecast and see for tomorrow.

The talk will raise and explore some of the important open issues in testing including linkage to specifications and other review and verification activities; life cycle methodology implications; the test-then-code philosophy; and how to best manage testing and measure its effectiveness.

Dr. Bill Hetzel

Bill is the current President and CEO of Software Quality Engineering, a leading software quality training and consulting company. He has helped many companies to develop and implement better standards and procedures and has been involved in many software assessments and improvement efforts. He has written several books including the popular Complete Guide to Software Testing. Bill has also written many articles and papers on software testing and engineering practices and is a frequent speaker at conferences and professional meetings. He earned his M.S. degree from Rensselear Polytechnic Institute and Ph.D. in computer science from the University of North Carolina.



Software Quality Engineering
3000-2 Hartley Road • Jacksonville, FL 32257
904-268-8639

AGENDA

1. INTRODUCTION
- What is Testing?
- Evolution—The Changing Ideas About Testing
- A Short Report Card on How We're Doing

2. TESTING AND SPECS—
A Key Technical Issue

3. TESTING AND MANAGEMENT—
A Key Effectiveness Issue

4. SUMMARY

WHAT IS TESTING ANYWAY?

1. **A 1950's View:**

   Testing is what programmers do to find and clean up bugs in their programs.

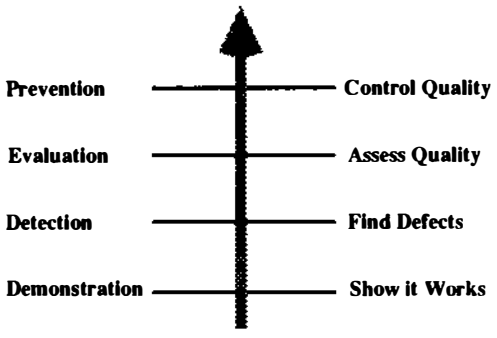
2. **A 1975 View:**

   "Testing is the process of executing a program with the intent to find errors."

3. **A 1990's View:**

   Testing is *planning, designing, building, maintaining* and *executing* tests and test environments (testware engineering).

EVOLUTION OF TEST OBJECTIVES



A SHORT REPORT CARD

Goal	Grade
1. Effective software demonstration	A/B
2. Detection of problems before release	B/C
3. Evaluation of quality risks	C/D
4. Control of risk and prevention of quality problems	E
5. Established technical process	B/C
6. Established management process	D/E

THE HARD REALITY IN MOST ORGANIZATIONS

1. There is Little Test Direction
2. Testing Is Out Of Control
3. The Test Process Is Poorly Defined
4. Substantial Improvement Opportunities Exist

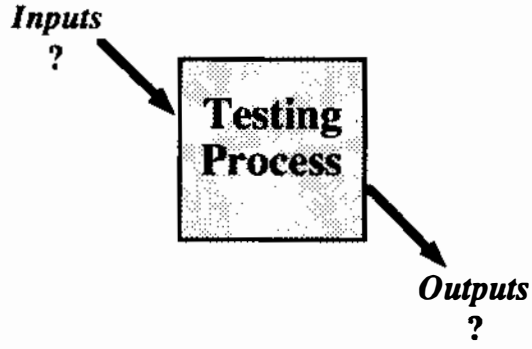
"The current state of testing in many organizations is such that there is not a perceived problem that is subject to solution, but merely a condition that exists."

Anonymous

AGENDA

1. INTRODUCTION
2. TESTING AND SPECS
   - Specifications—Testing Input or Output
   - When to Prepare Test Specs
   - Testware Engineering
   - Preventative Testing
   - What Is Testing Revisited?
3. TESTING AND MANAGEMENT
4. SUMMARY

Understanding the Relationship Between TESTING and SPECIFICATIONS



SPECIFICATIONS—A TESTING INPUT or OUTPUT

Three Views

1. **FIXED INPUT**

   Assume complete specs available as input

   Assume they don't change

2. **CHANGING INPUT**

   Understand that specs will change

   Don't view them as being changed by testing

3. **CHANGING INPUT and OUTPUT**

   Expect specs to change during the test process

   Expect specs to be changed significantly by the test process

TEST SPECIFICATIONS—WHEN TO PREPARE THEM

Three Views

1. **AFTER CODE**
   Bulk of test design after code is complete and "seems" to be working

   No sense trying to test what you haven't built

2. **PARALLEL WITH DESIGN and CODE**
   Bulk of test design done as the coding is done

   May be used to support code review and inspection

   Insures testing issues are raised and considered

3. **BEFORE DESIGN and CODE**
   Test design required to be completed before code starts

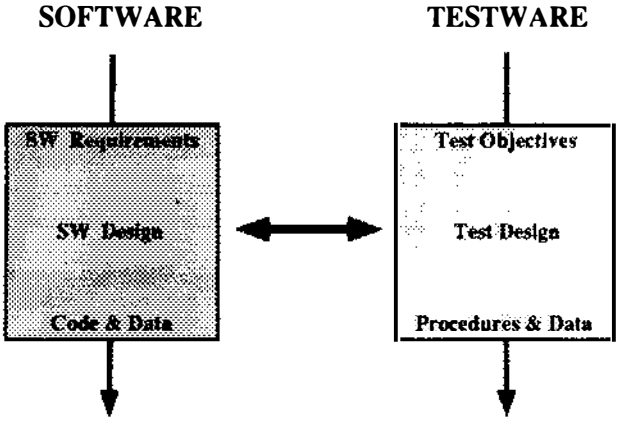
   Test specs are an input to detailed design and coding

   Tests used to support requirements and design inspections

TESTWARE ENGINEERING

TE is concerned with the development and maintenance of software test systems

Managing and engineering reusable testware assets.



PREVENTATIVE TESTING

1. **A NEW (REVOLUTIONARY ??) way of thinking**

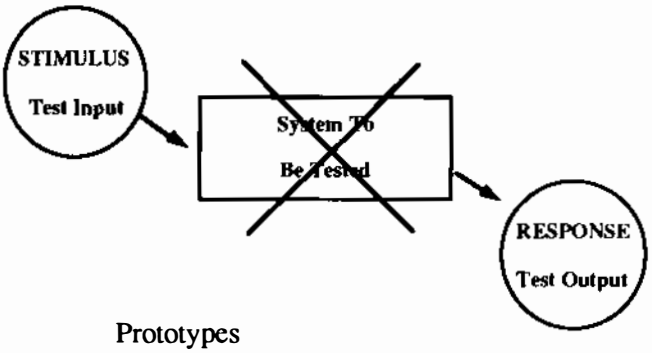
2. **Using tests to INFLUENCE and CONTROL software design and development**

"Preventative testing is built upon the observation that one of the most effective ways of specifying something is to describe (in detail) how you would recognize (test) the something if someone ever gave it to you."

Bill Hetzel

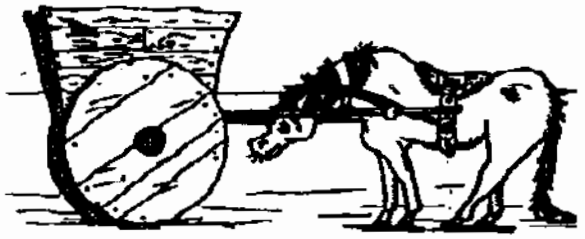
TESTS AS MODELS

Is This the Behavior We Require?

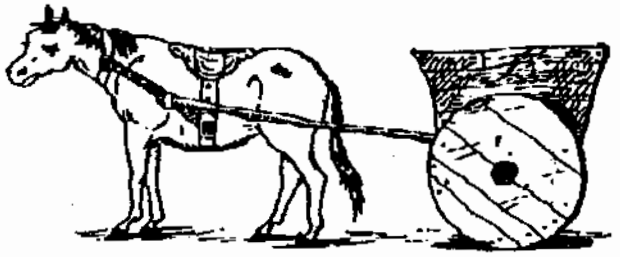


Prototypes
Screen Formats and Report Samples
User Guides
Scenarios
Decision Tables

CODE and TEST



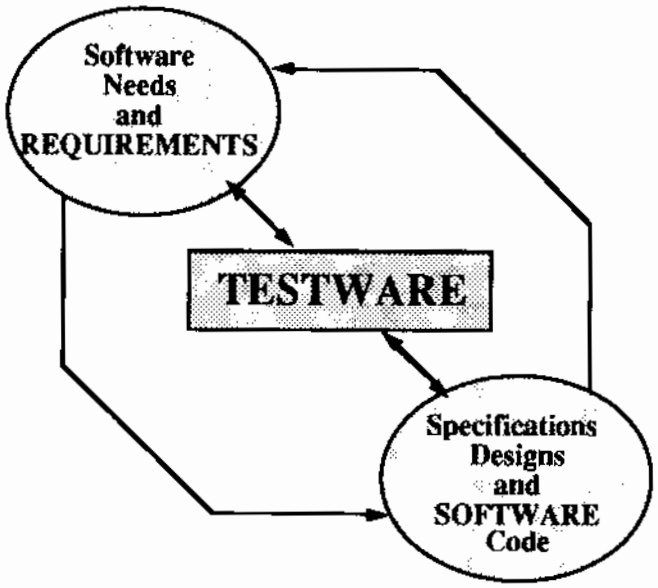
TEST, then CODE



KEY ELEMENTS OF PREVENTATIVE TESTING

1. **Tests are used as requirements models**
2. **Testware design leads software design**
3. **Defects are detected earlier or prevented altogether**
4. **Defects are systematically analyzed**
5. **Testers and developers work together**

MODERN TESTING REVISITED



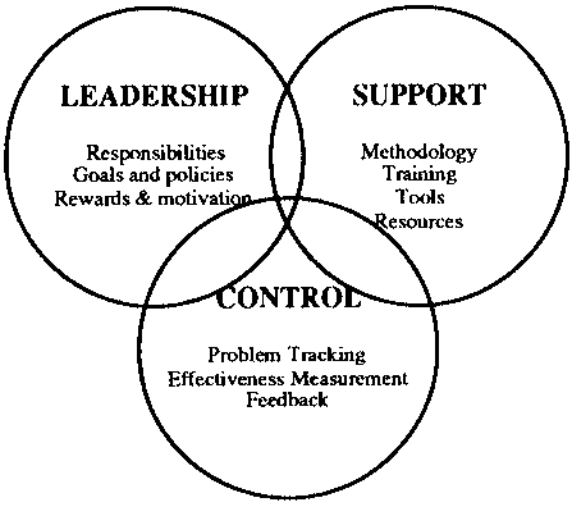
AGENDA

1. INTRODUCTION
2. TESTING AND SPECS
3. TESTING AND MANAGEMENT
   - What Should Management Be Doing?
     - Leadership
       - Setting a Vision • Organizing Effectively
     - Support
       - Tooling and Training
     - Control
       - Measurement • Defect Analysis
   - What the Industry Is Actually Doing
4. SUMMARY

"It is especially during testing that many managers cease to function as managers and become cheerleaders instead. They urge their troops on: 'Work harder, be smarter, go faster.' That may be heartening to the workers, but it just isn't management."

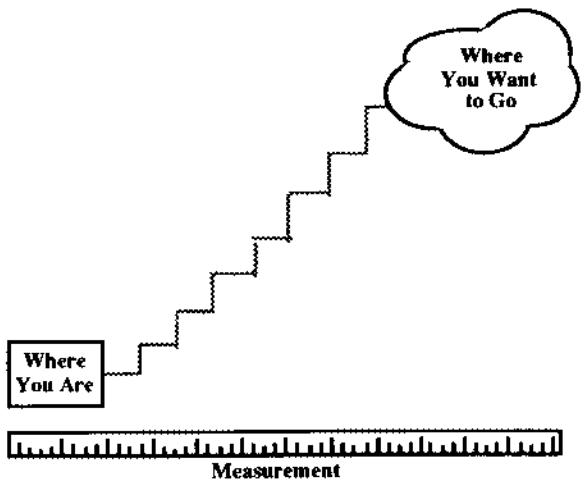
Tom DeMarco
quoting Tom Lister

What Should Mgmt Be Doing?

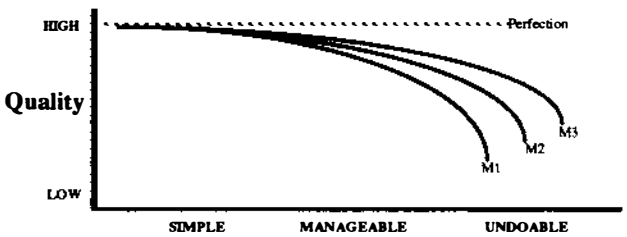


SETTING A VISION

Important to have a direction



MAPPING METHODOLOGY TO THE PROBLEM



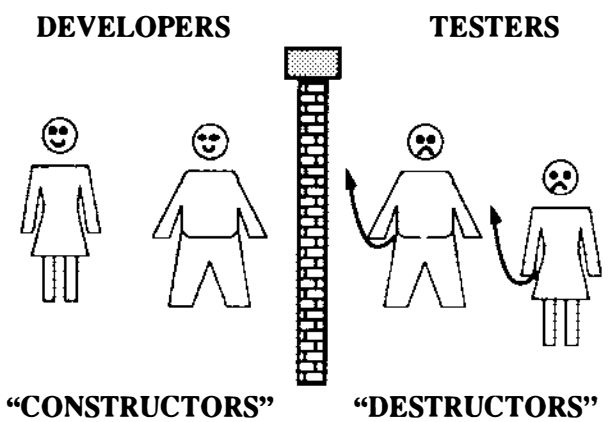
ORGANIZING EFFECTIVELY

Getting Good People Involved

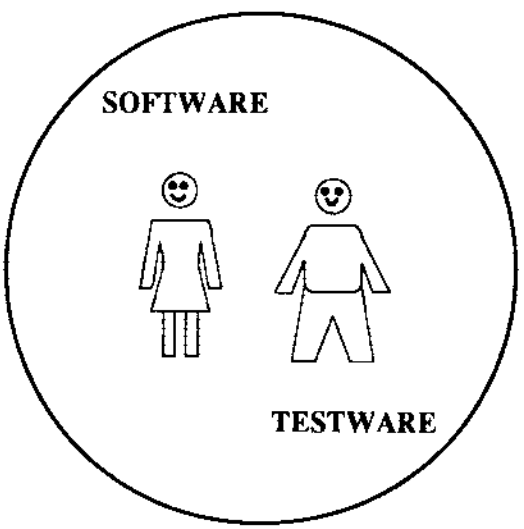
"I've got it, too, Omar ... a strange feeling like we've just been going in circles."

WHAT DOESN'T WORK

(For Long)



WHAT DOES WORK



SUPPORT

Tests must get automated

- Tool Selection
- Test Environments
- Infrastructure Investment

Training Must Be Supported

- Managers
- Software Practitioners
- Users

Resources Provided to Do the Job

CONTROL

You must establish a baseline

- Process
- Costs
- Results

Effectiveness needs to be measured

- Test Economics
- Defect Analysis

GOALS vs MEASURES

TESTING GOAL	*APPLICABLE MEASUREMENTS*
DEMONSTRATION To make sure the software satisfies the specifications	Test successes and failures
DESTRUCTION To detect implementation faults	Detected and missed implementation faults
EVALUATION To detect requirement, design and implementation faults	Detected and missed requirements, design & implementation faults Defect age
PREVENTION To prevent requirements, design and implementation faults	Rework Prevented, detected and missed rqmts, design & implementation faults Failure Cost (COQ)

DEFECT COUNTING PROBLEMS

When to start counting
- Start of development
- Completion of the work product
- Formal configuration management
- Start of test

Which activities are considered "defect finding"?
- Inspections
- Test Executions
- Test Design and Development
- Informal review
- Special analyses

What about omissions?
- Forgotten requirements
- Omitted features

How to treat severity and impact

How to treat confusion and dissatisfaction

WHAT IS A DEFECT?

FOUR DEFINITIONS

1. **A defect is a deviation from the specification**
   Bill Hetzel, *Program Test Methods*, 1972
2. **A defect occurs whenever the software does not behave the way the user reasonably expects it to**
   Glen Myers, *Art of Software Testing*, 1979
3. **A defect is something that, if not corrected, will result in an authorized problem analysis report (APAR) or result in a defect condition in a later inspection or test stage or is in nonconformance to a documented specification or requirement.**
   IBM Rochester, MN
4. **A defect includes any deviation from the specification and also errors in the specification. (Once the specification is accepted any new features added or old features deleted are considered defects.)**
   Bob Grady, *Software Metrics*, 1989

DEFECT ANALYSIS

Successful SQM requires ongoing defect analysis to obtain feedback and calibration

Supports

Risk Analysis	Process Management
Project Management	Prevention Strategies

WHAT ARE WE DOING?

SQE Surveys of Over 6000 Practitioners

1. **TESTING PRACTICES**
   Each Test Conference since 1988
2. **MEASUREMENT PRACTICES**
   ASM 1990 Phase 1 Survey
   ASM'91 Conference
3. **ASSESSMENTS DATABASE**
   Individual surveys plus on-site interviews
   Now over 100 projects
4. **BENCHMARK DATABASE**
   10 Selected "best" projects
   Each baseline assessed

CHOOSE ONE

What we spend on testing every year is:

- **a. ☐ just about what we should be spending**
- **b. ☐ too much and must be reduced**
- **c. ☐ too little and must be increased**
- **d. ☐ a mystery to me**

THE USE OF SELECTED TEST MANAGEMENT PRACTICES

Percentage of companies reporting common use in system testing practice

	1987	1988	1989	1990	1991	1992
Cost of testing measured	24	21	25	28	31	29
Cost of debugging separated from the cost of testing	21	18	22	20	23	24
Test efficiency and effectiveness measured	15	18	14	16	15	23
The patterns of faults and defects are regularly analyzed			30	27	26	32
Testers are formally trained			26	30	29	29

Source: SQE Annual Testing Practices Survey, 1988 - 1992

THE USE OF SELECTED TEST PRACTICES

Percentage of companies reporting common use in system testing practice

	1988	1989	1990	1991	1992
Requirements coverage is analyzed	24	28	32	45	42
Design coverage is analyzed	15	17	28	34	27
Code coverage is analyzed	13	14	18	18	27
Cases and procedures are assigned unique names	52	60	58	65	61
Cases and procedures are saved for reuse	60	69	65	76	73
Test design specifications are produced	50	51	53	59	50
Test summary reports are produced	46	50	52	67	65
Tests are specified before the technical design of the software		15	15	26	24
Tests are rerun when the software changes		78	70	76	77

Source: SQE Annual Testing Practices Survey, 1988 - 1992

SOME OBSERVATIONS

1. Big gap between what we really use and the technology that is available.

2. Sometimes a big gap between what we really do and what we think we do.

3. The transfer of accepted software technology into industrial practice is very slow.

4. Measurement used to support routine analysis and decision making is minimal.

AGENDA

1. INTRODUCTION

2. TESTING AND SPECS

3. TESTING AND MANAGEMENT

4. SUMMARY
   - Major Points
   - Discussion

TWO MAJOR POINTS

KEY POINT #1: Industry struggling with basics

What are the inputs and outputs of the test process?
When does testing start and stop?
Which comes first—code or test?
What does good testing mean?
How should it be measured?

KEY POINT #2: Management Is a Big Part of the Problem

PROGRESS ??

10 YEARS AGO

WHAT TESTS? (how to choose)
WHEN to stop? (criteria)

TODAY

WHAT TESTS? + WHEN to start? (how soon)
WHEN to stop + WHY TEST (objectives and benefits)

ONE LESSON from EXPERIENCE

TESTING CAN BE MANAGED!!

Yes, success is hard to measure
Yes, nothing is very easy

BUT

Action Is Needed From You Personally

LEADERSHIP
SUPPORT
CONTROL

Pacific Northwest Software Quality Conference

1992 Software Excellence Award Winner

"Software Quality at SRC Software"

SRC Software
Andrew Ferguson
2120 SW Jefferson Street
Portland, Oregon 97201

PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

Software Quality at SRC Software

Development of the Budget Advisor

Often, entrepreneurial success walks hand in hand with a great idea. The pet rock, Soloflex, and Visicalc to name a few. One cannot help but envision geniuses huddled in a back room saying to each other, "...that is a great idea, now, how about if we also..."

This simply is not the way it was with the development of The Budget Advisor. However, like most start up software development projects, this is the way it was hoped it would be.

So what went wrong? Absolutely nothing. Quite a few things did not turn out as planned, but they were not "wrong." Mistakes, setbacks, and failures are simply a part of a successful software development process.

What went right? A lot of hard work. Expertise in the chosen field. A deaf ear to the other "experts." And an unfailing willingness to go the extra mile by listening to every end user and by providing them with exactly what they need.

The Budget Advisor (the 1992 Pacific Northwest Software Quality Conference Software Excellence Award winner) is an unparalleled budget preparation and financial reporting package. It is unparalleled in platform diversity, ease of use, flexibility, speed, price, and user satisfaction.

Until the introduction of the Budget Advisor, virtually all financial planning and reporting software packages were either written on a mainframe/mini platform, or in a data base or proprietary language environment. These environments necessitate the use of fixed and stagnant "transaction based" approaches. Furthermore, these systems are rarely more than a "data accumulation" module so that usually Lotus 1-2-3 (or some other spreadsheet program) is used to "develop" the numbers. Users simply prefer "spreadsheet" environments for budgeting and analysis.

Unfortunately for users, the "experts" (those nebulous members of society that write articles, give speeches, and haunt programming divisions of some software organizations) said that budgeting personnel simply could not have their cake and eat it too. "It is impossible to marry the sophistication, ease of use, and intuitive environment of spreadsheets with the necessary consolidation and reporting capabilities afforded by databases," the experts said. "Therefore, these financial systems must be developed within a data base or proprietary language environment."

SRC's Software Excellence Rule #1: Listen to, but do not necessarily believe, the "experts." Instead of living with "innate" or perceived limitations, try to break the bonds of "impossibilities" and venture into the grounds of inspiration and innovation.

SRC Example #1 At SRC Software, the huddled geniuses, the true inspiration behind the development of the Budget Advisor, are the clients, the end users. They demanded a financial planning package that would allow for the continued use of spreadsheets yet provide the necessary consolidation and reporting. SRC simply responded (and continues to respond) to this demand.

SRC performed the standard research necessary to determine if indeed a spreadsheet environment was too limiting. In addition, a significant effort went into reviewing the "wish lists" resulting from standard market demand analysis. Two person years later, in March of 1988, SRC Software introduced The Budget Advisor Version 1. This was the first (and still only) spreadsheet based multi-department budget preparation and financial reporting system.

The Budget Advisor, in its purest form, is a sophisticated multi-spreadsheet consolidation and reporting system. Written in a Lotus 1-2-3 environment (all releases), the Budget Advisor is designed to easily and automatically consolidate up to 995 dissimilar spreadsheets, and then report upon these numbers in any manner, automatically.

The Budget Advisor allows users to take full advantage of the 1-2-3 features that have become an indispensable cog in the wheel of business, while overcoming its many inherent limitations.

The many features of the Budget Advisor include:

* Multiple Departments/spreadsheets (up to 995 per copy)
* Flexible Chart of Accounts (up to 995 "natural classifications" or pseudo accounts for a total of nearly 1,000,000 accounts per copy)
* Easily accepts downloads from and then uploads back to any general ledger or other system
* 100% Customizable to fit individual requirements
* Centralized or decentralized processing (download to distributed diskette, network compatible, etc.)
* Full spectrum of consolidation possibilities (all departments, all departments for manager "Smith," all revenue departments, by region, etc.)
* Flexible Report Generation (report on almost any information you wish with desktop publishing quality, run any dept. or grouping through any report, easily develop reports yourself)
* Global Assumptions - "what-ifs" (make one global change and it will automatically impact all associated spreadsheets)
* Fully menu driven with on-screen help

In addition, the Budget Advisor can integrate the numbers, graphs, and text often required in the budget presentation area. This "desktop publishing" ability is particularly important in the local government market (cities & counties) and other markets where the presentation of budgets is paramount.

As SRC's diverse clientele attests, the Budget Advisor is a truly flexible software system. Although to date its primary applications have been in the budget preparation, financial planning, and reporting areas, the Budget Advisor has served other purposes including, for example, multi-company consolidation and analysis with automatic intercompany eliminations in a tri-currency environment.

After the introduction of the Budget Advisor, could it be said that SRC was done? Of course not. This simply represented their best shot.

SRC's Software Excellence Rule #2: Be an expert in your field. Let your first effort represent your best effort. Yet never doubt for a moment that the end user is the true expert. No matter how innocuous a "software upgrade suggestion" may seem, provide it.

SRC Example #2: When SRC introduced the Budget Advisor, it was in response to prospects' "wish lists." The wish list or "needs assessment" is often the only source of information for developers. In the case of SRC, it was merely the starting point.

Like the competition, SRC Software made the mistake of assuming a few "standard" budgeting and reporting formats would be enough to meet clients' needs. "Now that we have a system in a spreadsheet environment," it was thought, "we have held up our end of the deal."

Wrong. Six installations and six full "customizations" later, SRC realized that there is no "standard" approach to budgeting. It is a very personal endeavor. No one organization nor individual wants to budget exactly like another. Yet SRC did not wish to be in the customization business. It was too time consuming, less profitable, and created "upgrade" problems. Enter drawing board number two.

SRC rewrote the Budget Advisor almost from the ground up with one key parameter in mind; "easy flexibility." The Budget Advisor would be known as the "Burger King" of budget prep - by allowing users to "Have it your way." Virtually every feature of the software, from columnar layout, to spreading and budgeting methodologies, to consolidation grouping codes, to reporting, was rewritten to be 100% "user-definable."

The software then had three distinct "segments"; a modular software framework; client specific parameters; and the client data. Any one of these three segments can be altered without negatively affecting the other two. Using this approach, SRC's Budget Advisor was able to continue using a spreadsheet environment while maintaining the highest level of flexibility, upgradability, and user definability.

Software installation and support Once a client has elected to acquire the Budget Advisor, SRC sends a letter requesting information relating to their current (and desired) budget system. Information requested includes sample budget input forms (their true data sources), budget reports (for individual departments and consolidations), and various materials relating to their current hardware and software. SRC's Consulting Manager then reviews all these materials in conjunction with phone discussions. This allows for a clear identification of the optimal budget layout (column headings) and methodology (the page layout and the formula interrelationships).

The Consulting Manager transfers the materials and conversation notes to the Consultant assigned to the job. The Consultant sets up a draft budgeting system at SRC's site including the optimal columnar structure, the client's chart of accounts, department listing, global assumptions (for what-ifs and flexing), department budget format and formulations, and several sample summary reports. This process usually involves several discussions with the client.

Once the framework has been set-up (1 to 2 days) structured test data as well as actual client data are loaded into the system and tests of formulations, relationships, and reasonableness are performed and documented. The Consulting Manager performs a quality review of the "draft" system well before the implementation dates.

This high degree of preliminary preparation and quality review ensures that the on site training and final customization is as productive and effective as possible.

When on site, the Consultant assists the client in installing the system, reviewing the various planning modules, and, in particular, helps the client develop their own budgets and reports. This hands-on approach provides the client with a high product confidence level by the end of the on site training. SRC has found that almost all clients prefer to learn "how to..." as opposed to simply watching the Consultant work with the system.

Upon returning, the Consultant's interaction with the client and minor system changes are reviewed by the Consulting Manager. The client's system is then loaded onto SRC's LAN and maintained for quick access by all Consultants. Whenever a client calls with support questions, any SRC consultant has instantaneous access to that client's personal system.

Client interaction does not stop after the implementation. The Director of Consulting Services personally calls each client approximately four weeks after the on site implementation. Client satisfaction, suggestions, additional needs, etc. are discussed candidly. Any follow up work needed to achieve "total satisfaction" is promptly performed. This comprehensive client support and system quality review has provided SRC with a client base that more closely resembles a sales force.

SRC's Software Excellence Rule #3: Do not sit on your laurels. No matter how successful the software seems, always seek ways to make it better.

SRC Example #3: SRC tracks each user's satisfaction with multiple mechanisms. This starts with "pre-sale" conversations including detailed notes. All suggestions or requests that are not standard features are documented and maintained in the "upgrade suggestions" file. Many times these suggestions come from the review of RFP's (request for proposals).

SRC also keeps detailed notes during the implementation. This is particularly important for determining which features may have been misunderstood or simply not known to exist.

Also, each user is sent an exhaustive "Performance Questionnaire." This includes detailed questions about the software, the consulting staff, the operations staff, marketing materials and staff, and provides each user with many opportunities to list what they like and what they do not like. Each questionnaire is then tabulated and maintained in a database for follow up analysis and software strategic planning.

Some users need to be repeatedly asked to "please return the questionnaire." SRC maintains an assertive yet unintrusive attitude when following up with users. Often times, those least eager to respond to the questionnaires ultimately provide the most valuable input. When all else fails (e.g., the third mailed request) SRC personnel call the users to get a verbal response.

SRC also maintains detailed client support notes. Each time a user calls, whether just to chat or with a problem, SRC's staff fills out a client support form. These support forms include detailed lists of problem "types" for easy and informative categorizations. Like the performance questionnaires, these support notes are then entered into a database for analysis. In addition, at the end of each year, users are sent an exhaustive analysis of support services provided. Users appreciate this because it shows the value of their maintenance dollars, and perhaps indicates areas that may need improvement.

SRC Software takes its clients very seriously. Many organizations' primary focus is on acquiring new clients. While this may be more productive in the short run, SRC believes that maintaining a highly satisfied user base will ultimately lead to more sales. SRC is proud to say that by far the best selling technique for the Budget Advisor is to have a potential client contact any user (not just a select few) for a reference.

Maybe as importantly as tracking users' suggestions, SRC makes it a policy to listen intently to non-clients. For example, when SRC experiences a lost sale, the reasons (or lack thereof) behind the decision are far more important than the loss of revenue.

SRC's Software Excellence Rule #4: Always ask the following questions of lost sales or dissatisfied users: "What are the perceived benefits of the alternative?" "What are the perceived shortcomings of our solution?"

The answers to these questions are the lifeblood of competitive advantage in any industry for any product. You must know your strengths and weaknesses. Then you can fortify your strengths and overcome your weaknesses.

SRC Example #4: It was "perceived" by some prospects that the Budget Advisor was nothing more than a "giant spreadsheet." One lost sale went as far as to say, "Yes, your software beat the competition across the board. It is faster, more sophisticated, easier to use, and less than half the cost of the competition. However, we just can't rationalize spending this much money for an 'overblown spreadsheet'."

Here was a prospect admitting that the only reason for selecting the competition was one of perception and not fact. SRC's response to this problem was to upgrade the look and feel of the menu system to be easier to use, but more importantly, to have a more generic appearance.

Although this is both a software and marketing example, it exemplifies another rule for software success.

SRC's Software Excellence Rule #5: You do not have to agree with the users - even when "perception" overrides "fact." If the user "perceives" an advantage, provide the feature.

SRC Example #5: Often times, SRC will receive software suggestions that in hindsight would never have been envisioned. As hard as one might try, and as "expert" as one might be, one can never know for certain precisely what the end user needs. For example, one client asked for the option to print only those budgets that had been worked on from one date to another. Great idea - now a standard feature.

Conclusion:

SRC has made it a policy to provide each end user with precisely what they want. The word "no" has seldom reared its ugly face when a prospect or client asks for something new. Rather, SRC responds with, "admittedly that would be a new feature, but we will be happy to provide it." The client is the expert.

What determines software excellence? The user. The best programmers in the world, the finest marketing staff available, and the best ideas will not necessarily result with software excellence. Only the user can define this, and only the user can determine success. The user must be the driving force behind the development process.

SRC focuses its entire software development around the user. It is SRC's goal to have 100% client satisfaction - with no exceptions. SRC believes it is because of this attitude, and the willingness to follow through with the necessary hard work, that excellence has been attained in software development.

Dr. Linda M. Beach
Director, Quality Assurance

NYMA, Inc.
7501 Greenway Center Drive
Greenbelt, MD 20770

Abstract

This paper discusses the implementation of Total Quality Management (TQM) concepts in a software operations and maintenance environment. In-process quality initiatives, measurement critieria, and the benefits of the program are presented.

Biography

Dr. Beach is the Director of Quality Assurance for NYMA, Inc. She teaches Quality Assurance, Testing, Requirements Analysis, Group Dynamics Methodology, and Computer Sciences.

Dr. Beach has successfully led NYMA's quality efforts to achieve such accomplishments as the 1991 Goddard Award for Excellence in Quality and Productivity, finalist in the 1991 RIT/USA Today Quality Cup Competition, and submissions for the 1992 NASA Quality and Excellence Award.

INTRODUCTION

NYMA Inc. took its first steps toward implementing Total Quality Management (TQM) in 1987. The corporate culture, work force, and work environment contributed significantly to the early success and continued improvement of the program. Recent corporate quality initiatives, based on lessons learned, have brought about an acceleration of TQM practices for all projects. The basic quality approach is that both the customer and employees form a team to accomplish total customer satisfaction. NYMA has successfully applied TQM to software projects. The scheme developed to work with our customers and employees is presented in this paper.

BACKGROUND

As a subcontractor to IBM, NYMA's Host Support Software (HSS) contract provides computer software and data base maintenance for the seven east coast FAA enroute Air Route Traffic Control Centers (ARTCCs). The HSS computer software and data base support the air traffic controllers as they direct aircraft through the airspace sectors within each ARTCC. This critical application includes making software changes to support local site requirements and system features, integrating all changes into the master (national) system updates provided by the FAA, inserting and maintaining data base adaptations to accommodate site-specific requirements, and thoroughly testing all patches and system updates.

APPLIED EXCELLENCE PROGRAM

During project reviews held in 1988, one theme held constant: the number of errors at each site would have to be reduced. Although not severe in nature, these errors required the controllers, who had been briefed and trained in the implementation of the software changes, to modify their daily operational tactics.

Employees became determined to accomplish two goals: (1) to ensure the FAA controllers receive the benefits from the software products and (2) to prove to IBM that they could produce a quality product. NYMA management planned to reward sites whose deliveries were free of errors with a plaque citing their accomplishments and a celebration dinner for all site employees and their guests.

That first year, employees delivered error free systems at four of the seven sites. Each site received a plaque noting its accomplishments. Management decided that the sites should be eligible for this reward on a continual basis. Thus, the need for a formal quality assurance incentive program was initiated.

To continue with the incentive, a quality program called Applied Excellence was implemented. The purpose of the program is to achieve project quality and productivity goals and provide the means to reward employees for contributions. Figure 1 summarizes the implementation guidelines and eligibility criteria developed for the sites.

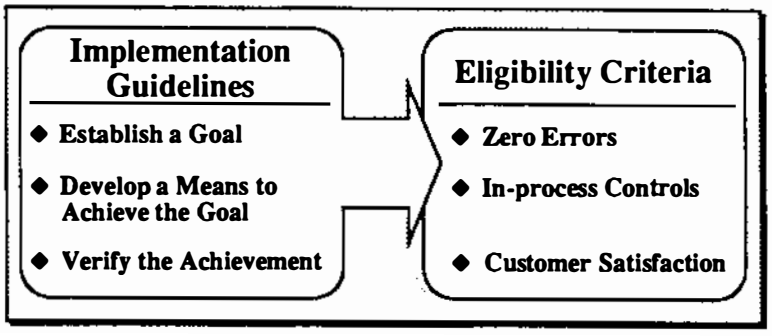


Figure 1. Implementation and Eligibility Criteria

To receive the Applied Excellence Award, each installed system must be error free. The system must be installed, operational, and running for 7 days without a system error. After 7 days, the customer verifies the system reliability, and employees are eligible for the award.

Since implementing the Applied Excellence Program in 1989, the number of errors on delivered systems has declined, the number of awards has increased, and the number of system updates has decreased as a result of fewer errors.

IMPLEMENTATION

Our teams accomplished this extraordinary record in a short time. Employees developed the program and were committed to its success. They developed in-process controls that would provide the quality systems they wanted. Several in-process quality initiatives have been implemented including peer reviews and testing, audits, standards, training, measurements, and a reward system. These are described below.

Peer Reviews and Testing

Project managers from all seven sites concurred on the need to formalize peer reviews and testing efforts. It was agreed that the they would get their employees together and "brainstorm" new methods. After these meetings, the site project managers met to assist in defining the core methods for all sites.

The methodology that evolved to ensure employees would do the job "right the first time" (Figure 2) is unique for the following reasons:

- ***It was developed by the employees***
- ***It involved the developers, customer, and users***

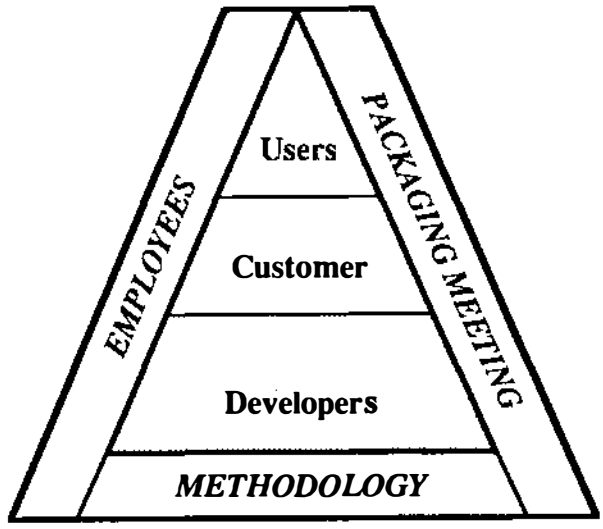


Figure 2. Doing the Job Right the First Time

All software work packages are peer reviewed at a *packaging meeting* with site software engineers, IBM management, and FAA staff personnel who have submitted change requests. At the meeting, FAA requirements are reviewed and clearly understood, and the methodology that will be used to make the changes and the test procedures to ensure accuracy are discussed and agreed on.

The completed software work packages are subject to the same peer review process. The software engineers must demonstrate that the work packages are implemented and tested according to agreements reached during the packaging meeting. Testing results are reviewed with the group to ensure that the expected results occur. If additional tests are required, the software engineer provides the rationale, and the documentation is presented for review.

Audits

Another initiative involved site self assessments. There were two objectives: to identify areas that needed improvement and to identify "best practices," for all sites. Eventually, all sites were standardized with proven operating procedures. For the audit, a 92-item checklist designed to identify work processes, documentation, standards, and employee understanding was developed.

The audit team consisted of the technical support engineer from the main office and a person from another site. ***The audit criteria and implementation are accomplished by peers, not by a management team.*** The audits surfaced several common items that needed improvement at all sites. However, most of the deficiencies centered in two areas: the lack of standards and the lack of meaningful training.

Standards

During the site audits, work processes identified as extremely well implemented were earmarked as potential "best practices" for all sites. Site-specific standard operating documentation was then compiled, reviewed, and revised. A standard operating manual was developed. As a result, all of the sites operate under a written set of standard operating procedures. Each site, of course, is free to adapt the standards for their specific working environment.

Improvements to standards covering test and implementation, adaptation, and software maintenance have proven extremely beneficial to the teams. These standards provide a quality control checklist to ensure that all steps are completed. They are also valuable in training new employees.

A technical library has been developed to aid personnel in maintaining familiarity and technical knowledge of field operations. Included in this library are:

- System level computer listings
- Computer program functional specifications

Training

HSS employees assisted IBM in developing training courses. These courses are used to introduce new employees to the FAA and the FAA automation systems. The first six courses, as shown in Figure 3, are self study courses. Students complete these courses during their on-the job training period. This approach enables the students to immediately apply the concepts that they learn and also reinforces retention of facts studied. Newly hired employees complete these courses as part of their indoctrination program. Courses 7 through 9 are conducted in a classroom. They are designed to provide advanced concepts to enhance the employees ability to respond to daily operating activities.

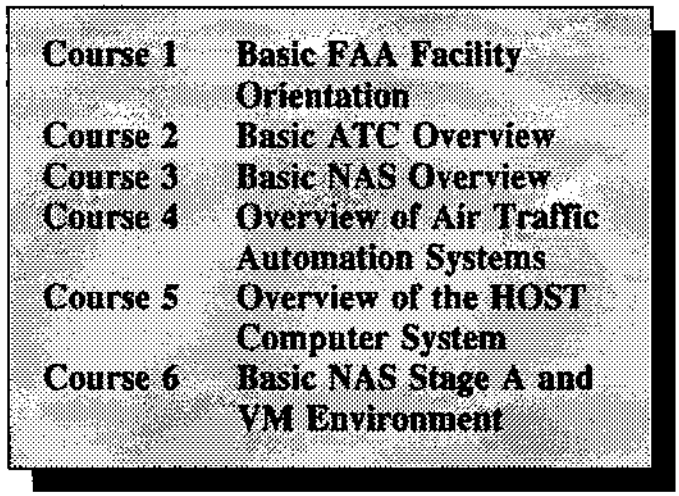


Figure 3. Training Program Courses

Measurements

Current measurement activities focus on five areas: product quality, productivity, problem analysis, software system effectiveness, and customer satisfaction. Figure 4 presents a summary of the statistical data used to measure these areas. These measurements are intended to help the teams maintain quality through trend analysis and problem prevention.

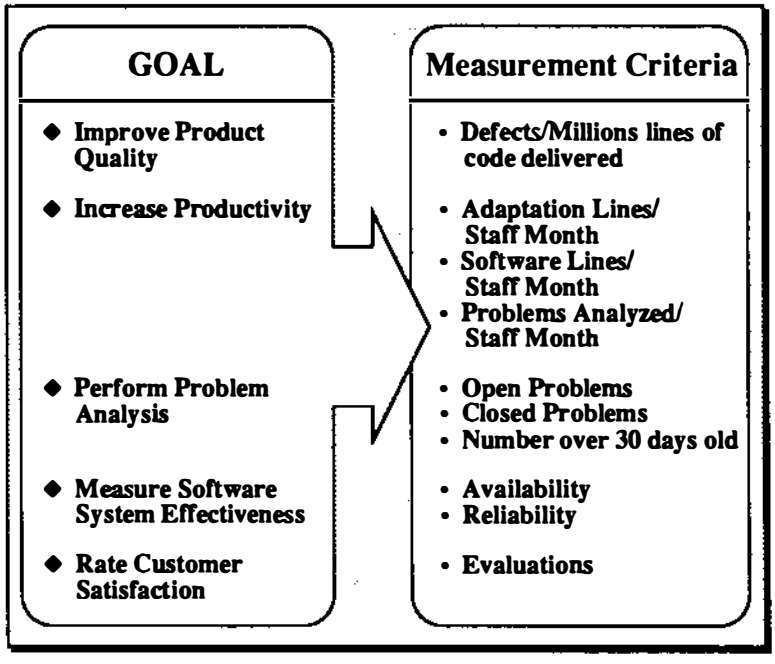


Figure 4. Statistical Data Measurements

Measurement techniques were introduced at the sites in 1992.

The quality monthly reporting criteria have been standardized. Procedures were developed to explain the purpose, goal, and methodology for reporting data. Such standardized guidelines assist each site in preparing and developing the required data.

Rewards

In addition to the Applied Excellence Award, there are several other awards earned by employees for individual quality and productivity improvements. These awards include:

- ***NYMA Employee of the Year***
- ***Division Employee of the Year***
- ***Division Team of the Year***
- ***Individual Applied Excellence Award***
- ***Customer Satisfaction Award***

CONCLUSION

Employees have benefitted from the Applied Excellence Program. They receive personal satisfaction, tangible rewards, and corporate recognition for their efforts. Attitudes have changed; the once seemingly impossible goal has become the standard. The initiative and success of the teams have had the single most significant impact on NYMA's total quality program. The teams have emphasized that quality is every employee's responsibility. The NYMA quality initiative for 1990 was to expand the Applied Excellence Program to all NYMA projects. During 1991, three projects qualified for the Applied Excellence Award.

As with any new program, NYMA learned a great deal from the implementation of the Applied Excellence Program. First, top management commitment and oversight is absolutely essential to implementing and maintaining such a quality program. Second, employees know their job best, and are in a better position to provide continuous improvement initiatives. Third, all parties need to be actively involved in developing quality initiatives. This concept was proved with peer review and testing methodology, which were developed by the employees and involved the developers, customer, and users. Finally, audits, developed and conducted by employees, were more stringent than management could devise. The employees were closer to the daily operational aspects of the project. Employees were committed to the improvement process that followed because of their initial efforts in developing the audit criteria.

NYMA has experienced exceptional growth and recognition as a result of our demonstrated ability to satisfy customer needs with high quality services and products. In 1991, NYMA was a finalist, out of 432 applicants, for the Rochester Institute of Technology (RIT)/USA Today Quality Cup. This is a national award that recognizes

teams who make significant contributions to the improvement of quality products. We were awarded the Goddard Space Flight Center Excellence Award for Quality and Productivity, which recognizes achievements and demonstrated accomplishments in quality and productivity. Patrick Air Force Base recognized NYMA as the National Small Business prime contractor. This achievement is in recognition of integrity, reliability, initiative, and capabilities in meeting significant government requirements.

Our next step is to refresh the Applied Excellence Program. Our commitment to produce a quality product for our customer is a long-term effort. To this end, we plan to continuously review our goals, look at new ways to improve, and reenforce our customer focus.

Improving Quality and Reducing Costs Using the Software Engineering Institute (SEI) Paradigm

by

Mary Sakry
&
Neil Potter

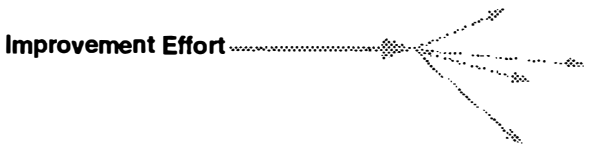
of

The Process Group
(214 437-3028)

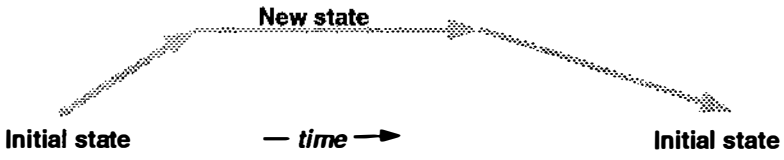
THE PROCESS GROUP

Typical Problems of Improvement Efforts

- **Improvement effort becomes unfocused and vanishes**



- **New methods are forgotten over time**



2

THE PROCESS GROUP

Possible Causes

- **Misunderstanding of the problem being solved**
- **Lack of consensus in the organization as to which problem to solve**
- **Implementation of solution poorly managed**
- **Everyone already too busy**

3

THE PROCESS GROUP

Basic Requirements of a Successful Improvement Program

- An **understanding** of the current status of the development process
- A **vision** of the desired process
- A **prioritized** list of required **improvement actions**
- A **plan** to accomplish these actions
- The **resources and commitment** to execute the plan
- **Agreement** that:

Known Problem = Opportunity for Improvement

4

THE PROCESS GROUP

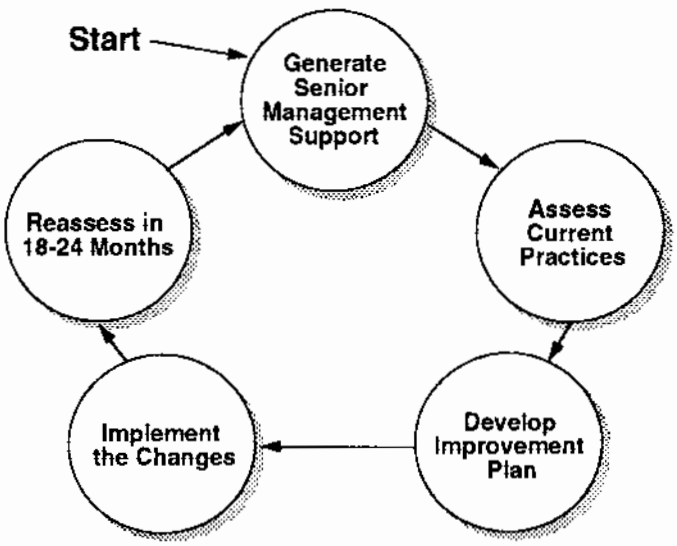
Principles of Process Change

- Major changes must start at the top
- Fix the process, not the people
- Understand the current process first
- Change is continuous
- Improvement requires investment
- Retaining improvement requires periodic reinforcement

5

THE PROCESS GROUP

Process Improvement Cycle



6

THE PROCESS GROUP

What is an Assessment?

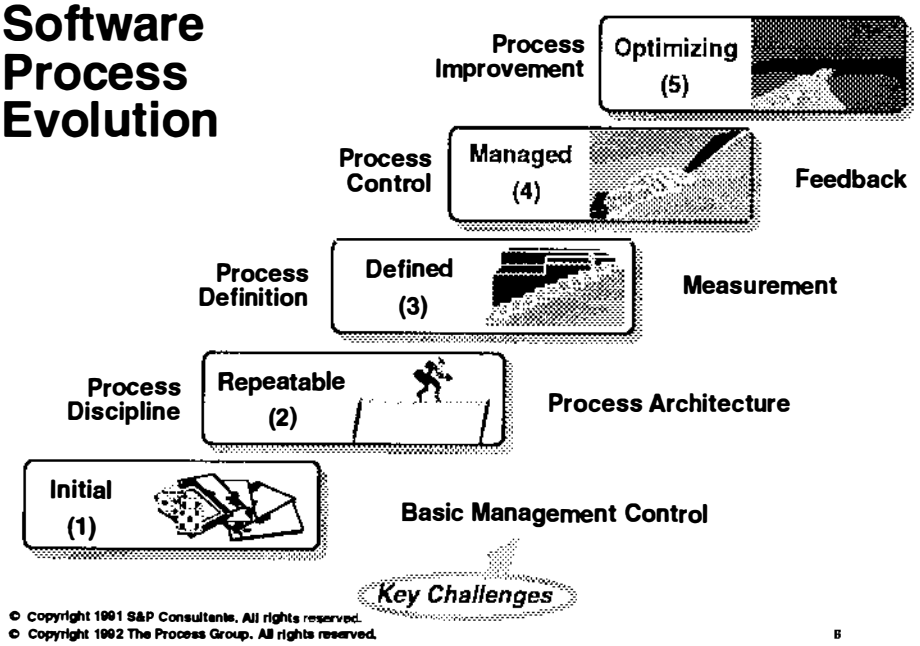
An appraisal, by a trained team of experienced software professionals, of an organization's current software process, based on

- **Review of 4 to 6 key projects**
- **Responses to assessment questionnaire**
- **In-depth interviews of project managers**
- **In-depth discussions with functional area representatives (practitioners)**
- **Collective assessment team knowledge and experience**

7

THE PROCESS GROUP

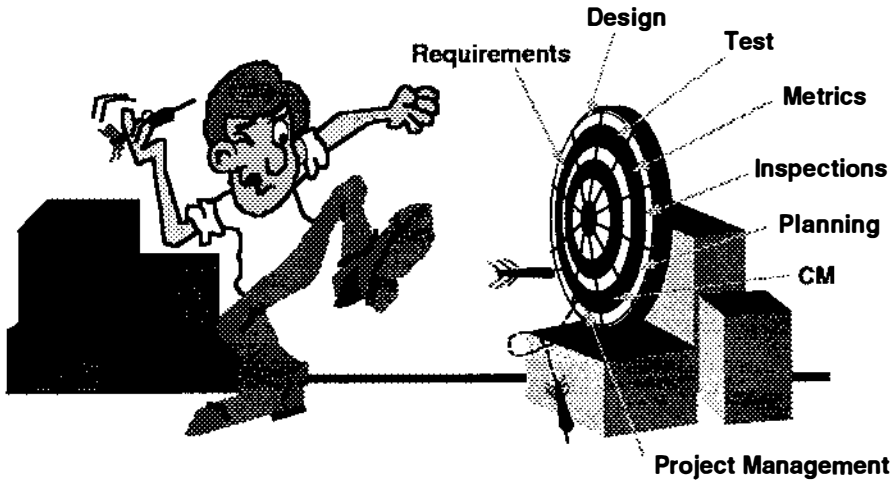
Software Process Evolution



6

THE PROCESS GROUP

Selecting What to Work On



THE PROCESS GROUP

Typical Approach to Improvement

Detailed Process	Task:	Develop a well-defined process and suite of standards
	Notion:	A detailed process, when followed, will result in a quality product
	Problem:	At level 1: - little time to write or adhere to a well-defined process - a well-defined process is too complex and provides too much information
	Result:	Process is ignored

10

THE PROCESS GROUP

Alternate Approach

SEI Maturity Model	Task:	Address project management and planning first
	Notion:	Establish predictability with respect to cost, schedule and resources
	Result:	Clearer understanding of commitments
		A more accurate schedule that sets realistic expectations

Process stability is essential before more advanced issues can be addressed

11

THE PROCESS GROUP

Oh no, not more standards!

The maturity framework does not advocate:

12

THE PROCESS GROUP

Tools and the Maturity Model

The Maturity model introduces tools based on the organization's ability to use them productively.

Level 2 tools include: ***Planning, configuration management, estimation.***

Level 3 tools include: ***Requirements analysis, testing (coverage & regression), design, standards checking, rapid prototyping.***

Note

No tool will solve a problem on its own.

For every tool introduced 1) the initial problem must be understood, 2) a solution must be developed (method/process), 3) the solution should be practiced, and then 4) opportunities should be identified for automation.

Jumping to step 4) will just delay step 1)

13

THE PROCESS GROUP

Assessment Advantages

- **Provides a framework for process improvement**
- **Identifies the key problems, with consensus, from the participants**
- **Stimulates people to expect change**
- **Uses the knowledge of the organization to change itself**
- **Sets priorities to ensure that time and money are spent wisely on improvement**
- **Enables investment to be spread over a long period of time**
- **Fast and accurate**

14

THE PROCESS GROUP

Software Engineering Process Groups

A Software Engineering Process Group is a group chartered to facilitate software engineering process improvement within an organization. It helps the organization determine areas for improvement, plan the improvement, and implement it.

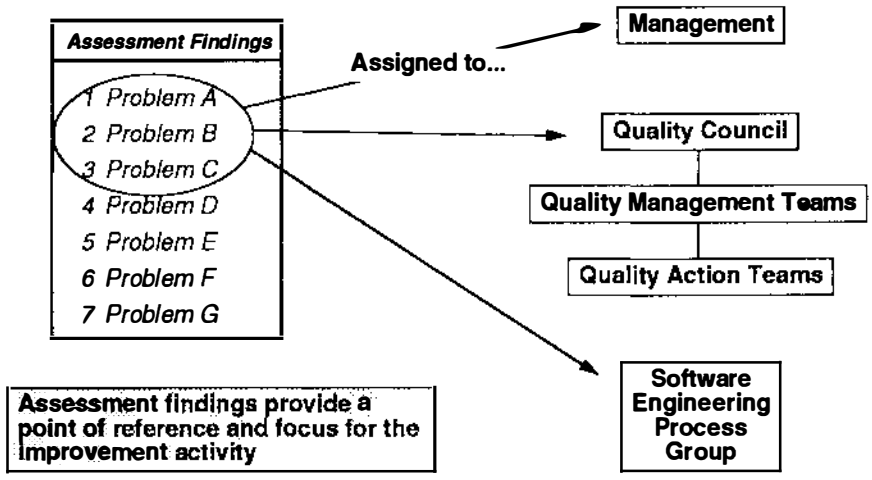
THE PROCESS GROUP

Relation to Software Developers

- Comes from development staff
- Has no authority over the developers
- Helps developers adopt new practices (provides a service that helps them do their improvement work)
- Involves the developers in the development of new practices and procedures
- Maintains support throughout the development organization (gets success stories, understands issues, communicates up/down/across the organization)
- Promotes a collaborative attitude

THE PROCESS GROUP

Improvement Using Existing Company Resources



17

THE PROCESS GROUP

Typical Things that an SEPG Does

- Sets expectations
- Conducts software engineering process assessments
- Helps analyze expectations versus current practices
- Helps write plans for improvement
- Helps implement the plans - facilitates improvement
- Maintains sponsorship
- Acts as an information source, sharing information across the organization
- Promotes and teaches new behaviors (process consultation and technology insertion focal point)
- Writes processes and procedures
- Measures process improvement
- Long-term: Sets up and maintains a process database

18

THE PROCESS GROUP

Things that an SEPG Does Not Do

- Audit or report specific problems to management
- Reveal specific confidential information to management or others
- Take responsibility for the actual improvement
- Write processes or procedures in a vacuum
- Issue edicts or strong-arm changes
- Ask for data or information that will not be used for the improvement effort

THE PROCESS GROUP

References

1. Brooks, F., "No Silver Bullet, Essence and Accidents of Software Engineering," *IEEE Computer*, April 1987.
2. Olson, T. G., Humphrey, W. S., and Kitson, D. H., "Conducting SEI-Assisted Software Process Assessments," SEI Technical Report CMU/SEI-89-TR-7, Software Engineering Institute, Carnegie Mellon University, February 1989.
3. Humphrey, W. S., *Managing the Software Process*, Addison-Wesley, Reading, MA, 1989.
4. Humphrey, W. S., Kitson, D. H., and Kasse, T. C., "The State of Software Engineering Practice: A Preliminary Report," SEI Technical Report CMU/SEI-89-TR-1, Software Engineering Institute, Carnegie Mellon University, February 1989.
5. Boehm, Barry. W., *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice Hall, 1981.
6. Fowler, Priscilla and Rifkin, S., "Software Engineering Process Group Guide," SEI Technical Report CMU/SEI-90-TR-24 ESD-90-TR-225, Software Engineering Institute, Carnegie Mellon University, September 1990.
7. Peters, T., Austin, N., *A Passion for Excellence*, Warner Books, 1986
8. Walton, M., *The Deming Management Method*, Perigee Books, NY 1986
9. Humphrey, W. S., "CASE Planning and the Software Process," SEI Technical Report CMU/SEI-89-TR-26 ESD-TR-89-34, Software Engineering Institute, Carnegie Mellon University, May 1989.
10. Zultner, R., "The Deming Approach to Software Quality Engineering", Quality Progress, pp58-64, November 1988
11. Humphrey, W. S., Snyder, T.R., Willis, R.R., "Software Process Improvement at Hughes Aircraft", IEEE Software, pp. 11-23, July 1991

SUCCESSFULLY USING AN EVALUATION TEAM TO IMPROVE PRODUCT QUALITY

Barbara A. Siefken
Tektronix, Inc.
PO Box 500 MS 39–750
Beaverton, Oregon 97077

e–mail: barbaras@tekig1.PEN.TEK.COM

Abstract

The traditional method of throwing the final product code over the wall to evaluation shortly before the desired product ship date often causes frustration and friction between design and evaluation. Our evaluation team adopted a group philosophy and testing strategy which lessened these problems and left more time for product improvement. The result was a product introduction with 1/20th the software bugs of previous products, and an evaluation team challenged and excited about their work.

Biography

Barbara A. Siefken is a Senior Software Engineer at Tektronix, Inc. She is currently evaluating a new product and is involved with design reviews, code walkthroughs, bug tracking, code quality, stress testing, and functional testing.

Previous projects at Tektronix include TDS series digital oscilloscope evaluation and software development for the Plot10 graphics applications library STI. Prior to that she worked at Bell Laboratories in Holmdel, New Jersey on a voice store and forward system and the Merlin Telephone System.

SUCCESSFULLY USING AN EVALUATION TEAM TO IMPROVE PRODUCT QUALITY

OR

LET MANAGEMENT THINK THEY'RE DRIVING AND THEY WON'T GRAB THE WHEEL

Introduction

The traditional view of an evaluator is that of either a repressed design engineer passing time until a ***real*** job comes along, or, even worse, a design engineer who failed. The traditional function of an evaluation group is as a quality gateway for the final product code, operating prior to the product ship date. There is pressure from the engineering team to give the product code the "stamp of approval". Evaluation has a "gut feel" about product usability, but can't seem to convince engineering it is important to fix the bugs they have found. This often causes frustration and friction between the engineering **team** and the evaluation **group**.[1] Meanwhile, marketing and management want the profits from product sales, and pressure engineering to ship the product as soon as possible.

A key ingredient to success as an evaluator is establishing credibility with a wide range of people, primarily management and the engineering team. The scenario described above not only hampers the establishment of credibility, but also creates an adversarial relationship between evaluation and engineering, with evaluation dampening the partyspirit by uncovering bugs. This relationship does not encourage outstanding accomplishments, continuous improvement, quality products, or even a nurturing work environment.

The software evaluation **team** at Tektronix adopted a group philosophy and simple testing strategy which lessened these problems and left more time and effort to spend working on product improvement. The result was a product introduction with one-twentieth the software bugs of previous products, and an evaluation team challenged and excited about their work. This paper outlines some of the methods used to accomplish this result.

The products, a series of GPIB programmable Digital Oscilloscopes, were introduced two at a time, six months apart. The human interface model was completely new. Each product used multiple processors and implementation languages.

Please be advised that this paper does not claim to have a formula for evaluating products. The specific actions we took may or may not be appropriate for your company. Choose an implementation that fits the personality and talents of the members of your team. Focus on a common goal, like introducing a product with half the defects of its predecessor, and strive to communicate the evaluation team's perspective of the product quality.

[1] The use of the words **team** and **group** here are not arbitrary.

Establish Credibility

Make A Good First Impression

Initially, one senior engineer was hired as a Software Quality and Evaluation Champion to set quality standards, develop a process for design and code reviews, and define and manage a software evaluation strategy. She had ***chosen*** to do software evaluation and was excited about it. She believed that the product being developed would have a positive market impact, and wanted to contribute to the product's success. She had prior experience working on successful products.

The hiring manager made a statement by selecting her. This task was important, and would make a difference to the product team.

Design and Code Reviews

The SQE Champion became the review facilitator. She defined the structure of, scribed at, and ran the reviews. There were twenty design and eighteen code reviews conducted over a seven-month period. The intention was to make the reviews productive and pleasant, and to create a positive work environment in which to exchange ideas.

The facilitator reserved specific rooms and time slots – three reviews per week for a six-month time period. This set a pattern for review times making them easier to remember, and eliminated scrambling to find a room for each review. She posted review times to the design engineers' calendars using the SUN calendar tool and an appointment file. In addition, a wall calendar in the hall and a large poster in one of the general meeting rooms showed the review schedule. This kept people informed of project progress, even when they were not directly involved in the review process.

She gave at least two weeks notice to allow time to review documents or code. She reminded participants about upcoming reviews, and rescheduled the reviews if participants did not have time to prepare.

Reviews started on time - fifteen minutes past the hour, to allow the attendees of the previous meeting to leave the room. All presenters used a microphone directly connected to a tape recorder. For each review, the facilitator brought flowers, cookies, and a poster as a gift for the presenter. Each review began with the participants sharing something they admired about the presenter with the group. At the end of the review, she gave each participant one flower, and gave the remainder to the presenter.

The cookies could be eaten anytime. They became a trademark of the reviews; something to look forward to and to speculate about. She had a chance to find out more about the engineering team through their opinion about cookies. Participants used cookies to acknowledge anothers' design alternative ("Good idea, have a cookie!"), and to crunch on vigorously when frustrated ("What do you mean you changed the interface?").

Within two days, the scribe typed the review notes, distributed them via electronic mail, and posted a copy on a common computer directory. She recorded action items along with the name of the person who suggested them, and the expected completion date. The scribe followed up on the action items until they were completed.

In spite of the pleasant atmosphere and all the review reminders, when the pressure of implementing the product features really got turned up by the Program Manager and the Engineering Manager, the engineering team stopped preparing for the reviews, and the reviews lost their effectiveness. Reviews were replaced by an "expert engineer gate", which meant **any** code change required the review and approval of one of three selected engineers from the engineering team.

Reviews are only effective if management supports the recommendations of the experts they hire. In one review, a cryptic variable naming convention which was used confused the reviewers and hampered their understanding of the code. However, after discussion, the reviewers felt that once you understood the model, it was consistent, so why not leave it? The evaluator objected, but was overruled. Today, an engineer on a follow-on project is struggling to learn this code because of that same cryptic convention.

Set Goals and Expectations - The Evaluation Plan

The evaluation strategy was defined in the Evaluation Plan presented by the SQE Champion. The plan was accepted by the engineering team project leaders, manufacturing, management, and marketing. The fact that the Evaluation Plan was ***approved*** and ***signed*** by all concerned proved to be very important in later phases of the project, both when management changed, and when the engineering team requested that more sub-systems be tested. The Evaluation Plan now serves as a template for new projects, reducing the time it takes to write new plans and providing consistency between products.

Product Ship Targets

The following Product Ship Targets were agreed upon as part of the Evaluation Plan acceptance process:

- The random GPIB command and Front Panel stress tests run for at least 80 hours.
- The product meets all advertised specifications.
- The estimated discovery date of the next level 1 or level 2 bug is at least three months away.
- The code has a minimum settling time of three weeks. During this settling time, the full regression test suite is run.

In addition to the Product Ship Targets, the Evaluation Plan specified exactly which sub-systems would and would not be tested, and defined the system white box tests, black box tests (from the customer point of view), stress tests, performance tests, GPIB tests, and expert user tests to be performed.

Launching the Team

In the negotiation that led to the acceptance of the Evaluation Plan, it became clear that although the engineering team would test sub-systems, there was no process for extensive system testing, and no likelihood that the engineers would perform those tests. At this point, management agreed to form an evaluation team. Two senior engineers and four interns joined the initial SQE Champion to perform the testing. The engineers came from other projects inside the company, and the interns came from outside the company.

The team initially set forth the following goals:

- Learn the products.
- Use as much existing code as possible for the testing.
- Run tests which use equipment we can borrow.
- Find lots of bugs.
- Maintain and foster the use of the on-line bug tracking system, DDTS.
- Test the full functionality of the product.
- Automate when possible.
- Cross-check each other's work.

It was extremely difficult in the early weeks for the new evaluation team to establish credibility with the engineering team for several reasons. The engineering team did not know the members of the evaluation team. The interns were given computer login names of eval1, eval2, eval3 and eval4, which de-personalized them even further. Although the evaluation team members quickly learned where the development engineers' desks were, the opposite did not occur. Due to this lack of familiarity and contact, an engineer might temporarily ignore an unclear bug submitted by an "unknown". He/she might seek more information if the same bug were submitted by someone he/she knew and came in contact with frequently. Hence, we had to be proactive in our communication.

To address these issues, we changed the login names of the interns to reflect their real names, began attending engineering meetings, and scheduled one-time classes given by development engineers on their sub-systems. The classes served to introduce us to the engineers and also teach us more about particular sub-systems.

One of the senior evaluation team members wrote a detailed test plan. Using networking skills and the on-line electronic mail system, he found an existing functional test tool and two types of stress tests being used on other projects. Test tool modification and test development was ready to begin.

Stress Testing

The most useful testing tools available on this project were the two types of Random Stress Tests, which both execute via the GPIB interface. One random stress test exercises the GPIB command set, and one exercises Front Panel Button Push commands. Each test runs to completion without regard for correct functionality, and only exits when GPIB communication stops. These tests were invaluable in attaining our quality goal for Mean Time Between Failure (MTBF).

We modified the tools for our needs. The Front Panel test required a concession from the engineering team in the form of a special internal GPIB command to simulate front panel button pushes. The functional tests also used this internal command.

Once the MTBF numbers exceeded 12 hours, we needed extra PCs and prototypes for stress testing, since we needed our desktop prototypes for test development during the day. Several engineering team and marketing team members also ran stress tests. We usually ran 8 - 12 stress tests each weekend. Evaluation team members took turns monitoring, recording the results of, and restarting these weekend tests.

The evaluation group found 60% of its crash bugs using these stress tests.

Functional Testing

The evaluation team discovered many bugs while writing the functional tests. These tests were the hardest to implement for two reasons. First, we lacked an updated Functional Requirements document. It was not clear whether the engineers implemented a particular feature based on the engineer's specification, or their own interpretation during implementation. The evaluators submitted some "pilot error" bugs due to lack of understanding or conflicting prior product knowledge, which temporarily dampened the spirits of the engineers.

When the evaluators brought to management's attention the lack of, and ambiguity of the Functional Requirements, management assigned a full-time System Engineer responsible for resolving outstanding issues and providing documentation.

The second problem was internal to the evaluation team. We did not set adequate guidelines for test development, nor did we have sufficient design reviews of the tests we wrote. Six evaluators used six test writing styles. We corrected this problem by re-writing and consolidating many of the tests after the first two products were introduced.

Since the evaluation team was formed after budgets had been finalized, we had no money budgeted for test equipment or workstations. We borrowed a variety of equipment for test development, and this impacted how we wrote the functional tests, and who wrote which test. This became especially apparent when we began cross-checking the tests by rotating test-running responsibility around the evaluation team. The lack of workstations limited our ability to access and generate shared on-line documentation.

The original testing tool required unique tests for each product family, mainly due to front panel differences. In limited cases, functionality and limit values were also different. A later modification to the testing tool removed the requirement for separate tests due to front panel differences, and allowed us to have 60 - 75% common tests. This greatly improved the cross-checking process, and improved productivity.

Expert User Evaluation

About three weeks before introduction, 30 in-house expert users signed up for one-hour blocks of time to evaluate specific areas of instrument functionality. In addition, several managers were required to review sections of the manual.

Upon completion, we sent each user a summary of bugs, along with a keychain or mug as an acknowledgment gift. The expert users submitted about 4% of the bugs. At least two bugs were unlikely to have been found using the other evaluation methods.

Improve Group Credibility

Our credibility improved as we consistently submitted non-trivial bugs. We found that the engineering team paid attention to the bugs that were effectively managed. This meant the bug severity was accurate, there were no duplicates, and the bug description included a succinct, reliable way to repeat the bug. To this end, we paid close attention to reviewing bugs before submitting them, especially those submitted by the less experienced evaluators. The engineering team on this project was under extreme pressure, already working 60+ hours per week. We needed to maintain our image as a team who was furthering, not putting up barriers to accomplishing, the goal of completing the code and releasing a quality product.

The Evaluation Team Manager chaired a twice weekly meeting to discuss the latest bugs. Meeting attendees included: a Program Manager, hardware representative, software representative, evaluation representative, Software Manager, and marketing representative. Each bug was either assigned, declared a duplicate, postponed, or sent back to the submitter for more information. The severity of the bug was confirmed, and corrected if necessary. An evaluation team member entered any changes resulting from the meeting into the bug tracking system immediately following the meeting.

It was important to keep the severity level of a bug separate from the priority to fix that bug. The severity level reflected the state in which the bug left the instrument: (1) crashed, (2) lost data, (3) workaround exists, (4) nuisance bug. The priority to fix the bug related to how a customer would react to the bug, and how likely a customer was to encounter it. The Engineering Manager assigned the priority to fix and managed his team accordingly.

Especially during the last two months, the postponed state was used frequently. Care should be taken not to postpone bugs simply to avoid fixing them.

Communication to Management and the Engineering Team

Part of the trick of communicating with management is knowing their language. This includes agreeing upon a concrete set of metrics that evaluation and management feel communicate the necessary information. It is imperative that management understand the data that is being presented.

Corporate Targets

Five hardware quality metrics and one software quality metric are reported to upper management each month. The formula used to calculate the software quality metric is:

Software Defects Per 1000 Lines of Code [includes level 1, 2, 3 bugs] * Units shipped

The most important thing about the software metric is the trend, which, for the above formula, should be declining over time. The exact value is not important. The steadily increasing quality of the products shipped to our customers at introduction is what is important.

The metric is calculated from the point of view of the customer. If a bug was fixed, but not shipped, the bug is a problem in the customer's eye and is counted in the metric. Similarly, if a bug was introduced during an enhancement cycle but has not yet been seen by a customer, it does not count.

We initially used the bug graphs which DDTS produced to inform people of product status. However the graphs, which should have drawn attention from management, invoked no reaction. This was not due to a lack of concern about quality, but rather due to the presentation of the data. Figure 1-1 is an example of one of three graphs which were produced nightly which shows the number of open bugs. The other two show the new bug and resolved bug numbers.

When we wrote a special program to combine the data from the three graphs into one graph, as in Figure 1-2 , the Program Manager began managing the project to reduce the number of unresolved bugs. Same data, different format. Hard to believe, but easy to change.

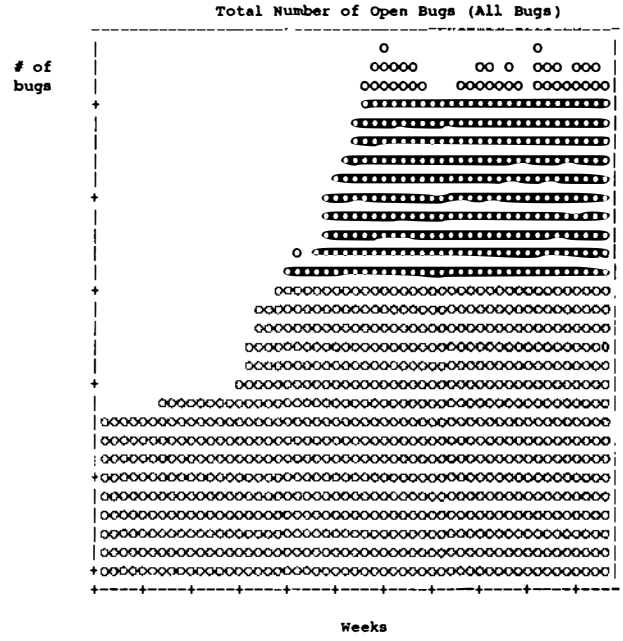


Figure 1-1: Unresolved Bugs as Displayed by the Bug Tracking System, DDTS

Although managers theoretically had access to the bug tracking system, few actually used it; it was another interface to learn. To invoke action, the Evaluation Team Manager began delivering hand-carried copies of the bug graphs each week to: the Manufacturing Manager, Hardware Manager, Software Manager, Program Manager, Product Manager, Software Project Leaders, and members of the Quality Group. The important thing is ***HAND-CARRIED***. This gave him a chance to converse with each person and fill them in on anything they did not know, or correct any incorrect perceptions **immediately**. This communication was essential to insure that management made informed decisions. And, after all, management's decisions affect all of us.

This communication method was so successful, when the Evaluation Team Manager was late delivering the weekly bug graphs due to other obligations, people would phone him looking for the information.

The Evaluation Team Manager also shared information with groups outside the organization, like TekLabs, to keep them abreast of the progress made on the project. One way to dispel rumors about a project, especially a project that has a long development cycle, is to communicate the truth. It is essential to speak about the successes of the project, even if they are small, and certainly not always of the shortfalls.

The Power of Words

You may not always realize the effect of your words on others. Just think back on how some of the things your parents told you as children still replay in your mind today. Or worse yet, how you now say them to your children. That same power exists when managers speak to engineers. For example, in a group meeting, the Program Manager said "Don't fix any bugs right now, just work on completing the feature implementation". He was surprised when we showed him the effect of his statement on the February and March bug fix numbers. (Figure 1-2) The engineers listened, and the number of unresolved bugs went up unchecked. You can also see where the focus turned to fixing bugs, in the middle of March.

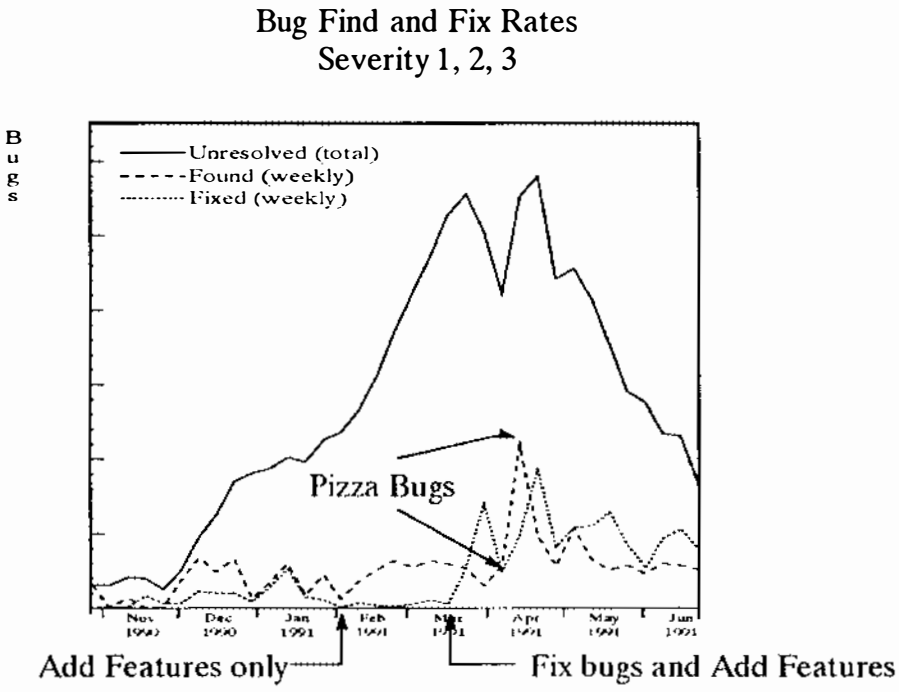


Figure 1-2: The Power of Words

Back Up Your Gut Feelings

When your full time job is evaluating a product or set of products, and you are intimately familiar with the workings and quirks of those products, chances are very good that your "gut feel" about the readiness of that product for customer release is accurate. However, managers do not make a decision about when to release a product in the marketplace by "gut feel" - especially not somebody else's. You need facts.

Stack the facts in your favor. For example, take the case of "pizza bugs". The Evaluation Team Manager, in an effort to get bugs recorded on the bug tracking system and into the hands of the engineers who could fix them, offered the team a challenge. He offered to treat the entire team of seven to pizza lunch if we could find a specific non–trivial number of bugs in one week. To make the progress more obvious, a Lotus spreadsheet depicted our progress on a ***PIE CHART***. (Figure 1-3) How appropriate! Of course, as you could guess, we reached our goal that week. [2] (Figure 1-2)

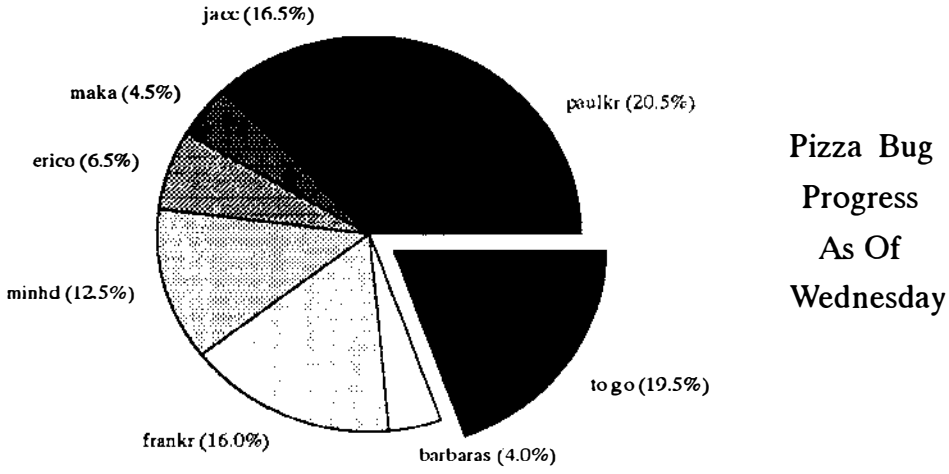


Figure 1-3: Pizza Bugs

This challenge had several effects:

- Showed how easy it was to find bugs in the system.
- Fostered teamwork amongst the evaluators.
- Made the bugs visible to the engineers that needed to fix them.
- Got us a free meal!

This caused management to postpone their estimated product introduction date.

Getting to Root Cause

The last six months of the project, there was a lot of pressure from management to complete the software and ship the product. As such, bugs were sometimes not welcomed by the software engineers. Especially when the description read "ran stress with seed 13999 and it crashed after 49 hours". Since the engineers needed their instruments for software development, and did not have 49 hours to run a test, they left these bugs for later. Often, with analysis, the evaluator was able to reproduce the bug by hand, or by paring down the stress test command log.

[2] Some bugs turned out to be duplicates and do not show up on the graph in Figure 1-2 .

One case in point involved a "loose cannon in memory" bug. This bug erased a sync bit during a stress test, causing a blank display while the scope merrily processed random GPIB commands. The engineering team could not do any useful debugging without knowing exactly what sequence of commands caused the blank screen. The evaluation team could not compile that sequence of commands without "catching the bug in the act", and then stopping the test by hand. This manual abort, done by typing "q" on the PC controller keyboard during the test, would stop the test and produce a command log. We built a small Lego crane attached to a light sensor, which stopped the test as soon as the screen went blank. By adjusting the beginning stress test seed, and reliably stopping the test when the screen went blank, we compiled an eleven-minute command file to recreate the bug. The engineering team then used a Tektronix DAS9200 logic analyzer to discover the root cause of the problem.

The flood of releases from engineering was a deterrent to finding root cause. By the time we submitted a bug with particular symptoms, the code no longer displayed the symptoms. This, and the fact that we preferred to run the entire regression test suite on each release, dictated when we began evaluating a new release.

OK, I'll Prove It!

Very, very rarely (only twice) did we find a bug that was isolated to a particular instrument. However, the instrument or person submitting the bug were always suspect, especially when the bug was non-reproducible. To alleviate these variables, we used multiple test stations, rotated instruments, and re-ran tests. When multiple instruments failed, we were in luck! We were especially lucky if the instrument at the engineer's desk showed the bug.

Depicting Progress

Management needs to see the progress of the project and the contribution made by the evaluation team towards attaining the project goals. Although the number of unresolved bugs climbs occasionally, in general, the number should decline as you approach product ship date. Although a new code release often causes a noticeable drop in the stress test run times, in general, there should be an increase in those run times as the product ship date approaches.

The evaluation team used another measure to depict progress. That measure was the percentage of functional tests run that "passed" using a given software release. We never successfully developed a way to present this data to management. Even after release of the first two products, the software project leaders did not know the extent of functional testing performed by the evaluation team had - they grossly underestimated the amount. Better data collection on the bug tracking system could have helped solve this problem.

One useful metric we could not provide was a code coverage metric - what percentage of code was executed by our test suite. This was because we used embedded software that did not provide access to this information.

We invited upper management to see our automated tests run, which looked impressive since so much activity was visible on the instrument display during a test. We also kept them informed with papers on testing innovations, such as a description of changes to the testing tools.

Team Management

How To Motivate Your Team

In general, people like to be part of a team and feel they are contributing to the success of the business. They want a challenging and interesting work environment, with the opportunity to be successful and learn. Fostering a team environment was a conscious decision, not an accident.

Each senior evaluator was responsible for at least one product evaluation effort. Particular interns worked primarily with a particular evaluator. However, when required, we could easily shift resources as priorities shifted. If a code release came out on Friday afternoon, the responsible evaluator would not be the only one here on the weekend testing it. The entire team would participate in the testing, and either join in on the weekend, or delve into it the next week. We promoted the idea of planning for releases, rather than operating in the constant "fire fighting" mode.

No one dictated decisions to others. Minimally, the senior members of the team needed to agree. When a team member wanted to change an initialization routine used by the functional tests, other team members were consulted first. This seemed like a barrier to getting things done at times, but it fostered a sense of team and kept key people informed.

The Evaluation Team Manager was excellent at creating a future for the group. He paid attention to our chosen career direction, and the contribution we each wanted to make. He discussed future products he expected the group would evaluate, and asked for input on what we would like to do. We could seek to provide our services to any product group in the division.

We set our goals at the beginning of each quarter by writing for ourselves the performance review we wanted to receive at the end of the quarter, including accomplishments and future direction. We could then refer to this document to determine our priorities.

After the initial products were released, the team set some new goals:

- Provide full evaluation for major new platforms. Once the product is released, turn the regression test suite over to the engineering team to run on incremental releases.
- Consult on incremental or small projects.
- Improve productivity by 50%, enough to continue avoiding extended periods of overtime.
- Port as many tests as possible to workstations, and automate the recording and presentation of Pass/Fail data.
- Standardize functional tests, and extract header information to show management what these tests do.
- Provide testing tools to other parts of the company, and be recognized as a center of evaluation excellence.

What We Improved

Once the first project was completed, we immediately directed ourselves to improving the test suite. We recognized the need for a configuration management system for the tests. The tests were managed using the source code control system RCS. An automated tool for placing tests into a testing directory was created. Scripts were developed for running tests based on categories such as test equipment needed or sub-system being tested, and we modified the functional test tool to allow shared tests between instruments.

The result of this work was that engineering could run the regression test suite before releasing code to evaluation and fix more bugs before we received the code. This facilitated more cross-checking of the tests, and allowed engineering to evaluate ***our*** code for a change!

Add New Product Responsibility

Once we cut down the time we spent developing original tests by using common tests, we began to branch out to cover more products. We informed other product managers that we wanted to help with their evaluation, and evaluators began rotations into other product groups. This meant that the evaluator would physically move to the new product area and become part of that product team. The evaluator's administrative manager was still the Evaluation Team Manager. This is the point at which we began evaluating instruments other than oscilloscopes.

Increase Productivity

One of the parameters by which the group was judged was attaining an annual 2x productivity improvement. We achieved this using automation and standardization. We adopted a standard format for graphically specifying system interaction and then describing the interactions and side effects in words. We used Interleaf, the common documentation tool used at Tektronix, and created our documents in a directory accessible to engineers in various parts of the division. This allowed us to share templates and documents easily with team members, and view other product documentation easily from our workstations. The Evaluation Plan has already been used successfully by a non-scope product in this manner.

Another area for potential productivity improvement is to free ourselves from history. We began redesigning tests based on the equipment needed to test the functionality, not the equipment we could borrow. Also, several tests originally required operator intervention - someone to verify that the display was correct. Additional functionality added to the testing tool to acquire checksums allowed us to automate these tests. The tests which originally took three to four people almost a week to run can now be done by one person in two days. Of course, the stress test still needs an elapsed time of 80 hours.

Although this redesign requires "rework", it is essential for generalizing the tests and making them consistent. Then, we can pass them off to the development engineers to be run as part of their evaluation suite, and direct our attention to new products.

Measures of Success

The evaluation team feels nervous when:

- The stress test times plummet as new functionality is introduced.
- The number of functional test failures go up with a new release instead of down.
- The total number of unresolved bugs is not decreasing. (Bugs should increase with each new release, but get fixed so that the general trend is fewer bugs.)

The evaluation team is done testing when:

- The product code meets the criteria stated in the evaluation plan, such as 80 hours stress test with no failure, and all functional and compliance tests pass.
- The engineering team stops fixing the bugs the evaluation team is submitting.
- The engineering team no longer provides you with the prototypes you need to do the testing.

These last two points were significant to the group. If you have been getting cooperation all along, and lately it seems that getting a prototype is like pulling teeth, you may be getting a message from the engineering team. Go find a group that wants your services and provide services to that new group. Evaluation is not the team that decides when to ship the product, that is the job of the Program Manager. Evaluation is not the policing organization, it is the informing organization. Present facts, not emotion, and let the people responsible and accountable for the decisions make those decisions.

The evaluation team feels they have been successful when:

- Management and engineering says evaluation has done a good job.
- The number of bugs reported by customers in the first six months after introduction is low.
- Other product groups call on them for evaluation services.

Summary

The most important factor in being an effective evaluator is establishing credibility. You can do this by being proactive, applying your experience, and communicating with others. Actions include:

- Attend engineering meetings.
- Study and learn the product specifications.
- Develop a specific evaluation plan and have management sign off on it.
- Inform engineers by submitting bugs.
- Inform management by presenting metrics.
- Think and speak positively of the product team and the product.
- Create a future for yourself in your job.

The most important thing upper management can do to insure a successful evaluation team is to treat them as partners. Hire senior people, provide equipment, include evaluation team members in meetings, and inform them of decisions. Promote early evaluation involvement, beginning with the design phase. Pay attention to the bug and MTBF data provided.

The first-level Evaluation Team Manager must be a communicator, a listener, and a visionary. Inform management and others about the work the evaluation team is doing. Promote reuse of the tools and tests. Listen to the roadblocks experienced by members of the team, and work towards solutions. A sense of humor helps too!

The evaluation team can make a recognizable contribution in attaining your product quality goals. Just establish a high level of expectation, and then let people live up to it.

Selection Criteria For A Software Test Automation Tool

Barton Weaver
Rajesh Jhunjhunwala
Technical Solutions, Inc.
16199 NW Jocselyn St.
Beaverton, Or. 97006

Abstract

Historically, testing a software product has been bogged down with repetitive, ad hoc manual testing, leaving little time for "strategic activities" such as: test architecture, coverage analysis, test data generation, metrics, performance measurements etc. In recent years, a surge of test automation tools have become available for evaluation engineers. If selected and used correctly, these tools can eventually automate much of the repetitive manual testing, allowing more time for "strategic activities". Testing can be continually expanded and enhanced by building upon automated test suites. In addition, the evaluation engineer's job is transformed from repetitive manual labor to evaluation engineering.

This paper discusses the criteria for selecting the right tool(s) for the testing task at hand including: general factors, actual commercial tool features, and implementation issues. It is intended for managers considering automated testing tools.

Biography

Barton J. Weaver received his BS in Computer Science from Brigham Young University in 1983. He worked for Sperry Computer Systems and National Semiconductor developing Systems software in UNIX. In 1987 he co-founded Technical Solutions, Inc. (TSI) and currently serves as the Chief Executive Officer. TSI provides software development and evaluation engineers on a contract basis to the high technology industry. TSI has a focus on software testing and evaluation.

Rajesh K. Jhunjhunwala received his MS in Computer Science, in 1979, from Birla Institute of Technology and Science, India, one of the top universities in India. He has worked for DCM Data products, Tandy and National Semiconductor developing systems software including operating systems and utilities with particular interest in evaluation issues. As a co-founder of Technical Solutions, Inc. he concentrates on managing the software quality assurance business. He has managed a number of software compatibility testing and evaluation projects. He is currently focusing on bringing software test automation to TSI's clients.

1. Introduction

1.1. Benefits of Automation

There are many benefits to automating software testing. Automation of software testing via a testing tool can provide better management and control over an often error prone process. Manually banging on the product, even with detailed written instructions, is extremely error prone. With batch file testing, the results are often determined by using file comparison utilities similar to 'diff' of UNIX or visually viewing a log file. This is also error prone and not applicable to testing GUI applications. Automation tools provide consistent and repeatable testing with greater precision and flexibility when comparing results.

Better management control over the testing process is achieved with automated testing. Exactly what is tested and how can be examined by looking at the test design and scripts. The scripts can be examined in a "test script review", turning intangible, unmanageable and error prone testing into a very precise, manageable and tangible process. Some tools aid in designing and managing the testing process. Once the process for a given tool is learned, the knowledge can be carried on to the next project.

Other benefits include: testing for previously fixed bugs, expandable test base, and tests for developers. With large software projects, bugs that have been fixed sometimes crop up again in the development cycle. With automated testing, test cases for each bug found can be put into the test suite. If a bug reoccurs, the test suite will spot it.

After a good base of automated test cases is formed, the evaluation engineer can expand and enhance the testing. The base test suites will run quickly and mostly unattended. Thus, the evaluation engineer doesn't have to completely stop test development to test the latest build of the software.

Another potential benefit, is the ability to provide an "acceptance test" for the developers. If the test tool is made available to a developer and a short test suite (the acceptance test) is created, the developer can run the acceptance test after making any changes to the software. If the test fails then the developer knows right away that the changes adversely affected the product. Furthermore, by viewing the test results or by single stepping through the test, the acceptance test can help isolate the problem.

1.2. Pitfalls in Automation

Some assume that automation will save the day and turn a testing crisis around, even if implemented at the last minute. This is a bad assumption. Automation of testing needs to be planned out. Throwing an automation tool at the evaluation engineer at the last minute will create more problems than solutions. It would be similar to throwing an application generator at a project that is 70% done and behind schedule. The ideal time

to start the automation process is at the start of product design. As the product is designed, the evaluation engineer can work with the design team and start designing the testing process. At this point, the evaluation engineer can determine if automation is appropriate. If automation is appropriate, then where and how it fits in can be determined.

Each automation tool has different characteristics that may or may not make it appropriate for the testing task at hand. The remainder of this paper discusses various issues that could affect the applicability of a tool for a project.

2. General features to consider when selecting a tool

2.1. Execution environment

Obviously, when selecting a software automation tool, the first question is: will it work with the environment that the software runs in? If the environment is UNIX, Windows, OS/2 or a mainframe, will the tool be able to test a product in that environment? In addition to these questions, the issue of multiple environments needs to be addressed. Is the software going to run and be tested in different environments? Will it in the future? It would be nice if the selected tool could be used to test the product in any environment. It would be even better if one test suite could be used for all the environments.

To understand how automation tools affect the environment we must understand how they interact with it. Different tools interact with their environment with different degrees of "tightness" depending on how integrated they are with their environment. We call it the tool's *coupling* with its environment. We will define three terms related to coupling: *loosely coupled*, *tightly coupled* and *de-coupled* tools.

2.1.1. Loosely coupled tools

Loosely coupled tools are software tools that interact with the environment of the Software under test (SUT) but are not assisted by it. They work only MS DOS, as Terminate and Stay Resident (TSR) programs. With no multitasking support, these tools use up part of the environment (i.e. memory, keyboard and screen) in an intrusive manner, leaving less resources available for the SUT. Additionally they alter the path of user input in order to be able to capture it. For example, Ghost and CAST run as TSRs under DOS and the SUT also runs under DOS.

A disadvantage with loosely coupled tools is that since they operate in the same environment as the software being tested, they are intrusive. An intrusive tool may alter the way the software functions. If the amount of memory available to a SUT affects its speed or some other factor, then since the tool uses up some of the memory,

it will change the SUT's behavior. If the tool uses some of the keys for its own use then a SUT using those keys will not be able to get them through the tool.

2.1.2. Tightly coupled tools

Tightly coupled tools are software tools that interact with the environment and are assisted by the environment. These tools are specifically designed for and integrated with the SUT's environment. For example, MS-Test runs under MS-Windows and is specifically designed to test MS-Windows Applications. X-Runner executes in the X-Window environment and is specifically designed to test X-Applications. Tightly coupled tools work only when testing software that runs in the same environment.

Tightly coupled tools differ from loosely coupled tools in the way they affect the SUT's environment. Whereas, loosely coupled tools consume memory and keystrokes from the SUT, tightly coupled tools run as just another application in the environment. The environment manages the memory and who gets keystrokes. Therefore the SUT is less affected by the test tool's presence.

A disadvantage with tightly and loosely coupled tools is that they are limited to testing software in one environment. But, this disadvantage creates the "tightly coupled" tools' greatest advantage. They are optimized for testing in their specific environment.

2.1.3. De-coupled tools

De-coupled tools are hardware assisted tools that do not interact with the software environment directly. These tools run on one host system and connect to another system (the test system) running the SUT. The Elverex Evaluator uses a custom video board in the system with the SUT to connect to the host. Another method, used by Ferret, involves 'Y' cables for the mouse, keyboard and RGB video output to connect to the SUT. De-coupled tools allow greater flexibility in the SUT environment. The tool interfaces to the hardware; therefore the operating system, amount of memory or type of CPU doesn't affect the operation of the tool. The following figure 1 show the configuration for a de-coupled tool.

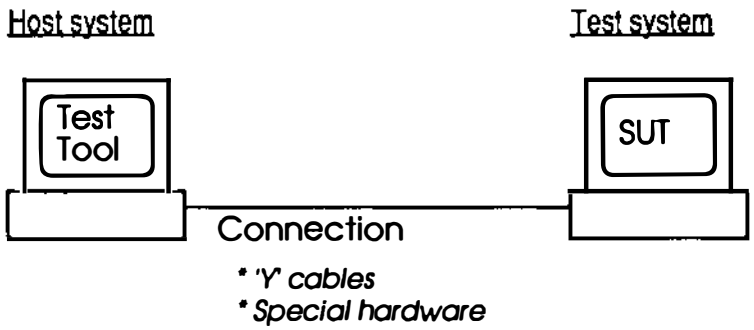


Figure 1: *System configuration for a de-coupled tool*

De-coupled tools can test software in multiple environments. If the software to be tested comes in both DOS and UNIX versions, then a system that can run both DOS and UNIX can be connected to the host system with the test tool. The de-coupled tools are also less intrusive since they do not run in the same environment as the software being tested.

The biggest disadvantage is that de-coupled tools are **not** optimized for a specific environment. Thus, more detail is needed in the test scripts to accomplish each task. Also, there are often problems with synchronization with the SUT. This issue is discussed later.

Even though de-coupled tools offer the most flexibility in general, your specific application may benefit more from another type of tool. Tightly coupled tools like SQA-Robot and Autotester for Windows are focused on testing MS-Windows applications. These tools are aware of window elements. They know what buttons, messages, dialogs etc. are. More importantly they know how to interact with each element in the environment. Because of this, tightly coupled tools have more direct and straight forward access to the software being tested. For instance, consider the task of accessing the **file/open** menu item. On a de-coupled tool, the test developer would have to issue numerous mouse movements and clicks using exact coordinates. For the same task the "tightly coupled" tool needs only one or two commands such as: WMenu("&file") and WMenu("&open") to select the same menu item.

2.2. Capture/Playback

Most tools have the ability to capture or record input to the software under test. Tools can capture keystrokes and mouse events. The tool then provides a method to playback the keystrokes or mouse event to the software under test. How this is done varies from tool to tool. SQA-Robot, MS-Test and Autotester for Windows capture and playback window messages. The Evaluator, Ferret and Test-Runner use 'Y' cables for the keyboard and mouse. For many of the tools, the time delays between inputs is also recorded.

Unfortunately, capturing and playing back inputs tends to be an unstable and awkward method for developing a test suite. The ability to accurately playback the scripts may be affected if the system or environment of the SUT is changed. In some cases simple things such as longer access time across the network can de-rail a playback session. Also, if viewed, captured scripts tend to be confusing. The action of moving a mouse to the file menu may result in 50+ mouse movement commands in the script. The captured scripts often contain extra information not intended to be captured. For instance, if you sneeze while your hand is on the mouse, 50+/- mouse movements will be captured.

Because of these problems, it is not wise to rely **just** on capture/playback to develop a comprehensive test suite. If tools allow the captured scripts to be modified and called

by the tools script language then they can be used as building blocks. For instance, Test-Runner scripts look like 'C' code. A captured script can be encapsulated as a callable function.

2.3. Scripting Language

The scripting language of the tool offers the features of a high level language: modularity, control constructs, error handling etc. With the scripting language, the evaluation engineer can design the test suite in a modular manner. Often used test sequences can be grouped together and included or called when needed. Some scripting languages allow captured scripts to be called or incorporated into the script. In addition to modularity, control constructs such as, if-then-else, while, and for loops add power and flexibility when developing tests.

Some tools use a standard language as their scripting language. MS-Test and SQA-Robot use different versions of BASIC for their scripting languages. X-Runner's scripting language is 'C' like. If a standard language is used then the evaluation engineer will not have to learn a special language just for the testing tool.

Whether a standard or custom language is used, the scripting language must have access to the SUT's output for comparison. This includes comparison of graphic items, underlying text and error conditions. It is important to be able to test for error conditions and be able to respond to them.

2.4. Test Errors

2.4.1. Error Logging

A typical test suite will have several test units in it. It is necessary to log the result and other information from each test unit before continuing to the next. A two level error logging system is useful not only in detecting errors but also establishing confidence in what works. First, a summary file logs which test unit was successful and which failed. This file simply logs the pass/fail condition of each test unit as it is run so that the result of the entire test suite can be determined at a glance by looking at this file. Secondly, a separate file clearly records details of errors, identifying the test script where the error occurred, the expected and actual results and any other information that will help the tester to reproduce the error.

2.4.2. Error Handling

To continue from the error condition, the test must be able to take some corrective actions before continuing. The severity of the errors determine the corrective actions. If the output of an expression evaluator is erroneous, the result can be recorded and no further action is necessary. However, if an input file was missing, then the rest of the

test unit that depended on the input file can not be performed. The entire test must be discarded. Sometimes the only corrective action might be to close the application and start over with the next script. Most scripting languages provide a feature to test for error conditions and branch to an error handler in case of errors. Some tools even provide a means to reset the entire test system, in case it is hung.

2.4.3. Analysis and Reporting

Unless an error analysis mechanism is present, the tester may spend most of the time in analyzing the results, since some applications can produce large amount of error logs. One simple mechanism is to log an identifying message for every error, that can then be searched for in the log files. Once an error is identified, it must be reported clearly so that it can be duplicated. In addition, some tools, CAST for example, interface with a Data Base Management System (DBMS). The test results are fed into the DBMS and reports can be generated with the DBMS report generator.

2.5. Ease of use

A tool must be easy to use for it to be used effectively. The more difficult the learning curve, the less likely it is for the full potential of the tool to be utilized. In evaluating the ease of use features it is important to look beyond the simple tasks. Most tools can perform simple capture/playback of keyboard and/or mouse events fairly easily, but testing will generally require functionality beyond capture/playback. The following are some of the points to consider for ease of use.

- Is it easy to get help? Are the manuals and on-line help effective? Some products have no manuals, only on-line help. You may or may not find that adequate.

- How long does it take to get started? Does the tool have a familiar interface or is it a new paradigm for you to get accustomed to? Some tools have unique interface concepts to represent the user input data. Some have confusing output formats with multiple windows overlapping each other.

- Does the tool provide powerful and flexible design options? Capture/playback is fine for simple tasks but, beyond a certain level of complexity, you will need a scripting capability. Test-runner and others provide the scripting capability combined with the capture/playback, so that a script is generated by the tool as you perform the capture. Ferret uses MS-Excel and Owl Hypertext to lay out the test design and link the design to the test scripts, test results and test matrices.

- Can you edit test scripts? It is very likely that the initial test scripts will have to be edited several times before the test script is finished. Do you have to recreate the test or does the tool provide a method of editing them? If the editing feature is present, how easy is it to use? Some tools like MS-Test provide custom script

editors with built-in aids for script development. Other tools allow you to use any text editor for creating scripts.

- Does the tool aid you in debugging the script? In the absence of debugging aids, the user is forced to cut the script at a suspected error point and run it. If no error is found then add more of the script and run it again and so on. Most tools provide a single step mode for controlled execution of test scripts for debugging. Other advanced debugging features such as diagnostic messages, breakpoints, stop execution, resume execution, etc. are very useful.

- Does the tool aid in running and managing test suites? As individual tests are built into complex test suites aids for handling the complexity become important. Some tools provide test selection, management and results analysis. On-Stage, for instance, is a graphical test manager with point and shoot and query test selection. Other tools have test managers that are sold separately.

2.6. Synchronization

Most modern applications are very interactive with an operator. The operator makes a request by providing an input and the application performs the requested task (which may take some time) and then informs the operator of the result. This repeats until the operator terminates the application. The operator automatically synchronizes with the application. This process is automated by having the tool provide the requests and look for the expected results. To simulate the real operator environment correctly, the tool must ideally be able to wait until the result of an operation is displayed before providing the next input. Some key considerations are:

- **Fast input:** A tool may provide input at a fixed time interval and therefore can get ahead of the application. This can cause some input to be lost. In most DOS programs where input is buffered, this may not be a problem. But some applications flush the buffer before receiving input into a form, causing loss of input data. To correct this problem the test must be run at less than full speed.

- **Quick test:** A tool may look for the result too soon, therefore reporting an erroneous result.

- **System changes:** Synchronization problems may occur due to hardware characteristics (network access time), unpredictability of processing time (length of a document being processed) or change in operating environment (position of the data file on a disk).

A good tool will provide the user with several ways to synchronize with the application.

- **Wait for a prompt string:** The next input is sent only after the specified prompt string appears on the screen. MS Test allows synchronization by waiting while the mouse cursor is the shape of the hour-glass.

- **Retry count:** If the expected result does not appear on the screen then the tool retries the match after a delay. The delay time and retry count can be set by the user.

- **Event queues:** In GUI-based tightly coupled tools, synchronization is done using event queues.

2.7. Intrusiveness

There is bound to be some intrusion by the tool into the SUT's environment. The severity of the intrusion must be evaluated and its affect on the SUT understood. If the test tool shares the keyboard with the SUT it will limit the key codes available to it. If the screen or memory is shared by the tool, the SUT will have less for its use. More serious intrusions alter the environment transparently, but may also affect the operation of the SUT. Some tools trap interrupts, altering timing and interrupt handling by the SUT. Some tools require special hardware, preventing testing on standard hardware.

Since the human is eliminated, the automation tool must perform the task of providing and observing the result of user inputs. This necessitates some form of intrusiveness. You must evaluate the kind of intrusiveness that will affect your testing. If you are testing a video driver, then clearly the tool that requires its own video board for the SUT is not for you.

2.8. Repeatability of Test

The key to automation is being able to repeat a test reliably. Each run of a test should be exactly the same as the next. The tool must not introduce any variations of its own into the test environment. Capture/playback scripts tend not to be identical for every test run, due to time delays in the captured scripts for synchronization. Over time, a file may take longer to access because of disk fragmentation.

2.9. Output Validation

Most interactive applications are either forms or window based. In response to user input, an application may display multiple data items. To test the results of any such input, one method might be to capture the window or screen. This could then be compared with the same screen or window previously recorded and certified correct. However, if part of the window displays the current date, then the comparison will fail every time, even though there is no error. A tool should provide alternatives to comparing the entire screen.

- User definable rectangles or "windows" can identify the part of the screen that contains the relevant information. The user must be able to define a reasonable number of such "windows".

- User definable exclusion fields in a window, the contents of which are excluded from window comparison.

- Often the result of a user request is in the form of a text string output on the screen. If the tool can extract the text string at a specified location, then it can make the task a lot easier.

- Some tightly coupled GUI tools can actually compare the internal representation of windows, dialog boxes and text strings which makes the job faster and simpler.

2.10. Screen Resolution

If your test system is upgraded to a higher resolution video, then screen comparisons are likely to fail. Automation tools can accommodate these video advances in a variety of ways. In earlier generation character graphics, ignoring color was the obvious solution. Pixel graphics complicate the task significantly. Some tools have a regenerate feature that will run the test script without doing any screen comparisons, but replacing the old screens with new screen information generated by this process. Some tools provide a FUZZY compare feature. This allows the user to specify tolerances for screen comparisons. With this feature, the high resolution screen image can be rounded off to match the lower resolution. In tightly coupled tools, since the tool can determine the internal representation of the screen information, the fuzzy comparison can be made intelligent and comparisons can be made based on screen coordinates instead of pixels. This relieves the user from trying to guess the tolerance.

2.11. Cost

Consider the total cost of test automation. This includes not only the cost of the automation tool but also the cost of learning the tools and generating a set of test suites. On the other hand, the benefits of the tools may more than offset this added cost. Some tools have a large price tag but come with a lot of the test suite development work already done and have small learning curves. Such a tool may have a smaller overall cost of automation than one that is low cost but require a lot of up front work in developing the test suites. Another factor in considering cost is the fact that some of the expensive tools have multi-user designs. The incremental cost of adding another user or test unit to such a tool is usually a lot less. This can result in a lower cost per user/test unit than the tool that is single user in design.

3. A summary of where each tool fits in the testing arena

No single tool can be declared the "best", like programming languages where no one language meets everyone's needs. No one test tool will meet everyone's needs as well. The tool needs to be selected with the specific testing requirements in mind. Each tool presents a different mix of features.

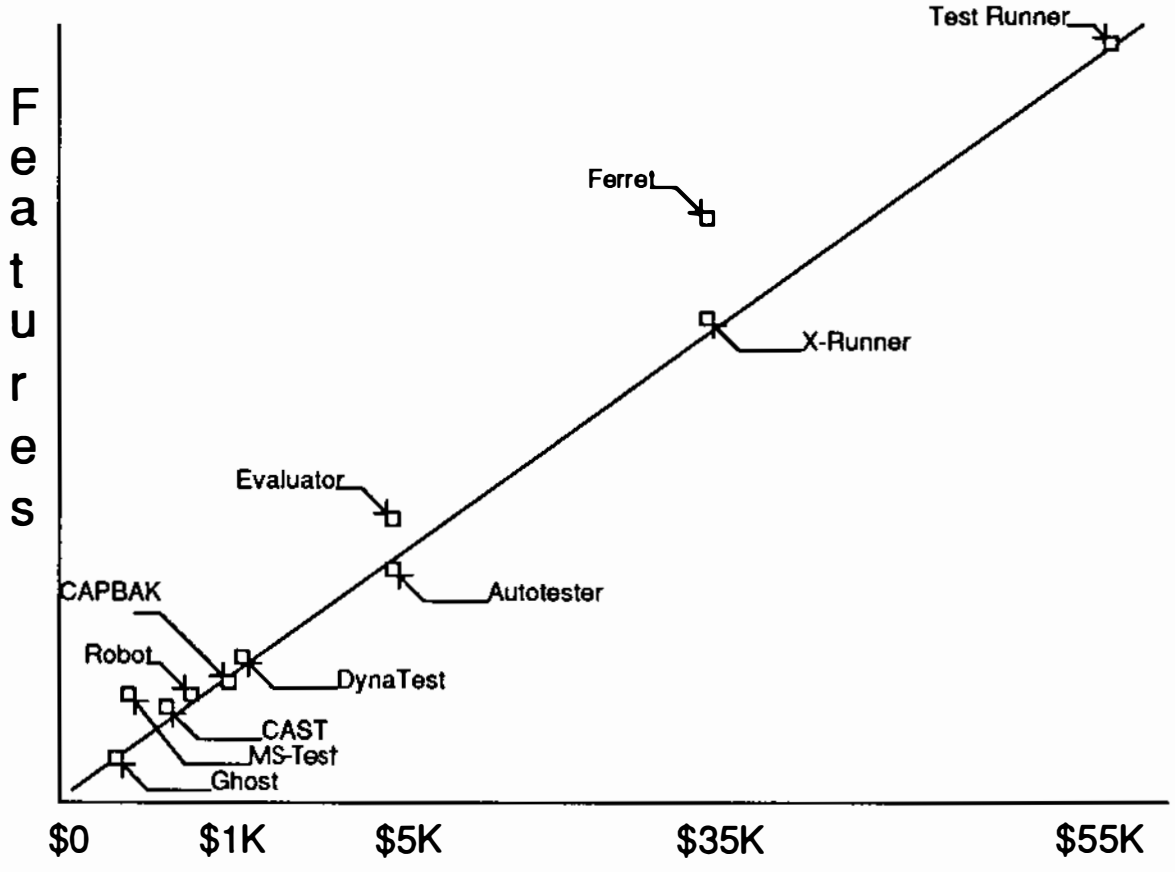


Figure 2: *Price - performance of some of the available tools*

Figure 2 shows a rough break down prices versus features provided for the tools mentioned in this paper. As would be expected, the more features in a tool, the more flexibility and the higher the price.

The table below displays the categories that the various tools fall into. All tool names are the tradmarks of their respective vendors.

Tools	Vendor
Loosely coupled tools:	
Ghost	Vermont Creative Software
CAST, On-Stage	InfoMentor
Autotester	Software Recording Corporation
Tightly coupled tools:	
Autotester for Windows	Software Recording Corporation
DynaTest for Windows	Innovative Business Systems, Inc.
MS-Test	Microsoft Corporation
XRunner	Mercury Interactive Corporation
Robot	Software Quality Automation
CAPBAK	Software Research
De-coupled tools:	
Autotester (in terminal emulation mode)	Software Recording Corporation
Ferret	Tiburon Systems, Inc.
TestRunner	Mercury Interactive Corporation
Evaluator	Eastern Systems, Inc.

4. Conclusion

When switching to automated software testing, it is important to do so with your eyes open. Automating software testing has its place and real benefits. There are also cases where it does not make sense to automate. In most cases, automating takes additional up front time, effort, and money. The real pay back comes over a products' life cycle. In a sense, automated testing is just for regression testing. Test suites can run unattended on stable features of the SUT. If the SUT changes features often during development, then the automation effort is going to be frustrating if not impossible. For instance, if the human interface changes from using menus to icons, then all of the test suites will have to be overhauled. Unfortunately, these type of changes happen even towards the end of product development. Then again, for products whose designs are more stable, automation makes more sense.

THE IMPACT OF REUSE ON SOFTWARE QUALITY AND PRODUCTIVITY AT THE HEWLETT-PACKARD COMPANY

Wayne C. Lim
Hewlett-Packard Company
Corporate Engineering
Software Reuse Program
1801 Page Mill Road 18DG
Palo Alto, CA 94304
415 857 2588

lim@hpcea.ce.hp.com

Abstract

Findings at the Hewlett-Packard company indicate that reuse has a significant impact on software development. Metrics describing the improved quality and productivity resulting from reuse will be shared.

Biography

Wayne C. Lim is a member of the Hewlett-Packard Corporate Software Reuse Program specializing in the economic, organizational and metric issues of software reuse. He had previously researched reuse library issues at Ford Aerospace and reuse managerial issues as part of his graduate studies at Harvard University.

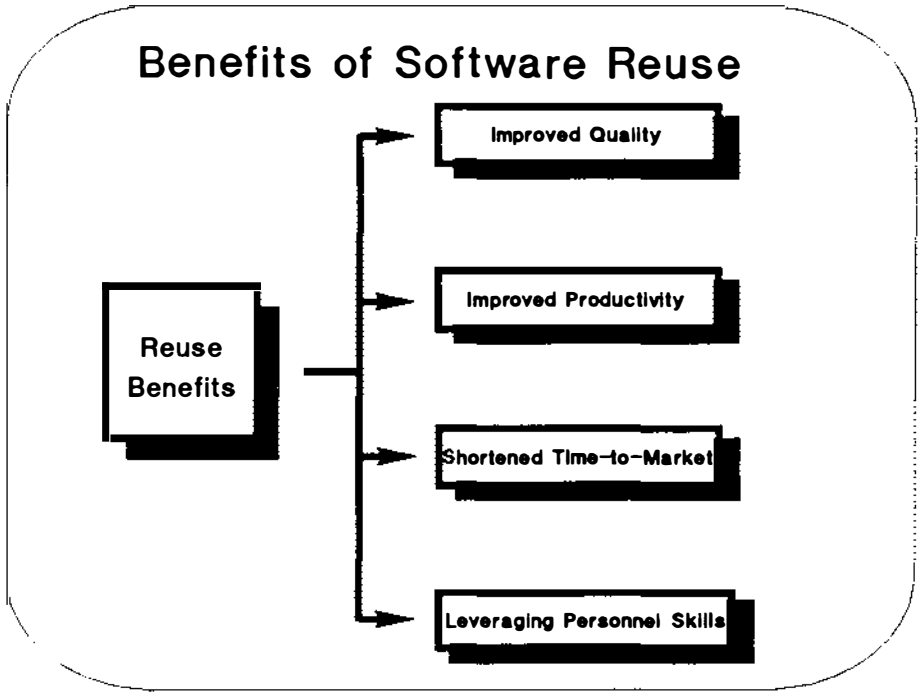
Introduction

Although not a new concept, software reuse as a means of improving software quality and productivity has only recently been aggressively pursued. Findings at the Hewlett-Packard (HP) Company indicate that reuse has a significant impact on software development. In this paper, we will present and discuss metrics from actual HP software projects documenting the improved quality and productivity resulting from reuse. We will include data from software reusers on a development project that indicate how they believe reuse has impacted the following software quality factors: Functionality, Correctness, Usability, Reliability, Performance, Supportability, Localizability, Flexibility, Integrity, Testability, Portability, and Interoperability. We will define each of these factors and discuss how reuse can be utilized to improve some of these software quality factors.

For the purposes of this paper, we define workproducts as products or by-products of the software development process. They include, for example, code, design and test plans. Reuse is the use of these workproducts without modification in the development of other software.

1. Benefits of Software Reuse

Among the major benefits of reusing software are improved quality, increased productivity, shortened time-to-market and more effective use of unique personnel skills (Figure 1).



pscnv/bene3.gel

Figure 1

Reusable workproducts are subjected to multiple uses and the accumulated defect fixes result in a higher quality workproduct. Increased productivity is achieved because the workproducts have already been created, tested and documented. Consequently, it now requires reusers less effort to generate the same output. Increased productivity from reuse, however, does not necessarily result in shortened-time-to-market. Shortened-time-to-market refers to the time it takes to deliver a product to market from time of conception. For example, when reuse does not occur

along the critical path of a software development project, increased productivity may result but not necessarily shortened-time-to-market. On the other hand, if reuse has been effectively utilized on the critical path of a project, time-to-market may be shortened. Finally, software reuse enables an organization to utilize their personnel more effectively because it leverages the expertise of experienced and specialized software engineers. These people can concentrate on creating workproducts which may then be reused by less experienced and specialized personnel.

Software reuse, however, does not come for free. Examples of reuse costs include the cost of creating and maintaining reusable workproducts, a reuse library (if necessary), and reuse tools. The results of an economic, or cost-benefit, analysis on reusable workproducts will be discussed later in this paper. We will first examine data collected at HP describing the improved quality and productivity from reuse.

2. Improvements in Quality and Productivity through Reuse

a. Findings on Improved Software Quality

Because workproducts undergo multiple uses, the defect fixes from each reuse accumulate, resulting in a higher quality workproduct. More importantly, software reuse provides incentives to prevent and remove defects earlier in the software development life cycle because the cost of prevention and debugging can be amortized over a greater number of uses [LUBA].

Historical data from Robert Grady's book, Software Metrics, and current data indicate that projects which employ reuse realize improved quality at HP. In Figure 2, Grady [GRAD] presents prerelease data indicating that defect densities (average defects per Thousand Lines of Non-Commented Source Statements (KNCSS)) for HP projects reusing code at levels greater than 75% were up to 40 percent better than those that did not. The defects shown are only those discovered during formal testing. The total number of projects of each type is indicated in the parentheses in Figure 2.

Data provided by Alvina Nishimoto [NISH] of the Manufacturing Productivity Operation (MPO) at HP indicates that the defect density rate for reused code is approximately 0.4 defects/ compared to 4.1 defects/KNCSS for new code. Use of reused code in conjunction with the new code (46% reuse) resulted in approximately 1.0 defect/KNCSS rate for the product, a 76% reduction in the defect density compared to new code (Figure 3).

Jack Cassidy of the San Diego Technical Graphics (STG) Division in HP has also reported a positive experience with reusable code. This division estimates a defect density rate for reused code of 0.4 defects/KNCSS and a rate of 1.7 defects/KNCSS for new code. A recently released product with approximately 30% reuse resulted in a defect density rate of about 1.3 defects/KNCSS (Figure 3).

b. Findings on Increased Software Productivity

Higher productivity is achieved through reuse because the software product life cycle now requires less input to obtain the same output. This may be accomplished through a number of ways. For example, reuse can reduce the labor cost input by encouraging personnel specialization (e.g. software engineers who specialize in user interfaces). Because of their experience, specialized personnel working on their particular aspect of development usually accomplish the task more efficiently than non-specialized personnel.

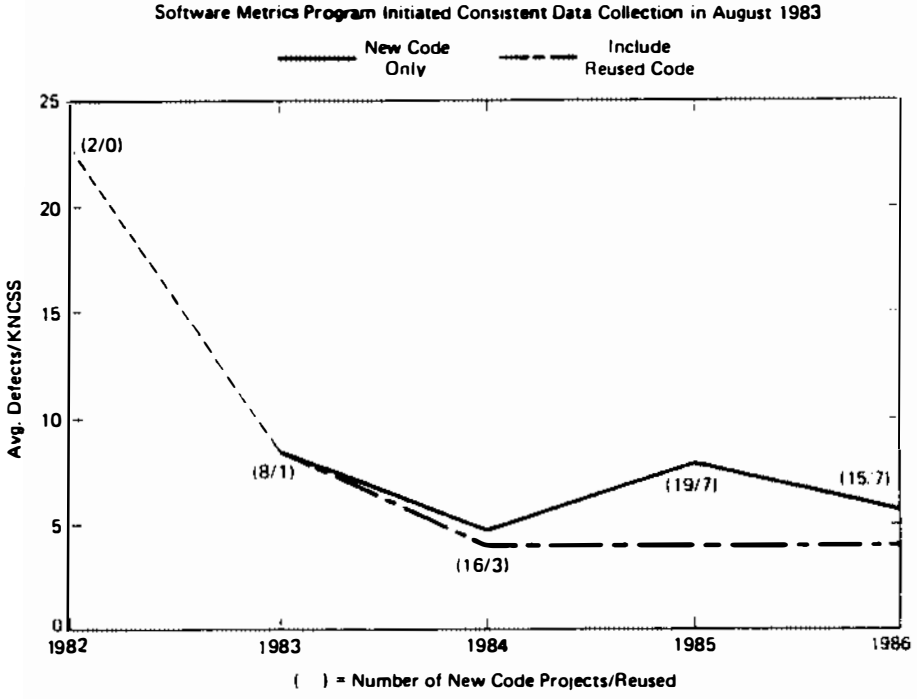
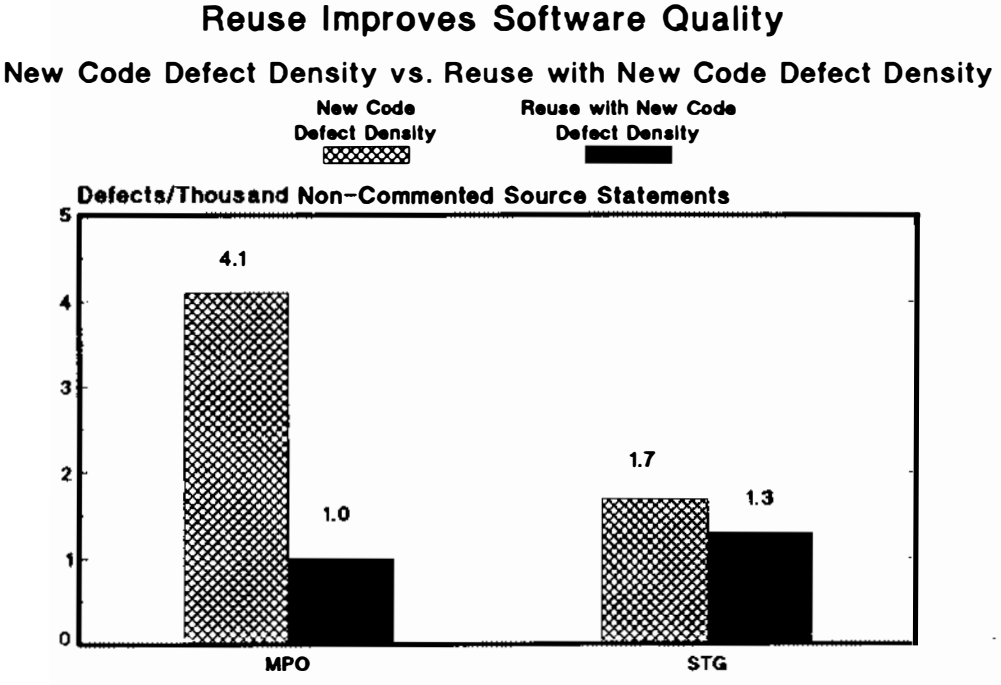


Figure 2

Figure borrowed with permission of the publisher from SOFTWARE METRICS: Establishing a Company-wide Program, Grady/Caswell, 1987.



3/pacnw/qual3.gal

Figure 3

Furthermore, productivity is increased simply because fewer software workproducts need to be created from scratch. For example, if reusable workproducts have already been documented and tested, these activities for the final product are reduced. Reuse can also improve the maintainability and reliability of the software product, thereby reducing labor input in the maintenance phase.

In general, software reuse improves productivity by reducing the amount of time and labor needed to develop and maintain a software product. To illustrate, Grady provides data on 93 projects indicating productivity gains of up to 33 percent when significant reuse (greater than 75%) is employed (Figure 4).

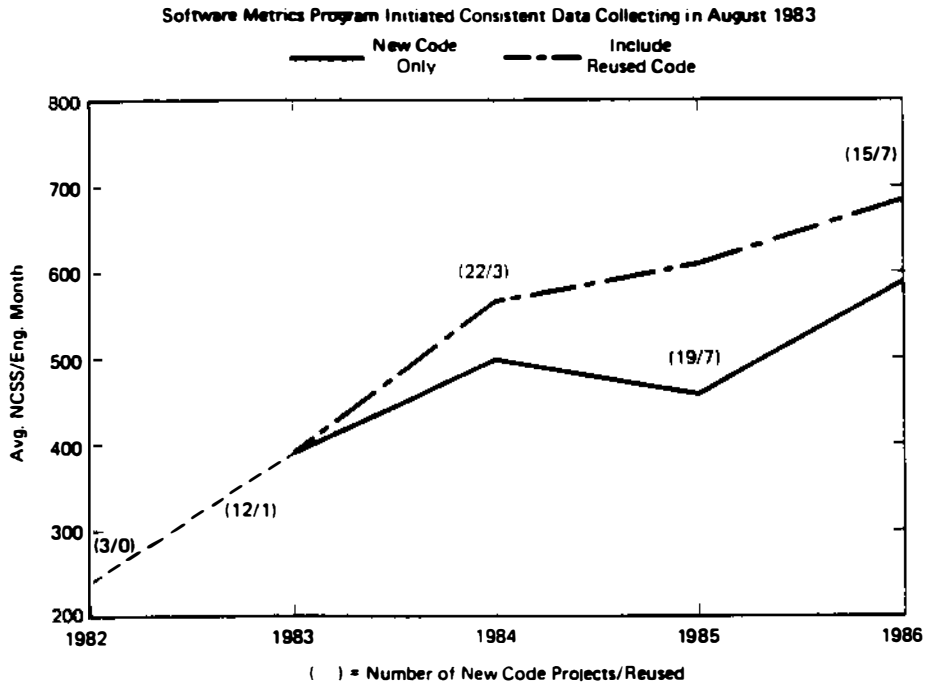


Figure 4

Figure borrowed with permission of the publisher from SOFTWARE METRICS: Establishing a Company-wide Program, Grady/Caswell, 1987.

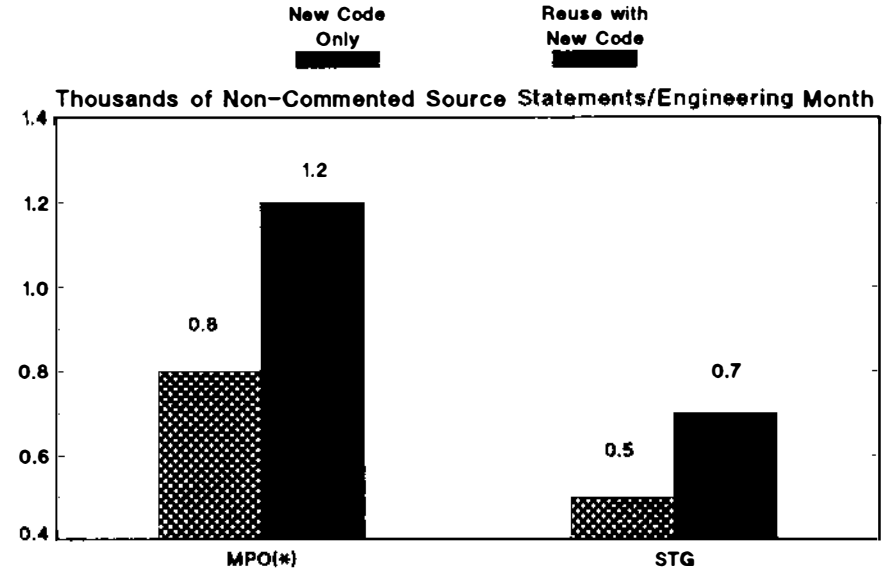
Using data that covers the investigation to test phases only, another project at MPO reported a productivity rate of .08 KNCSS/engineering month for new code. Use of reused code with the new code (38% reuse) resulted in a productivity rate of 1.2 KNCSS/engineering month, a 57% increase in productivity (Figure 5). The languages used were Pascal and SPL, the Systems Programming Language for the HP 3000 Computer System.

STG estimates a productivity rate of 0.5 KNCSS/ engineering month for new code ("C" programming language). A released product with 30% reuse resulted in a productivity rate of 0.7 KNCSS/engineering month, a 38% improvement in productivity (Figure 5).

A firmware division within HP has been tracking the reuse ratio to the productivity rates in the development of their products. As shown in Figure 6, by 1987 several products had already exceeded their 1990 productivity goal of 2 KNCSS/engineering month with greater than 70% reuse, as well as being above the projected productivity rates. It should be noted that the reuse ratio calculation (shown at the bottom of Figure 6) used in this division includes leveraged code as well.

Reuse Increases Software Productivity

New Code Productivity vs. Reuse with New Code Productivity

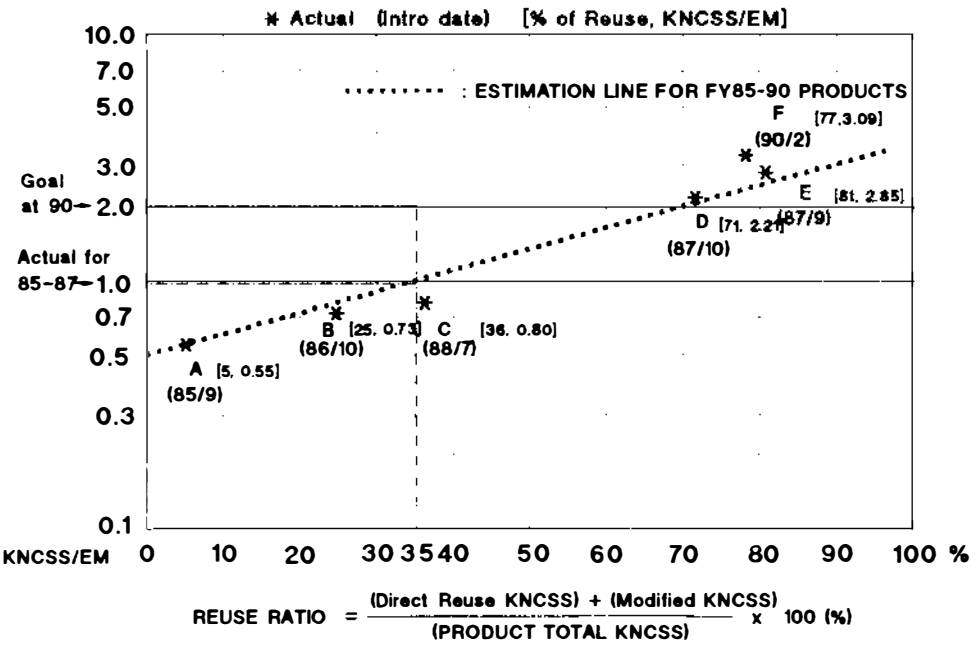


* Eng Months for Investigation to Test Phases only

3/pacnw/prod2 gal

Figure 5

FIRMWARE PRODUCTIVITY IN AN HP DIVISION



$$\text{REUSE RATIO} = \frac{\text{(Direct Reuse KNCSS) + (Modified KNCSS)}}{\text{(PRODUCT TOTAL KNCSS)}} \times 100\ (\%)$$

3/pacnw/fw1 gal

Figure 6

3. Economic Savings from Reuse

An important aspect of software reuse is the economic return that consumers and producers receive for their efforts. The method of economic analysis used is the well-established Net Present Value method. Net Present Value takes the estimated value of reuse benefits and subtracts from it associated costs, taking account the time value of money. This analysis was conducted for 15 reusable workproducts (Figure 7) and the results indicate the consumer savings from reuse of a workproduct ranged from four to 0.2 to 4.0 engineering days. The workproducts were written in Pascal or SPL (Systems Programming Language for the HP 3000 Computer System) and ranged in size from 58 to 2257 Non-Commented Source Statements.

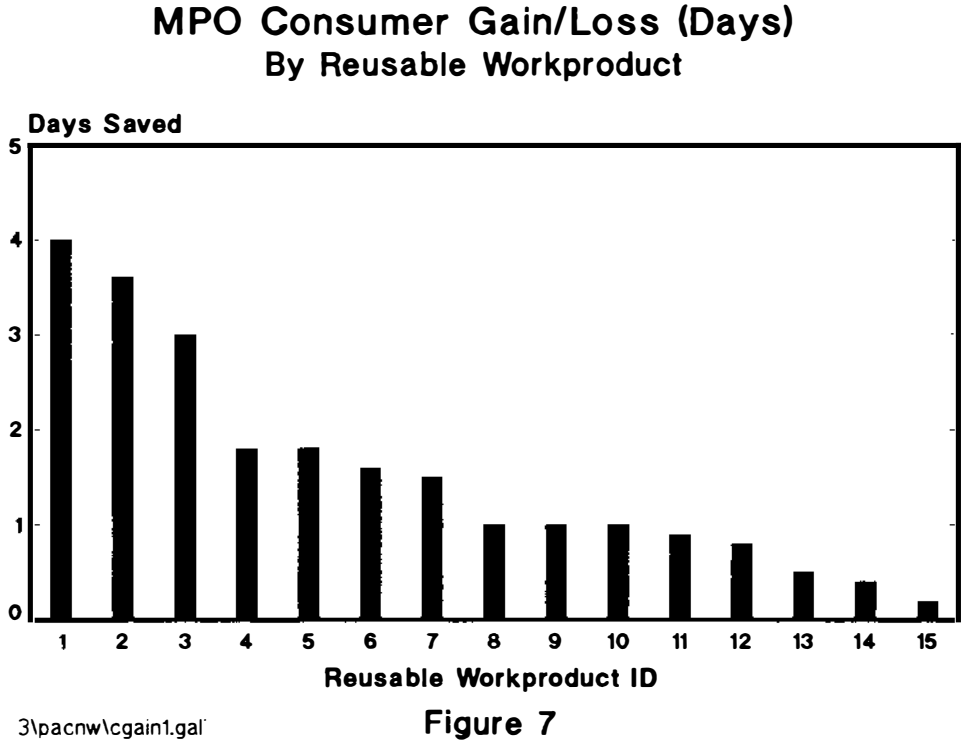


Figure 7

Likewise, an economic analysis from the perspective of the producer was done for the same reusable workproduct. Here the analysis attempts to answer the question: "Is it worthwhile for me as a producer to create this reusable workproduct?". This economic analysis was conducted for the same 15 reusable workproducts and is shown in Figure 8. The economic gain/loss ranged from a gain of 43.3 engineering days to a loss of 31.5 engineering days. The 31.5 engineering day loss was the result of fewer reuses and higher maintenance costs than expected.

4. Software Quality Factors

While we have shown that reuse improves quality by reducing the number of defects relative to the number of lines of code, we wanted to survey the users of reusable software on their perceptions of various software qualities and the impact that reuse has had upon them. Strictly objective measurements (e.g. Mean-Time-To-Failure for measuring reliability) of the software quality factors would have been optimal. However, we decided that at this stage of our efforts (and given the limited amount of resources) to survey users of the reusable code for their subjective perceptions. The qualitative surveys identify differences in perception among the producers (creators of the reusable workproducts) and consumers (users of the reusable

workproducts). The factors with the most significant differences can be studied further with the collection of quantitative information.

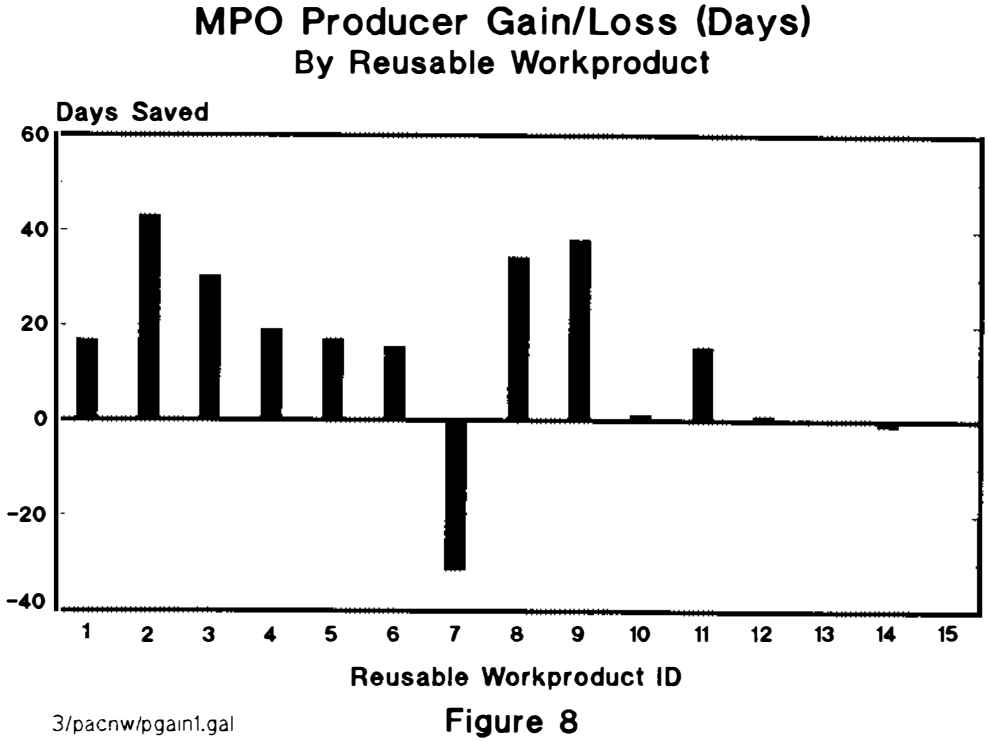


Figure 8

As expected, the perceived importance and impact of reuse varied across different consumer projects within different domains. We hope to examine the data for significant trends and commonality after assessing a number of these consumer projects.

After defining each software quality factor, we will present results from the analysis of one consumer project as an illustration.

a. Software Quality Factor Definitions

Software quality can be improved through reuse from the perspective of both the consumer (the user of reusable workproducts in the development of software products) and the end-user (customer of the software product). Figure 9 depicts the factors of software quality as adapted from McCall, et al [MCCAL]. These factors are not necessarily independent, i.e. the existence of a given factor may have a positive or negative effect on another factor.

SOFTWARE QUALITY FACTORS

1. Functionality
   Extent to which a program fulfills the end-user's mission objectives in terms of its feature set and capabilities.

2. Correctness
   Extent to which the program is free from design and coding defects.

3. Usability
   The friendliness and ease of use to learn, operate, prepare input, and interpret output of a program.

4. Reliability
   Extent to which a program can be expected to perform dependably and with the absence of failure.

5. Performance
   The efficient use of system resources; adequate response time

6. Supportability
   The ease in which the program is maintained; cost of ownership

7. Localizability
   Effort required to modify a program in order to meet the needs of the regional user.

8. Flexibility
   Effort required to modify an operational program.

9. Integrity
   Extent to which access to the program or data by unauthorized parties can be controlled.

10. Testability
    Effort required to test a program to ensure it performs its intended function.

11. Portability
    Effort required to transfer a program from one hardware configuration and/or environment to another.

12. Interoperability
    Effort required to couple one program to another.

Figure 9

b. How reuse can improve software quality factors

The following are examples of how reuse can help consumers enhance software qualities in the final end product for their end-users:

FUNCTIONALITY - Functionality may be improved because reused components can be used to quickly create prototypes. These prototypes help customers clarify their requirements so that the consumers can fulfill them.

CORRECTNESS - Correctness may be improved since the reusable workproducts have been tested and reused in a variety of situations.

USABILITY - Usability is also improved, for example, when a user interface is reused among several products. End-users who own several of these products will benefit from the consistency.

RELIABILITY - As Mitch Lubars [LUBA] of the Microelectronics and Computer Corporation observed, more reliable software is achieved through reuse because the cost of debugging can be amortized over a larger number of usages.

SUPPORTABILITY - Because reusability implies that software workproducts come with clear documentation so that a potential consumer can determine its applicability, ease of maintenance is improved as well.

FLEXIBILITY - Flexibility is enhanced by reuse, for example, when modifications to a system entail only changes to parameters on a reusable component or when easy integration of additional reusable components into the system adds functionality.

LOCALIZABILITY - Localizability may be improved when reusable workproducts are used interchangeably to provide the necessary functionality for a given region (e.g. unit of measure, regional tax rate).

INTEROPERABILITY - Reuse also contributes to interoperability. For example, when a component that contains an error-message handling routine is reused by several systems, these systems can expect standard data representations and protocols.

End-users may not be aware of how reuse improves software quality in the products they use. Rather they will see the results of increased reliability, maintainability, correctness, and flexibility offered by reuse in the form of less down time, and fewer and faster turnaround time on maintenance or enhancement requests. It should also be noted here that there are trade-offs between the software quality factors. One may choose to use reuse to emphasize one software quality factor over another.

In the next section, we present the perceptions of software reusers on one project on how reuse has impacted the SQFs.

5. Reuse Impact on Software Quality Factors (SQF)

a. Importance of SQF to Product/System

MPO consumers were asked to rate the importance of the software quality factors to the end-user of the product. In order to determine how "in tune" the producers were with the end-users, producers were also asked to rate how important the SQFs were to the end-user.

The scale and mean values for the consumers and producers are shown in Figure 10. For this lab, there were 6 respondents: 3 consumers and 3 producers. As is evident from the overlay, the perceptions of importance coincide well in general, with the greatest differences being in integrity and interoperability. A close match between the consumer and producers' scores indicate a common understanding of end-user's needs and is the first step in determining whether consumers and producers are in concert to address the end-users' needs.

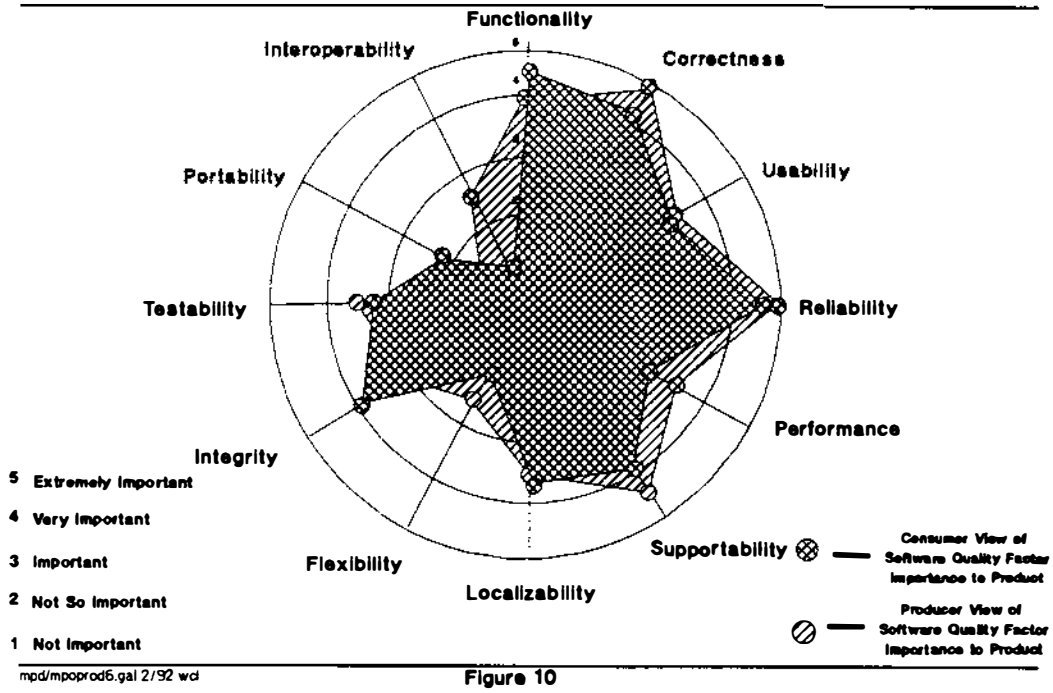


Figure 10

In the case where the producer has consumers who address multiple end-users with significantly differing needs, we obviously expect to see greater score differences. In this situation, more emphasis should be placed on analyzing the producers' perceptions of the consumers' needs.

b. Reuse Impact on Product SQF

The MPO consumer rating of the impact of reuse is overlaid onto the consumer rating of importance of the software quality factors to the product in Figure 11. The factors where importance exceeds impact most are in Performance, Localizability and Integrity. Flexibility and Portability are the two areas where reuse impact exceeds importance. This chart contrasts the consumers' perception of how much impact reuse is having upon those areas that are considered important to the them. This analysis is useful as a starting point for a producer and consumer group discussion on how reusable workproducts can be created or utilized to address some of the important but as yet underaddressed SQFs.

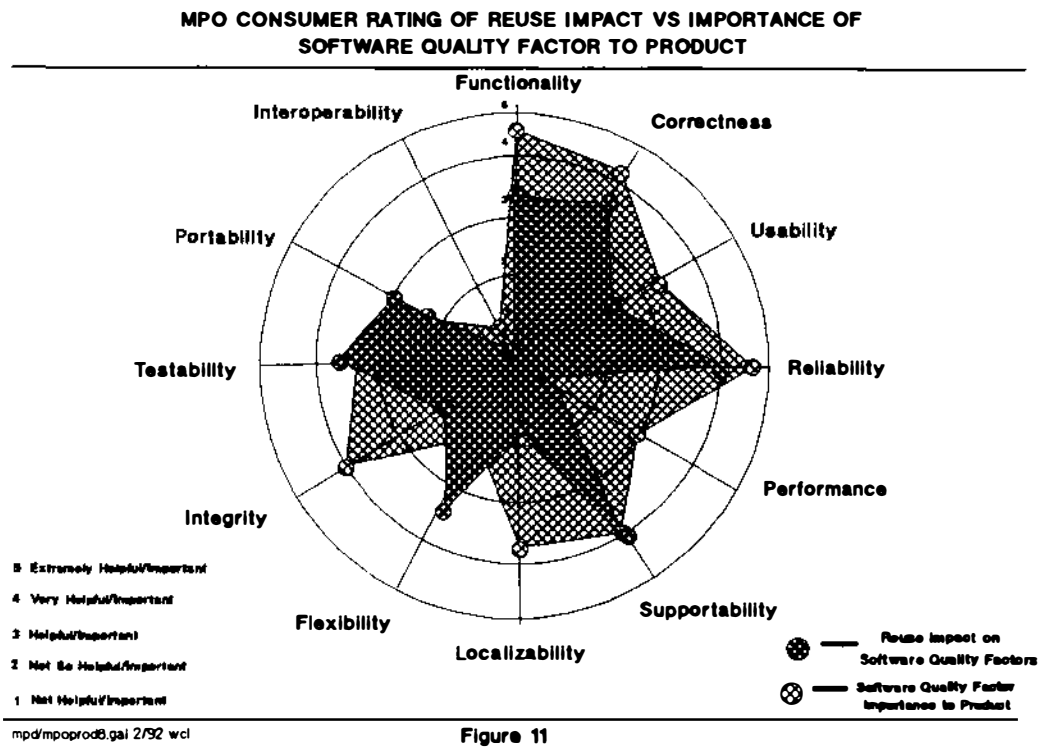


Figure 11

Conclusion

Software reuse can provide substantial benefits. Recent data collected at the Hewlett-Packard company indicate reuse can improve quality and productivity significantly. However, these benefits are offset by costs associated with reuse. An economic analysis on 15 workproducts indicate that some of the workproducts have resulted in an overall economic gain while a few have resulted in a loss.

Surveying producers and consumers on their perceptions of the importance and impact of reuse on software quality factors is useful as a starting point for a discussion on how reusable workproducts can be created or utilized to handle important but underaddressed software quality factors.

References

[GRAD] Grady, Robert and Caswell, Deborah, "Software Metrics: Establishing a Company-Wide Program", Prentice-Hall, 1987, pages 186-187.

[LUBA] Lubars, Mitchell D., "Affording Higher Reliability Through Software Reusability", ACM SigSoft Software Engineering Notes, vol. 11, no. 5, Oct. 1986, Page 39.

[MCCAL] McCall, J.A. , Richards, P.K. Walters, G.F. , "Factors in Software Quality Assurance: RADC-TR-77-369, I, November 1977, cited in: Vincent, James P. , et al, Software Quality Assurance, Prentice-Hall, 1988.

[NISH] Nishimoto. Alvina. "Evolution of a Reuse Program in a Maintenance Environment", 2nd Irvine Software Symposium, March 1992.

Implementing An Inspection Process

By Steve Whitchurch

Mentor Graphics Corporation
8005 S.W. Boeckman Road
Wilsonville, OR. 97070-7777

(503) 685-7000, x2671
stevenw@pdx.mentorg.com

Abstract

The inspection process has been around since the mid seventy's when Michael Fagan from IBM introduced it to the software community. There have also been a lot of papers written on the impacts that inspections can have on the quality of production software. This paper will focus on the implementation of such an inspection process.

Biography

Steve Whitchurch has worked for Mentor Graphics Corporation for three years as a Software QA Engineer. Before coming to Mentor Graphics, Steve was in a QA group at Intel Corporation where He tested real time operating system software. Steve is a recipient of the Mentor Graphics Chairman's Achievement Award and the Chair Person for the Software Association of Oregon QA SIG.

Introduction

The focus of this paper will not be on the inspection process, but on the implementation of an inspection process. The paper will start with a brief history of my experiences with different types of inspections and a brief explanation of why I chose the process that I did. The rest of the paper will present my experience from implementing an inspection process at Mentor Graphics. The paper will conclude with data from pilot groups using the process and future plans I have for the inspection process at Mentor Graphics.

History

Like most software development organizations their is some type of review process being used. It could be code reviews, documentation reviews, or walkthroughs. And like most software engineers I have been involved in all three processes.

Code Review

Code Reviews usually involve some one handing out code a day or two before the actual code review meeting. At the code review meeting every one jumps in with comments related to defects found in the code, and for the most part there is no structure to this meeting. This is the reason that code reviews don't work. Without some type of process model, it's very difficult to have an efficient and cost effective process. A problem that will result from an inspection process without some type of structure, is that the focus can change from reporting defects to a debate about a defect reported.

Documentation Reviews

Documentation Reviews are the worst kind of inspections. Most documentation reviews involve handing out a document with instructions to return the document to the author at some date with comments. What usually happens is that the documents sits on some one's desk past the due date without being reviewed. This is also a process without structure.

Walkthroughs

Walkthroughs are often confused with inspections. Walkthroughs are a way to communicate information, and are not focused on finding defects. The walkthrough process is a very good process used to teach and stimulate debate about a design. The inspection process does not replace the walkthrough process. The walkthrough process is used when discussion about a problem or design needs to take place, while the inspection process is used to find defects.

After years of being involved with these types of inspection processes and not satisfied with the results. I researched inspection processes and found an inspection process that I thought would work in my organization. The inspection process that I chose supported an inspection process model similar to the Fagan Inspection Process. The process supported a good process model, and had defined entry and exit criteria. The rest of this paper will present the process that I went through to implement the inspection process at Mentor Graphics.

Implementing the Inspection Process

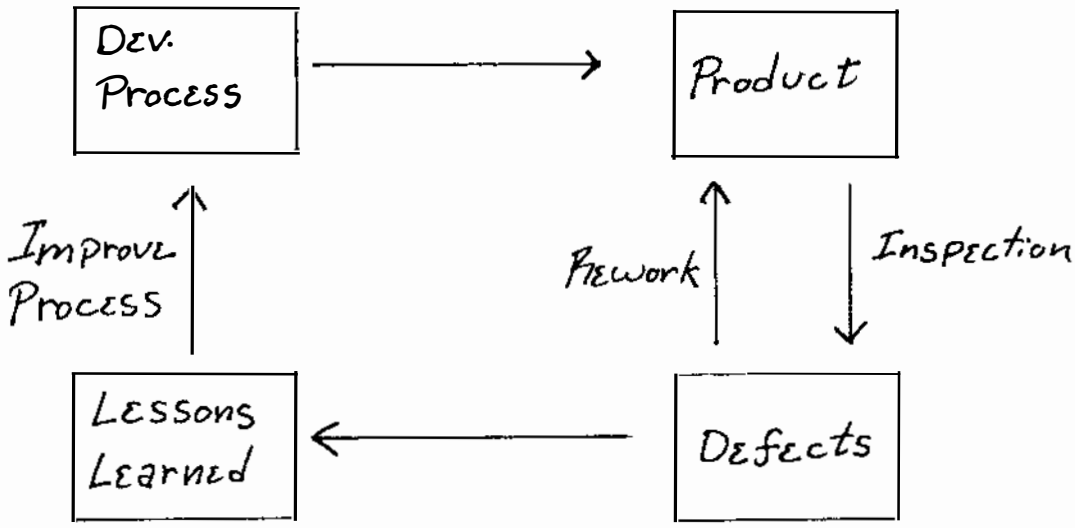
The real challenge associated with the inspection process is the implementation of the inspection process. The following sections represent the implementation process I used to implement an inspection process at Mentor Graphics. I will present some ideas that can help define a process for your organization. I also talk briefly about selling the process, pilot groups, and supporting the process.

Defining the Process

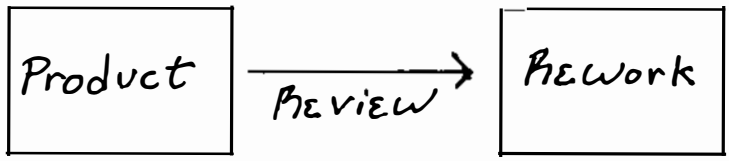
The first step in implementing any process is defining what that process should be, and what are the attributes of a good inspection process.

A good inspection process should include.

1. *A good process model.* **A good process model would look like the following.**



An example of a bad process model would be the average code review or documentation review.



2. *Metrics support.* **The process should include metrics that can be used to monitor the inspection process.**

3. *Entry and Exit criteria.* **Defined entry and exit criteria for each phase of the process model.**

4. *People friendly process.* **If the inspection process is not people friendly, people will not use the process.**

5. *Software life cycle support.* **The process should be applicable at any phase of the software life cycle without modification.**

6. *Support data collection.* **Their are five categories of defect information that should be collected.**

   i. **The severity of the defect: Was it a major defect or a minor defect.**

   ii. **What type of defect was it (e.g. documentation, logic error, etc).**

   iii. **The location of the defect: Where in the document or code was the defect found.**

   iv. **A description of the defect: A way to record the defect found.**

   v. **The most common defects found (common errors check list).**

The next step in defining the process should address the logistics of teaching the process. If you have selected a common inspection process like the Fagan Inspection Process, you will find that there are several consultants that teach a version of this process, this is what I did.

Selling the Process

Selling the inspection process can be the most difficult part of the inspection process implementation. If the first phase of implementation process (defining the process) was well thought out, the selling of the inspection process can be made much less difficult.

The first group that will need a sales presentation will be your management. Management buy-in is very important to the implementation of the inspection process. The following ideas can help with the management presentation.

- **Slide set that presents the inspection process model. The slides should include how inspections can cut costs in the development of a product as well as increase the product quality.**

- **Include Metrics in your presentation. The question will be asked, "How do we measure this process"**

- **Have some examples of other organizations using inspections, and present the data from those inspections.**

The next group you will need to sell the process to is the people that will be expected to use the inspection process, the engineering staff. A good rule to remember is "25/50/25". This rule means that you will have 25% of the staff want to use the process without any selling, 50% of the staff will use the inspection process over time, and 25% of the staff will never use the process. The thing to keep in mind when implementing the inspection process is, don't beat your head against a wall trying to convince the later 25% to use the inspection process. Focus your efforts on ways to sell the 75% of the people that will use the process. The following are ideas you can use to help sell the staff.

- **Present the inspection process at group meetings**

- **Hold a general meeting where you can present the inspection process.**

- **Use EMail to let people know that the process exists.**

- **Use the companies news letter to inform people that the process exists.**

Pilot Groups

Now that the inspection process has been defined, and management has agreed to fund the project, the real fun can start.

The first thing to do is identify a pilot group or groups that would be interested in using the inspection process. The thing to keep in mind at this point, is that you are still selling the process. So you will want to pick a pilot group or groups that will return positive feedback. You do not want to pick a pilot group that is out to discredit the inspection

process.

Once you have identified the pilot group or groups, you will want to teach them the process. This part of the implantation is very important and should be thought out very carefully. The implementation process will fail if the introduction to the inspection process is of poor quality. This is where an off site consultant can help. I found that the consultants could teach the process much better then I could.

As the pilot group or groups perform inspections, keep a data base of the results from these inspections. This information will become very important in the future to prove that the process is working and if it's cost effective. The data from these inspections will also tell you were improvements need to be made in the inspection process.

Supporting the Process

Supporting the inspection process is the same as supporting any product or process. without good customer support, the process will fail. The following is a list of ideas that will help support an inspection process.

- *Support Group*. The support group meeting is a very good forum where questions can be asked about the process, where problems can be resolved, and where people can get involved in the inspection process. This group meeting is also a good place to invite people to talk about there successful inspections.

- *Mail Group*. A mail group can help with the distribution of information about inspections and the inspections process. The Mail group is also a good forum for questions.

- *Local News Group*. The news group has the same advantages of the mail group.

- *Central Depository*. The central depository would be a place users of the inspection process could go to get copies of common error check lists and copies of standards.

- *Training*. On-going training is the key to any process. New people wanting to get involved in the process will need to be trained.

- *Certification.* **Some type of certification for inspection moderators is very important. The only way you can collect maningful data from an inspection process, is by having the process be consistent in the organization. Certification will help solve this problem.**

The main theme here is to keep the inspection process visible to everyone in the organization. If the visibility fades, the inspection process will also fade.

The Human Factor

One of the main problems that will be encountered when implementing an inspection process will be getting people to use it. The following is a list of the common negative comments made about the inspection process.

1. **The defect information collected by the inspection will be used against me in my review.**

2. **Every time I have my work inspected, I'm personally attacked.**

3. **Every time I find a defect in an inspection, I'm told it's not a defect and its not recorded. Why should I participate ?**

Comment number one is probably the most common of the comments listed. It's very important that the inspection process is never used as a data point for a personnel review, or a way of grading people. If your inspection process gets this reputation, you will fail in the implementation of an inspection process in your organization. The inspection process should be viewed as a tool to find defects, and not a way to grade people. The following are some ways to address this problem.

1. **No Managers or Supervisors at the inspection meeting. The only exception to this rule would be if the Author feels comfortable with this person, and would like them at the inspection meeting.**

2. **Keep any defect data local to the group doing the inspections. The purpose of doing inspections is to find defects and learn from the common errors that are being made by the group. The key word**

here is "group". Work on promoting the inspection process as a group activity and not a personal review process.

3. **In some cases statistics from the inspections need to be kept by a group outside of the group doing the inspection. In this case do not make the names of the inspection participants public. Leave the name off any inspection forms. Again the goal here is to promote inspection within the organization and to collect data to measure their effectiveness, and not to review people.**

Comment number two is the second most frequent comment you will hear when implementing an inspection process. One of the problems with most inspection processes, walkthroughs, or code review processes is the lack of control over the defect reporting meeting. If the process does not have trained inspection moderators the inspection process will become a contest of egos. The following is a list of possible ways to avoid this problem.

1. **Offer inspection moderator training. In the training class emphasize the importance of keeping the defect reporting meting on track, and not allowing personal attacks during the meeting. If the meeting gets out of hand, cancel it and re-schedule.**

2. **Let the author pick the inspectors that will attend the inspection. If the author and inspectors are comfortable with each other, attacking during the defect reporting meeting will be minimal.**

3. **Make it clear that the inspection defect reporting meeting will be used to log defects, and that personal attacks will not be tolerated.**

Again the key to making the inspection process work, is to emphasize that the process is a tool to help find defects in software, and not to be used as a people review process.

Comment number three is a direct affect of not having a good inspection process in place. The key to any good inspection process is the moderator training. If the inspection process does not make use of the moderator role, you will experience this problem. The moderator will keep the defect reporting meeting focused on finding and reporting defects, and not let the meeting turn into a discussion meeting. The following are some ways to help address comment number three.

1. **All defects reported in the defect reporting meeting are logged without debate.**

2. **Make it clear that every one involved in the inspection was picked because they are experts, and every defect and comment will be logged.**

The important thing to remember about the inspection process, is that it's a people process. It's very important that this theme is used from the beginning of the implementation of the process.

Metrics

No process would be complete without some way to measure its effectiveness. A good inspection process will have two types of metrics. The first type would be a metric that groups using the inspection process could use to gauge their inspection effectiveness. This metric would be used to adjust the amount of material that the group can effectively inspect in a defect reporting meeting. This metric is called the logging rate.

Logging-Rate = Total-Defects-Logged / Logging-Time

The second type of metric would be more of a management metric. This metric supplies management with data on how effective the inspection process is.

Defects/Man-Hr. = Total-Number-Defects / Total-Inspection-Time

The true indicator that the inspection process is working, will be the decline in the number of defects found in the testing phase of the software life cycle. These numbers can be derived from past product defect history. A decline in the number of defects found at test will reduce the schedule time.

Results

The following table shows inspection process results from ten pilot groups doing inspection at Mentor Graphics.

Pilot Group No.	Logging Rate (per Min.)	Defects / Man-Hr.
Pilot #1	1.9	4.02
Pilot #2	1.27	4.33
Pilot #2	1.33	2.0
Pilot #3	1.0	2.75
Pilot #4	0.95	2.98
Pilot #4	1.675	5.7
Pilot #5	0.44	6.0
Pilot #6	1.56	5.43
Pilot #7	3.375	10.2
Pilot #8	1.8	6.5
Pilot #9	1.4	4.53
Pilot #10	1.7	5.1

The data that would be useful to a group using the inspection process is the logging rate. Most of the pilot groups fall with in the industry standard of 1 to 2 defect found per minute. For a pilot group like number five whose logging rate is below the standard, there could be a couple of problems.

1. **Inspecting too much material at once.**

2. **Logging nit-pick defects (e.g. spelling errors, grammar errors). These type of defects can be marked on the review material and handed back to the Author.**

3. **Could have a problem with the Moderator letting the reporting meeting turn into a discussion meeting.**

Pilot group number seven has the opposite problem of group number five. This pilot group's logging rate is above the industry standard. This group would want to increase the amount of material being inspected at each meeting.

The data that would be of importance to management would be the defects-per-hour numbers. This data would need to be compared to past product history.

Even though the implementation of the inspection process at Mentor Graphics is still in the pilot phase, the numbers from these pilot groups do indicate that the process is working.

Future Plans

The inspection process that has been implemented at Mentor Graphics is far from being complete. The following are future plans for the inspection process.

- **Continue to collect data on inspections being done.**
- **As projects complete, identify any impacts that the inspection process had on the quality of the product, the product schedule, and the number of times regression tests had to be run.**
- **Continue the monthly support group meetings.**
- **Sell the process to all Mentor Graphics sites.**
- **Develop a base of certified Inspection Moderators.**

Summary

The main things to keep in mind when implementing any inspection process is to select a process that best fits your organization, and address the concerns that people could have with the inspection process. A good thing to remember is that the inspection process is a people process. And if the people in your organization do not like the process you have chosen, no one will use it.

The inspection process at Mentor Graphics has been a grass roots type of a process. I don't think it would have been received as well as it has if the process was dictated from above. I think this emphasizes the theme that the inspection process is a people process. I would discourage implementing an inspection process that forces people to use it.

The final recommendation would be, strongly consider using a consultant to teach the process to your organization. I found this to be one of the key decisions that has contributed to the success of the inspection process at Mentor Graphics.

Acknowledgments

I would like to thank the following people for helping me with the implementation of the Inspection process at Mentor Graphics. without these people the process would not be what it is today.

- **The Process Group**
  Mary Sakry and Neil Potter.
- **Scott Killops, John Stedman, Lyle Sweeney, and Eileen Boerger.**
- **All the Engineers, Writers, and Managers that attended the Inspection Moderator Training classes.**

References

Edward Yourdon, "Structured Walk-Throughs", fourth edition, Prentice-Hall publisher.

Daniel P. Freedman and Gerald M. Weinberg, "Handbook of Walkthroughs, Inspections, and Technical Reviews", third edition, Dorset House publisher.

Daniel P. Freedman and Gerald M. Weinberg, "Reviews, Walkthroughs, and Inspections", IEEE Transactions on Software Engineering, Vol. SE-10, No. 1, January 1984

A. Frank Ackerman, Lynne S. Buchwald, Frank H. Lewski, "Software Inspections: An Effective Verification Process", IEEE May 1989

M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, Vol. 15, No. 3, 1976

Michael Fagan, "Introducing Inspections in your Organization - The Defect-Free Process", SRI Conference Proceedings, May 1990

Michael E. Fagan, "Advances in Software Inspections", IEEE Transactions on Software Engineering, Vol. SE-12, No. 7, July 1986

James S. Collofello, "The Software Technical Review Process", SEI Curriculum Module SEI-CM-3-1.5, June 1988

Clen W. Russell, "Experience with Inspections in Ultralarge Scale Developments", IEEE Software, January 1991

Carl R. Dichter, "Two Sets of Eyes", Unix Review, Vol. 10, No. 1, 1992

Focus On Requirements: First Things First!

Tamara Baughman
Dave Kearns
Trevin Pick
Nitin Rai

Mentor Graphics Corporation

8005 S.W. Boeckman Road
Wilsonville, Oregon 97070-7777
(503) 685-7000
Email: nrai@MENTORG.COM
tpick@MENTORG.COM
dkearns@MENTORG.COM
tamarab@MENTORG.COM

Abstract

This paper describes industry experience using innovative approaches to generate, analyze, prototype and track requirements during the development process of a highly interactive tool. Our emphasis on requirements, prototyping, and soliciting feedback from users throughout the development process kept us on target to deliver a quality tool that met the user's needs. Our choice of support tools helped a great deal. We used the RDD-100^{tm} tool from Ascent Logic Corporation to do our requirements tracking and the Mentor Graphics Framework^{tm} application called Decision Support System^{tm} to prototype user interface ideas.

Biographical Sketches

Nitin Rai has a B.S. in Computer Science from Dalhousie University, Halifax, Nova Scotia, Canada. He has worked at Mentor Graphics Corporation for 3 years in the Common User Interface group. He is currently the Project Leader for the GUI Builder (Dialog Box Editor) and AMPLE debugger projects. He has been working in the areas of object oriented user interface toolkit development and object management systems for the last 5 years. He is a member of ACM and SIGCHI.

Trevin Pick has worked at Mentor Graphics Corporation for the past 5 years as a software development engineer. He is currently in the Common User Interface group. He has a B.S. in Computer Science from California State University, Chico. Prior to working at Mentor Graphics, he worked on PC project management software at IBM in San Jose.

David Kearns has worked at Mentor Graphics Corporation for several years as a software development engineer in the Common User Interface group. He has a B.S. in Computer Science from the University of Oklahoma. He has worked in the areas of user interface toolkit development and avionics control systems for seven years.

Tamara Baughman has a B.S. in Computer Systems Engineering from the Oregon Institute of Technology. She has worked at Mentor Graphics Corporation for 5 years and is currently a QA engineer in the Common User Interface group. She is currently a team member working on improving the development process and capturing process/product metrics for an interactive GUI Builder tool.

1. Project Vision

Our project was initiated to add usability tools to the Mentor Graphics Falcon Framework^{tm} Common User Interface (CUI). The CUI is an object oriented user interface toolkit developed using C++ with Motif appearance and behavior. The CUI also has an interface to a C-like extension language called AMPLE, which is used to create and modify portions of the user interface, including dialog boxes and menus. Historically, application developers and end users creating or modifying dialog boxes have been required to write AMPLE code. Writing dialog boxes in AMPLE is a complicated and time consuming task with a large learning curve. There was a clear need and demand for a graphical tool which end-users could use to create and customize dialog boxes. In addition to the dialog box editor, we are considering developing a menu editor and an AMPLE function browser tool.

Our project vision was not only to create a graphical builder tool to improve the CUI's usability, but also to produce a bug-free software tool that developers liked to use. Towards this end, the team members reviewed projects they had been involved with in the past. We believed that many of the bugs fixed late in the development cycle on previous projects were caused by late customer requirements. These requirements were either not captured or were not clearly understood early on by the engineer. The projects we reviewed also had little or no requirements traceability for functionality. This caused problems in differentiating between what was a bug and what was expected functionality. Previous projects also lacked early participation of quality assurance (QA) and documentation teams, forcing QA and documentation to play "catch-up" at the end of the projects.

The team also reviewed other projects within our company that had received high marks from customers for quality, usability, and performance. One particular project team had done an exemplary job at capturing and understanding customer requirements up front. They tracked requirements and delivered a product that met those requirements. Our plan was to emulate the requirements driven approach that they took when we started our own project.

The Corporate Engineering group was the other motivational source for us. Our team took a training class from the CE group in systems engineering. The class focused on software engineering methodology and its value in producing high quality software that met customer requirements. This training further enforced the ideas of capturing customer requirements and building a behavioral requirements specification up front. Also, training gave the team some experience with RDD-100^{tm} (Requirements Driven Design), a tool under evaluation by the group for requirements capture and analysis. Despite some shortcomings in the tool, the team made an early decision to use RDD to capture and track requirements, to perform requirements analysis, and produce a behavioral specification for our builder tools.

Requirements focus became an important component of the overall project vision both for the team and engineering management. This focus continued through all project phases.

2. Initial Requirements Gathering

At this stage in the project, we wanted to collect a database of as many requirements for our tool as possible, independently of whether it was possible to implement them all in the desired timeframe. Our technique was to use several methods of requirements generation from different sources and to organize this data for efficient review and traceability later. We also needed a way to track requirements so that none would get lost. This might not seem important at the project start, but becomes so later when a customer asks "What happened to my requirement?" or someone asks you to generate a list of all requirements which came from Company X. In cases like this, an electronic database tool becomes invaluable. Another difficulty when gathering requirements is that people confuse true requirements with the functional description (or behavior) of an item. It's important to keep these two things separate and focus on the requirements first.

We started with a "marketing requirements document" (MRD) written by our group marketing representative. It justified the need for a builder tools project with market analysis and customer input. The MRD also provided working definitions and analyzed the target market. In this case our target market varied from internal application developers to external customers wishing to customize and extend our CUI. The MRD also discussed existing customization tools available for competitive user interface toolkits such as Motif™ and Openlook™. Finally, it outlined about twenty basic functional operations necessary for an interactive dialog box editor to be adequate. See figure 1 for an example MRD page.

The next step was for engineering to generate an investigation proposal and a formal proposal. These proposals were needed to get management approval to formally start the project. The proposal typically includes such things as current technology review, proposed schedule, resource needs, and basic functionality which will be delivered. Our proposal included two sections which management and other engineers found very useful: a section on actual customer input and a section of simple storyboards which illustrated basic scenarios and possible visual layout of our product. The project engineers developed the storyboards based on early requirements gathered and a general knowledge of good user interface design. We used MacDraw™ on the Macintosh to develop the storyboards.

The formal proposal meeting was attended by both managers and interested engineers. We presented justification for the project and then went through storyboards for the three proposed tools. This generated some very productive discussions and we noted comments and suggestions as additional requirements. At this early stage, it's unusual to have storyboards but we found they generated useful requirements, solidified some concepts which would be hard to describe in words, and helped generate excitement about the project. People seem to respond well to visuals, and even at this early stage it was beneficial to generate some possible tool pictures and behaviors. See figure 2 for an example storyboard.

During this initial requirements phase we scanned through our online database of reported defects and enhancement requests for our products. By checking this database, we were able to mine a few more requirements. Also, the engineers working on the project had a significant amount of CUI and customer experience, so they contributed additional ideas. At this point we had dozens of requirements from different sources and were a little overwhelmed on how to address them all. Fortunately, at about that time, we heard of an electronic database tool, RDD-100™, which was being used in some parts of the company. We decided to investigate to see whether this tool would help us manage our requirements and provide traceability throughout the project lifecycle. After taking systems engineering training and learning more about the tool, we decided that it would indeed solve our requirements management problem.

We used the RDD-100™ tool to support requirements gathering and analysis. This tool is sold by Ascent Logic Corporation and is actually a full fledged CASE tool; however, we only used the requirements gathering and analysis portions of it. The first step in using the tool is to extract requirements from source documents. RDD™ allows you to import an ASCII document and then interactively highlight those portions that specify requirements. You then paste these fragments in a "requirements element" and add it to the permanent database. The tool allows you to track source documents for later reference. Whenever you receive additional requirements, you insert them into the database. This takes a certain amount of discipline, but provides comfort in that requirements are never lost and can be sorted in various ways. Also, once the requirements are in the database you can specify the behaviors, critical issues, and decisions made on critical issues with the tool providing methods of recording all this work. See figure 3 for an example RDD database requirements page.

3. Prototyping the Requirements

After gathering many "raw" requirements, we developed a prototype of the dialog box edit tool in order to prioritize desired behaviors and to generate more feedback from potential users. The problem was that we did not have much time and we thought it would be too time consuming to use our existing user interface library to build full prototypes. We were lucky in that the Mentor Graphics Framework^{tm} includes a very useful application called Decision Support System^{tm} (DSS).

The DSS spreadsheet application not only supports number crunching operations but also lets you put graphical widgets like buttons or dials inside the spreadsheet cells. You can write formulas for these gadgets that will execute various intrinsic functions available to all Mentor Graphics^{tm} applications. In this way, we rapidly developed a prototype interface which performed some simple operations. These operations were not "real", but rather simulated possible functionality in a limited fashion. We developed prototypes for all three tools within one week: a menu editor, a dialog box editor, and a function browser.

Once the prototypes were ready, we invited some potential users to participate in a hands on demonstration. These customers were able to actually simulate many operations which would be implemented in the tools. This type of early prototyping generated excellent feedback and requirements. Again, people responded to visuals and hands-on experience much better than to printed words. Even though it was still early in the lifecycle (not a single line of code designed or written) and the prototype had limited functionality, users seemed genuinely excited and eager to give us more data. We felt this prototype gave us very useful and high quality feedback at a stage in the project where we really needed it. The highly interactive, graphical nature of our dialog box editor required careful attention to design and usage paradigms. We took the feedback from these demo sessions and entered it into our existing RDD-100^{tm} requirements database. See figure 4 for sample interview questions and figure 5 for an example DSS prototype screen shot.

4. Requirements Analysis

We finally felt ready to start our detailed analysis of requirements, system behaviors, critical issues, and functional priorities. This critical phase made large demands on the project team due to the requirements volume and our desire to understand requested behaviors. We also realized it would be impossible to deliver everything desired in the first release, and we had to figure out how to build a system with enough features that it would be useful. Again, the RDD-100 tool helped us to

record critical issues and decisions on those issues, as well as system behaviors. It also generated hardcopy reports of these areas for others to review. For example, before meetings we printed out a list of unresolved critical issues. This was invaluable in making sure no issues were forgotten.

Determining system behavior is the fundamental goal of requirements analysis. The team developed a number of tool behaviors which satisfied certain categories of requirements. For example, the behavior might be "Tool Invocation". This incorporated several requirements such as:

The tool shall invoke in under 10 seconds.

The tool shall invoke within an existing application rather than standalone.

The tool code shall not increase the size of the base libraries.

The invocation behavior would be documented as a textual description which satisfies the requirements. In this case, you would describe in detail how the tool would invoke, and how the user would invoke in the two modes (create a new dialog box and edit an existing dialog box). We described the system for the dialog box editor tool in 17 behaviors which incorporated 33 base requirements. The behavioral description was entered into our online database and a relationship was established between each behavior and the documented requirement for that behavior. This formed the traceability link between source documents, requirements, and behaviors. This sounds fairly easy, but in fact required a great amount of group discussion and understanding of how users will use the system.

People suggested alternate behaviors which technically satisfied the requirements, but which might have resulted in poor usability. We found that developing "scenarios" helped bring out certain critical issues and allowed further explorations. A scenario can be as simple or complicated as the individual wants. Some team members drew cartoon-like pictures which illustrated the users work flow. Others verbally described a situation and possible solutions. We captured these scenarios in the RDD tool for future reference. See figure 6 for RDD^{tm} screen shots showing requirements, critical issues, and decisions. Figure 7 shows an example behavioral description from the tool. Discussing these scenarios in an open forum without allowing personal attacks generated lots of ideas and alternatives. The team could then weigh the alternatives and reach consensus on the best behavior. During this step, it definitely helps to have a team of mature developers who work well together.

It is also important to have engineering, quality assurance, documentation, marketing, and customer support people present to give alternate views and begin planning for how this product will affect their area. QA can remind developers of quality issues up front, documentation can start thinking about a user's guide, marketing can think about how to position the product, and customer support can start planning for assisting customers who use the tool. All can begin to understand how the product will work. Usually product information is known only by the development engineer, who later has to explain it repeatedly to the rest of the team after the fact. Imagine how much understanding and history is preserved by spreading the information through the entire team throughout the development process! Everyone is able to do their job better because they understand the details and product history.

5. Proof of Concept

A number of our requirements were fundamental to making our product useful, but we didn't know if it was possible to build a system which would satisfy those requirements. In some cases, we were unsure that the usage paradigms we envisioned were actually usable. Finally, we realized the functionality provided by our user interface library would have to be modified or extended considerably to meet our needs. Since clients of our user interface libraries requested that we not add functionality, we were forced to build an application which was intimately tied to the user interface libraries without modifying those libraries. We reached consensus on functionality and designs that we thought would work, but rather than design the entire system before beginning implementation, we chose to prove that certain key elements in our design were feasible before beginning detailed design and implementation. We did proof-of-concept testing on these elements and saved ourselves the rework caused by poor design. These design elements included:

- gadget layout highlighting scheme for groups of gadgets (rows/columns), indicating whether the groups are selected or deselected

- display list (dialog box and gadget) design which supports the ability to insert/delete/modify gadgets

- ability to generate valid AMPLE descriptions of our display list dialog boxes

- ability to alter and restore code-level visibility of AMPLE bindings on the fly which was needed to create our own versions of AMPLE defined dialog boxes as needed

For example, our model for the user editing groups of gadgets in the dialog box had the system painting borders around related groups of gadgets. The groups could be in rows or columns, and the groups could be nested within other groups, as well. We built an executable using modified C++ user interface libraries which painted borders around all rows or columns on any dialog box which was displayed. We solicited feedback on the way our paradigm looked and used the prototyping experience and feedback to refine our design.

Another issue we faced was that the user interface libraries provided by the Falcon Framework™ did not support run-time editing of dialog boxes. This forced us to build our own dialog box display list system by deriving from some library C++ classes and defining other classes from scratch. Our proof-of-concept work in this area pointed out to us that we had more work to do than we had thought. We would have to completely re-implement each of the gadgets and all of the dialog box display list underpinnings to make things work. We concentrated our prototyping efforts on re-implementing a small subset of the display list and gadgets, and based our overall design on what we learned from that work.

We also needed to be able to generate AMPLE descriptions of dialog boxes during the edit sessions to make the product useful. At any time during the edit session, the user should be able to save the dialog box, generating syntactically correct AMPLE. The AMPLE needed to be readable, compact, and correct - whatever the dialog box state during the edit session. We had an idea for how to traverse the display list to produce the userware description, but we wanted to be sure that our algorithm for AMPLE generation worked before trying to hook more than thirty gadgets into the code generation system. Our prototype worked like a charm, so we went with the design used for proof of concept.

Doing this proof of concept work gave us a number of benefits. It allowed us to explore alternative designs early in the development cycle, before we were so far along with our implementation that it would have been impossible to try something different. Also, it proved to us that our fundamental designs were sound, so our detailed designs were based on things that we had proved would work. Finally, we were able to solicit feedback from potential users earlier in the design cycle, to be sure that they would be satisfied with the user paradigms.

6. Benefits of the Requirements Process

The upfront requirements capture and analysis process was very beneficial to later project phases, from the functional specification to actual implementation, documentation, and testing. The system reports from RDD helped us to generate system behaviors, which served as a backbone for all other project phases. The functional specification not only defined system functions, but also the behaviors that mapped to those functions. The system behaviors and requirements guided the engineers in creating the functionality essential for meeting the requirements. By tracking the requirements so closely, it was easy to get rid of functionality that did not trace back to a particular requirement. This helped prevent creeping functionality and kept us focused on those things which customers asked for.

The design and system architecture was based on the behaviors we specified. The implementation of the design was fairly straightforward, as our design was thorough and pre-tested in our prototypes. Despite having limited resources on the project and a large number of unscheduled interruptions, the functional specification, design, and implementation were completed ahead of schedule. Our detailed understanding of what behaviors we needed to implement helped improve the schedule accuracy. Everyone agreed that having a clear vision of what needed to be done made it much easier to develop the tool. A complete set of requirements gave us that vision.

The documentation and testing activities were part of the project from the start, and the result was better quality in the product and earlier documentation available for our product. A user's guide and a reference guide were easily generated based on the behaviors and system functions documented in the functional specification. Complete, usable documentation was shipped with the alpha release.

The behavior database also assisted the QA engineer in planning testing activities even before the behaviors were implemented. The test case design was completed in parallel with the system design and architecture phase. The QA engineer was able to write automated test suites based on the design and the functional specification during the software implementation phase. The tests were run on internal integration releases prior to the Alpha release. Consequently, defects were found concurrently with the development process and all critical ones were fixed before the Alpha release was shipped to customers. This was clearly a step towards improved software quality.

Basing the entire software development process on the requirements provided a common understanding among all team members of how the tool was going to work.

It left us with few problems in the integration of work from different engineers and reduced the overhead of communication between engineers to resolve programming inconsistencies. The team members were constantly sidetracked with other higher priority tasks in other projects but the team was able to complete the required Alpha functionality on schedule. We think that this was due to our focus on requirements.

7. Areas for Improvement

Here are some things we would like to improve on or do differently for our next project:

- We missed interviewing some customers who would likely be heavy users of our tool. It would be nice to spend more time conducting interviews. Usually time constraints and company budgets prevent this.

- We would like to have used a simpler tool than RDD™. The learning curve for the requirements capture tool was large and it provided much more functionality than we ever used. A tool with a cleaner user interface that just handled requirements, critical issues, decisions, time functions would have been enough.

- We would have liked more help from marketing in gathering input about our product from key customers.

- We wished we could have spent more time prototyping. We felt a bit uncomfortable only spending a week on the DSS prototypes.

5.3 Requirements For a Dialog Box Editor

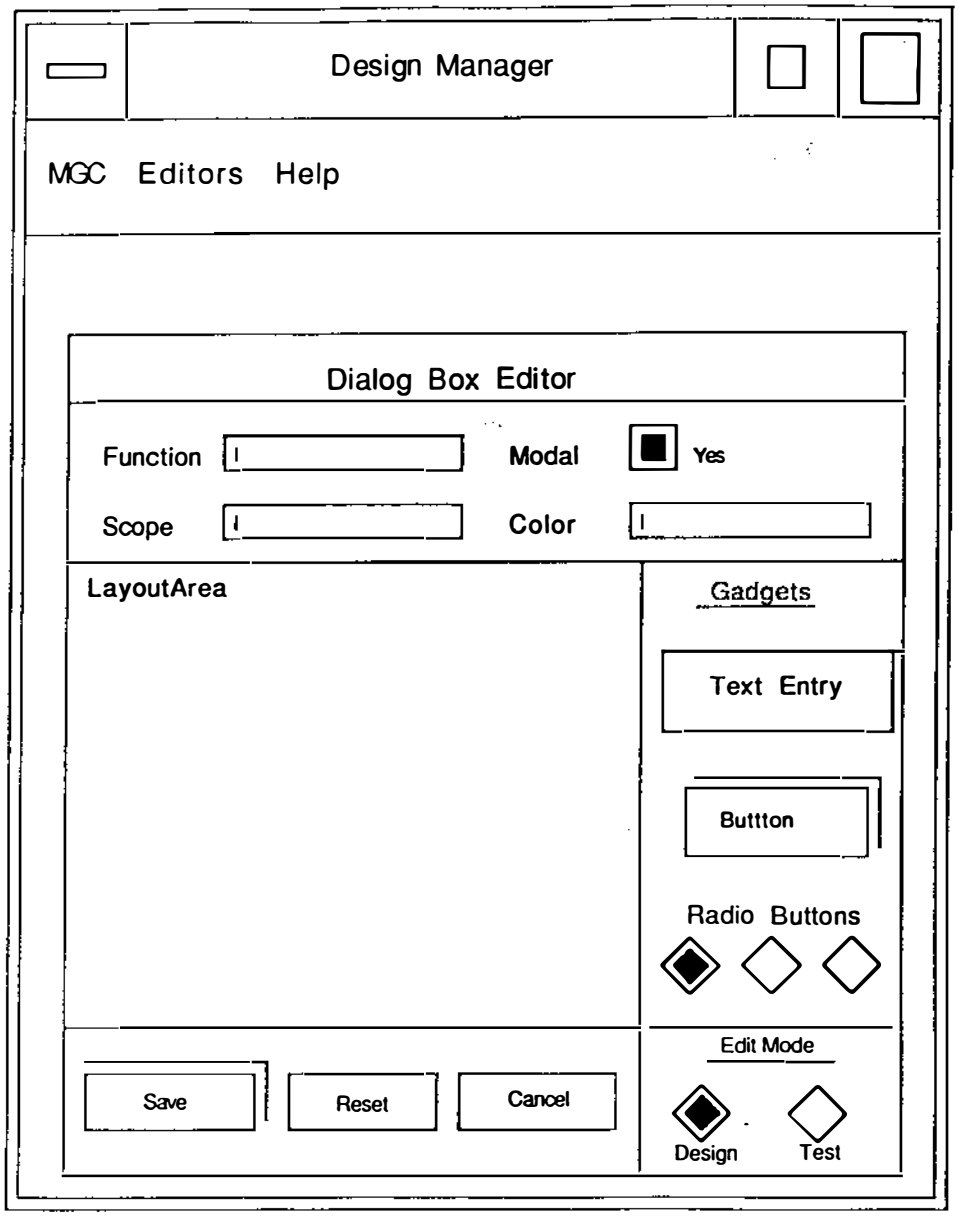
A simple dialog-box builder was part of the phase 2 and phase 3 UI toolkit releases. Users could place controls, move them around, re-size them and write out the definition when the design was complete. Even a simple tool like this has great utility.

Basic dialog-box editor operations must include:

- create new dialog boxes
- modify existing dialog boxes
- write out an AMPLE definition to a file
- load dialog boxes from scopes or from files
- display scope associated with a definition
- a palette of tools (a list tool, a check-button tool, etc.)
- a select-drag-place method of placing controls
- select a control
- move a control
- delete a control
- link operation to connect dialog box to a function
- enter and compile a new dialog box function inside the editor

MRD Page

Figure 1



Storyboard

Figure 2

Traces To:
CriticalIssue: 10 - Grow/resize a gadget
TimeFunction: 13 - Moving Gadgets

Source Document:
mrd document

2.4 - Gadget placement paradigm

Description:
a select-drag-place method of placing gadgets must be implemented

Traces To:
TimeFunction: 5 - Gadget Placement

Source Document:
mrd document

2.5 - Gadget selection

Description:
select a gadget

Traces To:
TimeFunction: 14 - Selecting Gadgets

Source Document:
mrd document

2.6 - Load

Description:
load dialog box definitions from scopes or from files.

Traces To:
TimeFunction: 16 - Load
Critical Issue: 32 - Loading after tool invoked?

Source Document:
mrd document

Requirements Page
Figure 3

DIALOG BOX QUESTIONS

1. What is your general impression about the look and feel of the layout editor for placing gadgets?

2. Is the gadget palette on the right hand side a comfortable location, or would you like to see it being configurable to left or right.

3. Do you want free flow placement of the gadgets and add constraints to lay them in rows or columns, or would you prefer the row and column constraints to be built in to the system, to allow a grid like layout?

4. What gadgets and sites would you like to be supported most?

5. Would you like to see the code being generated when the dialog box is saved.

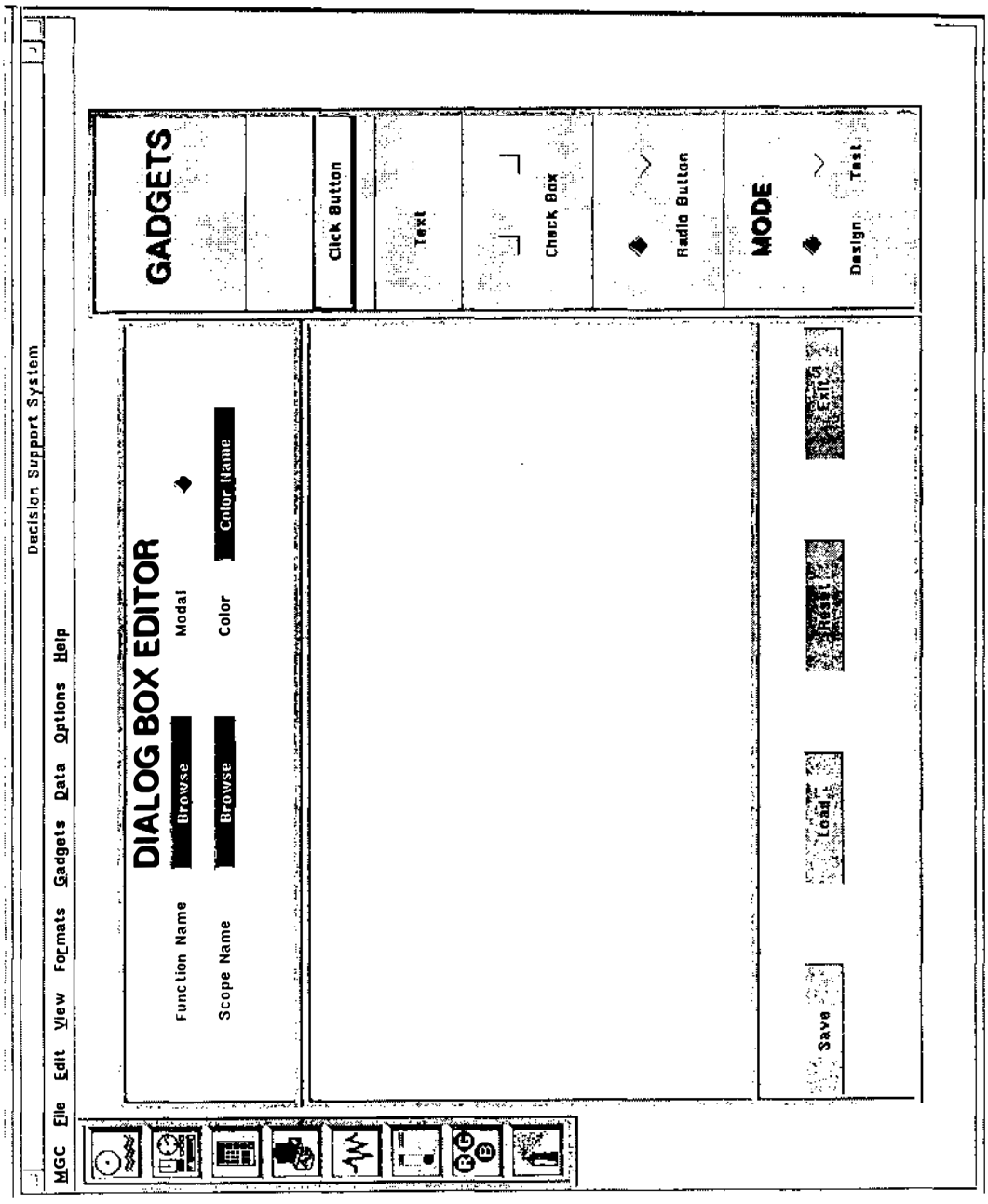
6. Would you like to edit the function source associated with the function that brings up the dialog box.

7. Would like to be able to change existing dialog boxes and save them as new dialog boxes.

8. Any general comments? Other ideas? Potential problem areas?

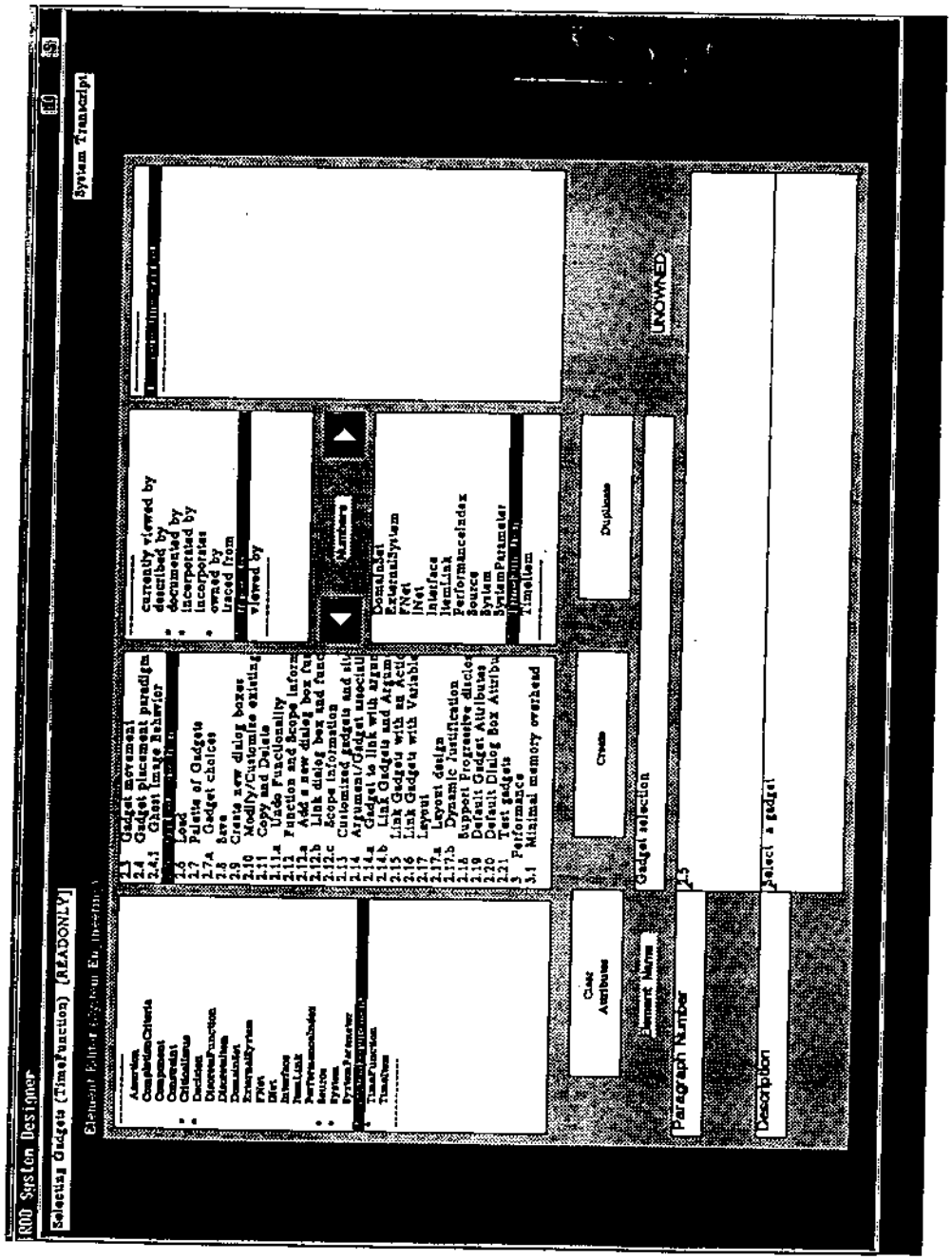
Sample Interview Questions

Figure 4



DSS Prototype

Figure 5



RDD Requirements Screen showing
requirements tracing to behaviors
Figure 6

Figure 7

Behavior Description Example

An Integrated Approach to Software Process Assessment

Joel Henry and Sallie Henry
Department of Mathematics and Computer Science
Dickinson College
Carlisle PA 17013

Abstract

This paper describes a methodology for assessing the software process (both development and maintenance) used by an organization. The assessment methodology integrates the principles of Total Quality Management and the work of the Software Engineering Institute. Assessment results in a well-understood, well-documented, quantitatively evaluated software process. The methodology involves four steps: investigation, modeling, data collection, and analysis of both process content and process output. The investigation step of the methodology gathers information about the activities used by an organization and the environmental factors affecting the process. The modeling step produces a graphical representation of the activities comprising the process. Data collection gathers quantitative process and product data. The analysis step reveals problem areas within the process through statistical analysis and process content analysis. Analysis results determine process improvements.

Biographical Sketches:

Joel Henry is a doctoral candidate in the Computer Science Department at Virginia Polytechnic Institute & State University. He is currently Assistant Professor of Computer Science at Dickinson College in Carlisle Pennsylvania, and has worked on software development projects for Digital Equipment Corporation, Microsoft, and General Electric. Mr. Henry earned the BS and MS degrees in Computer Science from Montana State University in 1985 and 1986 respectively. His current research interests include the software process, software metrics and software engineering methodologies.

Dr. Sallie Henry is Associate Professor of Computer Science at Virginia Polytechnic Institute & State University. Dr. Henry received the BS in Mathematics from University of Wisconsin-La Crosse in 1972, and the MS in Computer Science from Iowa State University in 1977, and the Ph.D. also in Computer Science from Iowa State University in 1979. Her current research interests include software complexity metrics, software design methodologies, and software engineering.

An Integrated Approach to Software Process Assessment

Joel Henry and Sallie Henry
Department of Mathematics and Computer Science
Dickinson College
Carlisle PA 17013

1 Introduction

The development and maintenance of software have yet to employ a body of scientific knowledge and a measurable set of engineering practices similar to those used by such disciplines as Electrical Engineering, Mechanical Engineering and Chemical Engineering. Software Engineering as a discipline is struggling through the early stages of technical development. While statements such as "The software crisis is dead" have been made [HUMW89], a realistic view of the current state of the practice places Software Engineering early in the commercial stage of technical development, as shown in Figure 1. Software Engineering is a craft becoming commercial.

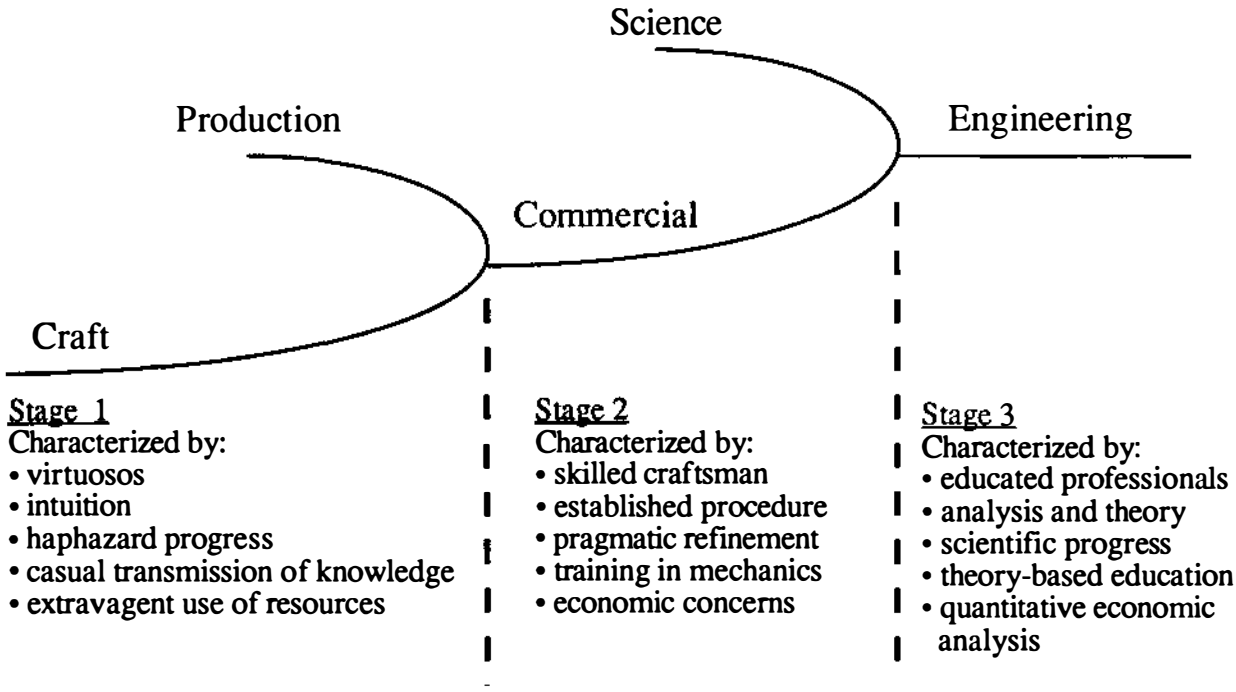


Figure 1. Stages of Technical Development

The need remains for the consistent production and effective maintenance of high quality software, on time and within budget. In a time of reduced budgets and increased competition for software contracts, organizations must improve the quality of the software products they produce and maintain while reducing cost.

The software process, including both development and maintenance, has significant impact on the quality, timeliness and cost of the software products produced [HUMW89] [MOGJ90] [OSTL87] [BOLT91] [CRAS85] [HUM91]. Arthur et. al. substantiate a linkage of process to product quality through the use of software engineering principles such as life cycle verification, early error detection and concurrent documentation [ARTJ86]. This work argues that the effective application of the appropriate software engineering principles imparts quality attributes to software

products. The software process largely determines product cost and timeliness through software engineering management practices [QUIR80]. Huff et. al. have shown the management practices employed in the software process significantly impact product cost and delivery schedule [HUFK86].

2 Software Process Assessment

Software process assessment is the critical first step in software process improvement. Assessment provides information about what type of process exists, how effective the process is and where problem areas within the process exist. Process improvement not based on a thorough assessment is at best a collection of educated guesses, and the results obtained can only be subjectively judged as well.

The two most important approaches to software process assessment and improvement are the Software Engineering Institute (SEI) approach and the principles of Total Quality Management (TQM) [SEIS92]. The SEI Contractor Capability Assessment, Process Maturity Framework, and Capability Maturity Model represent significant advancement in the field of software process research. The principles of TQM, including the use of teams, application of statistical analysis methods, and focus on the customer, have proven successful in other industries.

SEI process assessment, based on investigation of the software process, establishes what activities and tools exist within the process. The SEI assessment method evaluates a process based on substantiated answers to 101 yes/no questions. The questionnaire results evaluate an organization's process on a process maturity framework containing five levels.

The process maturity levels within the Process Maturity Framework form a logical progression of more sophisticated software process descriptions. The activities included in each maturity level are drawn from accepted software engineering practices and proven industrial engineering techniques. Despite these strengths, serious questions exist regarding the grading scheme used in the assessment procedure [BOLT91]. The lack of tailoring within the assessment method is also a problem [HENJ91a].

TQM is a general approach to product improvement focusing on the development process. TQM involves defining the process used by an organization, measuring both product and process, and using statistical techniques to analyze the process [DOD88]. Commitment to improvement throughout the organization and the use of teams are also important ingredients in a successful TQM effort.

The strengths of TQM are reliance on objective measurement, the application of statistical quality control, the commitment of all personnel to quality throughout development [DOD88]. The application of TQM to software development is not clearly defined. The development activities used in a manufacturing environment are well understood (activities such as welding, soldering, painting, etc.) and involve physical components. Software process activities are not as well understood and operate on concepts and abstractions. Exactly what activities a given organization should include in their software process are not clearly defined.

2.1 SEI Process Assessment

SEI Contractor Capability Assessment satisfies the need to discover and understand problems within an organization's software process. The assessment is a balance between an unconstrained search for problems and the urge to prematurely specify solutions.

The objectives of SEI process assessment are to:

- learn how the organization works,
- identify problem areas within the process, and
- enroll it's leaders in the change process [HUMW89].

The SEI Contractor Capability Assessment method is based on the five-level Process Maturity Framework [HUMW89]. The process maturity levels are:

1. Initial - Ill-defined procedures and controls, result in little consistency in managing the software process.

2. Repeatable - An organized and stable software process results in the ability to measure and trace the process and produce accurate estimates of costs and schedules.

3. Defined - A well-defined software process and the collection of tracking data allow verification of compliance with designated standards.

4. Managed - Analysis of quantitative measures of all aspects of the software process results in control of the process.

5. Optimized - Control of the software process provides an opportunity to modify and improve the process using process data.

The process maturity levels form a logical progression of more sophisticated process descriptions. The activities required in each level implement accepted software engineering practices and proven industrial engineering techniques.

SEI process assessments involve the following steps:

1. Secure management commitment to change prior to assessment.
2. Administer the questionnaire and collect the results.
3. Conduct follow-up interviews to substantiate the results of the questionnaire.
4. Give a preliminary presentation of the assessment results.
5. Produce a thorough written report of the assessment results.

The SEI assessment method begins by receiving management commitment to the assessment and subsequent improvement. Without such commitment assessment is difficult if not impossible and subsequent improvements have little chance of implementation.

The second and third steps gather substantiated answers to 101 yes/no questions[HUMW87]. The questions are separated into groups corresponding to process maturity levels 2 through 5. The purpose of each group is to evaluate the organization's process against the required activities for a specific maturity level.

Questionnaire and follow-up data assess the maturity of an organization's software process as having achieved one of the five levels in the Process Maturity Framework. The organization's process improves by adding the activities missing from an organization's process, but required by the SEI Process Maturity Level.

The SEI process assessment detects the presence or absence of the activities required to achieve a specific maturity level is discovered. However, the assessment method elicits much more information than is captured [BOLT91]. Activities not specified within the Process Maturity Framework are not detected. In addition, the assessment method does not specify how to tailor assessment results to an organization. Organizational goals and environmental factors do not factor in the associated assessment, consequently this type of information is not captured [HENJ91a].

2.2 TQM Process Assessment

TQM is a general approach to product quality improvement. TQM is "A cooperative way of doing business that relies on the talents and capabilities of both management and labor to continually improve quality and productivity using teams." [JABJ91] The six principles of TQM are as follows:

1. Maintain customer focus.
2. Focus on the process.
3. Prevent of defects rather than inspect for defects.
4. Utilize the knowledge and expertise of labor.
5. Perform fact-based decision making.
6. Integrate feedback to continuously improve the process.

These general purpose principles are applicable to nearly all types of industry, and particularly to the commercial production of software. The emphasis on process assessment and improvement reflects the belief that quality products result from a quality process.

TQM includes five phases:

- preparation,
- planning,
- assessment,
- implementation, and
- diversification.

Organizational assessment is typically performed through self-evaluation, customer surveys or training feedback. No single structured approach to organizational assessment is specified in TQM.

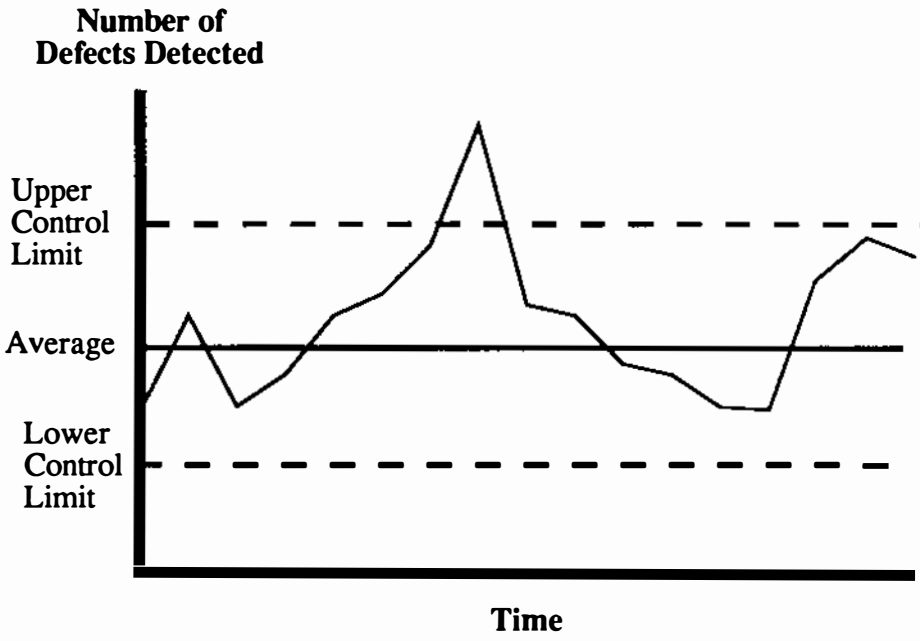


Figure 2. Control Chart

The implementation phase requires process definition, measurement and improvement based on statistical analysis [DOD88]. Process definition documents the process and supports specification of measurement points within the process. Measurement quantifies product quality and process effectiveness at various points throughout the process. Control charts are established to monitor and manage the process. These control charts show the average number of product defects detected following each activity and control limits. The control limits depict acceptable deviations from the average. If the number of product defects exceeds control limits the process is out of control and management alerted. Process improvement is achieved by reducing the average number of product defects. An example of a control chart is shown in Figure 2.

3 Integrated Process Assessment

The SEI and TQM approaches differ significantly The SEI approach can be viewed as a process content assessment while TQM can be viewed as a statistical data assessment. SEI process assessment examines activities within the software process in detail, adding activities or altering the flow of control between activities. TQM also considers the activities within the process but concentrates on improving product measures following each task.

The assessment methodology outlined in this paper integrates concepts from both the SEI approach and the TQM principles. The basis for integrating these two approaches is:

- The SEI assessment specifies the activities comprising the software process. The TQM implementation phase evaluates the effectiveness of the activities.

The specific goals of this integratedassessment methodology are:

1. To discover the activities comprising an organizations software process and the environmental factors affecting the process.
2. To graphically document the process in a form usable by software personnel and amenable to analysis.
3. To gather quantitative process and post-delivery product data.
4. To analyze an organizations process by considering activities present and absent within the process as well as statistical relationships between and among quantitative data.
5. To determine what process improvements are needed.

The assessment methodology outlined in this paper achieves these goals using four steps:

1. Investigate the process.
2. Model the process.
3. Gather data.
4. Analyze the data.

The investigation step of the methodology determines the software process used by an organization and the environmental factors affecting the process. Investigation is the basis for process documentation in the form of a process model. The process model and investigation results support instrumentation of the software process. Quantitative process assessment utilizes statistical analysis techniques.

The methodology utilizes both the SEI approach and the TQM approach to process assessment. Investigation and modeling discover and document the activities used by an organization. Data gathering and statistical analysis discover product-process relationships throughout the software process. Consideration of the activities within the process and statistical relationships is needed to determine what types of improvements are needed and how to implement the improvements.

3.1 The Investigation Method

An effective process investigation method captures both objective and subjective data. A well-structured investigation obtains data from organizational, functional, and behavioral views. Integration of SEI techniques, process audits, and modeling views results in a powerful, multi-step investigation method [HENJ91a]. The process investigation method included in this assessment methodology integrates these three areas of process research.

The process investigation method uses the SEI questionnaire, obtaining data that can be analyzed using statistical techniques. Unlike the SEI Contractor Capability Assessment, the results of the questionnaire are not used as the sole basis for assessing an organization's process. Rather than analyzing questionnaire results to determine what answers must be substantiated, the results are used to develop more general questions used in the follow-up interview. The follow-up questions are created by consideration of process maturity framework goals, software engineering principles and accepted software management practices. These questions investigate the process in a top-down fashion, discovering the activities used in the software process, regardless of their inclusion in the SEI process maturity framework.

The three primary objectives of the investigation method are:

1. To determine the software process used by an organization.
2. To discover the significant factors affecting an organizations software process.
3. To define the methods and techniques used to implement the activities comprising the process.

There are four specific phases to the process investigation method. The first phase in the investigation method determines the staff members to be interviewed. An initial background questionnaire determines whether each staff member is directly involved in the software process, only tangentially involved, or not involved. The SEI assessment questionnaire is used in the second phase to establish the existence of specific activities within the process. The third phase requires follow-up interviews be conducted using more general questions. The purpose of the interviews is to discover activities not included in the SEI Process Maturity Framework. In addition, follow-up interviews establish how and by whom process activities are performed. The fourth phase analyzes the follow-up data to determine the differing views of the process within the organization, environmental factors affecting the process and other process activity information. The analysis phase carefully considers organizational factors as well as investigation information to form a correct, complete view of the software process used by an organization.

This investigation method has been successfully performed with excellent results [HENJ91a]. Information about an organization's software process is captured and important problem areas are discovered. Many of the recommended improvements have been implemented with immediately beneficial results [HENJ91a] [BOYJ91].

3.2 The Modeling Technique

Once a software process is understood, it must be documented to ensure consistent use and effective process improvement. Documenting a software process typically results in clarification of the process and the roles of both individuals and groups within the organization. A documented process also aids in establishing what type of data to collect, where to collect the data, and what the collected data represent. Process documentation, in the form of a model, is critical to process assessment and improvement.

The modeling technique is based on control flow diagrams. Control flow diagrams are well suited to modeling software processes because they clearly depict the interactions among process

activities. Control flow models of a process are constructed according to specific rules based on the concepts of top-down functional decomposition, information hiding, and stepwise refinement.

The control flow diagrams link the Department of Defense development standard to SEI style activity definitions. Contractual requirements require adherence to the DOD development process standards. The SEI process definitions include entry and exit criteria. The process definitions shown in Figure 2 meets these requirements using models based on control flow diagrams.

figure 3

Development of the modeling technique involved four major steps, each intended to impart some of the desired characteristics to the resulting models. First, it was determined that three tiers, each with increasing amounts of detail, would be used to construct process models. Second, the purpose of each tier was defined. Third, model traceability constraints were specified to insure consistent representation of a process between different tiers. Finally, rules governing the generation of a model through all tiers and specific rules for the generation of each tier were defined.

The three tiers used in this modeling technique are:

1. Tier 1 - Phase Tier. The purpose of Tier 1 is to name each phase within the process. For each phase, the products input and output, interphase communication, measurements, and the group responsible for implementing the phase are specified.

2. Tier 2 - Task Tier. The purpose of this tier is to describe what major tasks need to be accomplished to implement a phase. A separate task tier exists for each phase.

3. Tier 3 - Procedure Tier. The purpose of this tier is to specify how the major tasks specified in Tier 2 are performed. The major implementation steps for each task shown in Tier 2 are described in Tier 3. A separate procedure tier exists for each phase.

The purpose of each tier is to progressively elaborate the activities within each phase. The tiers relate to each other in specific and well documented ways. Sets of tiers, such as tier 1, tiers 1 and 2, or tiers 1, 2, and 3, form a self-contained graphical representations of process phases.

The traceability constraints insure that each construct shown in every tier has a well-documented, consistent relationship with associated constructs on the next higher tier of abstraction as well as the next lower tier of detail. A structured numbering scheme permits a task to be traced to the procedures implementing the task. Similarly, procedures are be unambiguously accumulated into the task the procedures implement. Additional traceability constraints and generation rules support verification of the process representation from tier to tier.

A complete listing of the model generation rules is contained in [HENJ92a]. The generation rules provide the resulting models with cohesive tiers. The rules apply the principles of information hiding, functional decomposition, and stepwise refinement to process modeling. In addition, the rules encourage progressive elaboration of detail through tiers 2 and 3.

SEI work in process modeling includes entry and exit criteria for each unit activity. Recognizing exit and entrance criteria are most useful when applied to high level activities, the tier 2 task representations are supplemented with tables. The tables describe entry and exit criteria, task description, and task number for each task.

The modeling technique successfully integrates organizational, functional, and behavioral information into a multi-layered process model. The multiple layers provide visibility into the software process used by an organization, as advocated by the SEI. The addition of measurements to the model depicts what type of data is extracted and where. The data supports statistical process management as required by TQM.

3.3 The Data Gathering Techniques

The data gathering techniques described here specify how to instrument a software process. The data gathering techniques:

- classify the data such that it is amenable to statistical analysis,
- define the data clearly and accurately, and
- require review of data definitions.

What data to collect and where in the process to collect the data depends on investigation results, the process model, and organizational priorities. Process investigation may discover high-priority problem areas within the process which need immediate improvement. The construction of a process model may suggest easily obtainable data about a specific problem within the process. An organization may already recognize a significant, recurring problem within the process needing improvement.

The purpose of this step in the assessment methodology is not to propose a new set of process and product measures but rather to advocate that data gathering be performed. The data is selected by using investigation results, the process model, organizational priorities and previous software measurement work. The data is specified using a defined procedure and a data definition form.

Selection of data to collect should not be performed without a specific purpose and guidelines. The data to be collected must be carefully defined. Haphazard collection of a wealth of data is unlikely to be useful. The assessment methodology specifies data collection from two different approaches using a data definition form.

The two specific approaches to data collection are horizontal and vertical collection. These approaches are shown in Figure 3 and defined as follows:

1. Horizontal - Data about a single activity or product is collected over time or successive process phases to determine characteristics or trends. For example, the number of errors found during successive test phases of a single specification change would be collected.

2. Vertical - Data about a group of activities or products is collected at a single point in time. For example, the estimated number of man-months needed to implement all specification changes approved during Month 2 in Figure 3 would be collected.

These two approaches organize the data acquired in several important ways. Vertical data can be viewed as a snapshot of project data at a particular point in time, making the data suitable for use as status and project review data. Horizontal data can be viewed as process and project tracking data, used to discover trends in the process or project. Classifying data as vertical and horizontal allows the application of statistical analysis techniques. These techniques include analysis of variance, multiple regression, analysis of covariance, and analysis of categorical data.

Prospective data should be tailored to the organization and the process used. Both data definition and tailoring are more easily performed when data is described and collected using a well-organized, stepwise procedure.

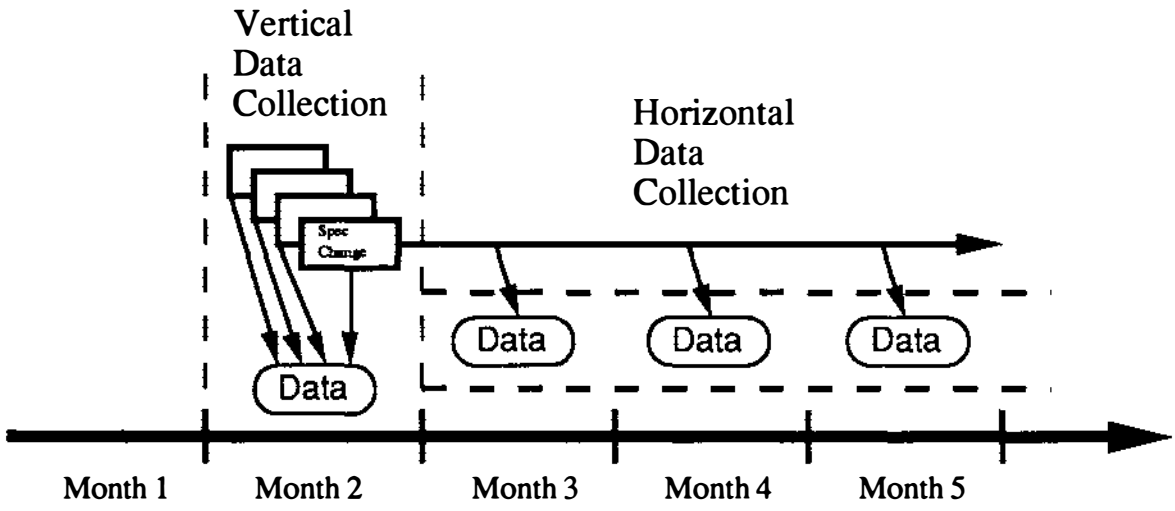


Figure 4. Vertical and Horizontal Data Collection

The procedure advocated in this assessment method uses a data definition form containing the following information:

1. Describe what characteristic the data measures.
2. Define why the characteristic is of interest.
3. Describe how the data will be obtained.
4. Specify what the data will look like in units, frequency, etc.
5. Specify how the data will be validated.

An organization's confidence in the validity of the data, and support in the collection of the data, is crucial to obtaining meaningful data. The data definition form specifies the data to be collected in a logical, understandable form. Data definition forms are reviewed by the personnel involved in the software process. Data definition review involves the organization and is critical to gaining the support of software process personnel.

In order to move Software Engineering toward recognition as a true engineering field formal methods must be applied. Scientific investigation utilizes a structured approach to gathering data about an object or process. A structured approach insures the data gathered accurately reflects the characteristics of interest. The vertical and horizontal data-collection approaches structure data gathering. The data definition form and supporting review insures that the data is clearly specified and completely reviewed. Data gathered using the techniques described here is amenable to many types of statistical analysis methods.

3.4 The Analysis Techniques

The analysis step considers both the content and the effectiveness of the software process. Process content analysis focuses on investigation results and the process model. Analysis of process effectiveness involves the process and product data. Analysis techniques for these two different types of assessment are presented separately.

3.4.1 Process Content Analysis

The information obtained during investigation includes the activities comprising the process and the environmental factors affecting the process. The process model documents the activities

used and the structure of the activities within the process. This information answers the following significant questions:

- What activities are missing from an organization's process?
- What activities should be added to the organization's process?
- Where in the process should these activities be added?

Content and structure of the process are compared with the detailed and extensive Capability Maturity Model for Software(CMM) produced by the SEI [SEI91]. The CMM describes the activities appropriate for process maturity levels 2 though 5. The activities are classified into major categories such as Project Management, Process Management, Software Quality Assurance, Software Configuration Management, etc. While the content and interpretation of the CMM continues to evolve, it does represent a significant repository of accepted software engineering principles and project management practices. The purpose of this comparison is not to force an organization to adhere strictly to the process defined in the CMM, but rather to discover missing activities beneficial to the organization. Previous experience in this area has shown the major benefit of this type of analysis is in the utilization of accepted principles and practices by an organization [HENJ91a] [HENJ91b].

Environmental factors are significant considerations in the assessment of an organization. These factors often explain why process activities exist, don't exist, or why a process works in a particular fashion. Previous assessment of small organizations discovered that very different organizational goals significantly impact the software process used [HENJ91a]. For example, the emphasis on error prevention is very different if an organization produces proof of concept software rather than performing long-term maintenance on an established product. Careful consideration of environmental factors allows the appropriate software engineering principles and practices to be selected for inclusion in a software process.

3.4.2 Statistical Analysis of Process Output

Software project managers and division managers typically lack quantitative data regarding project status, progress, and product quality. Given this lack of objective data, the affect of man-power loss, specification changes, error-rates, and other process factors is estimated using rules of thumb, experience, or "gut feel." Such estimates leave a large margin for error. More mature engineering disciplines rely on quantitative data for determining project status, progress, and product quality. Software Engineering is in need of quantitative project management based on objective data and statistically valid relationships between and among data.

Two specific types of relationships are of interest. First, relationships among process data are examined to determine where statistically valid prediction can be performed and where significant differences in process data values exist. Second, relationships between process data and post-delivery product data are examined to determine if relationships exist, what type of relationship exists and what process data is statistically valuable in predicting the post-delivery product data.

Some of the questions which are empirically evaluated are:

- Can future process and product characteristics be predicted from process data?
- What relationship exists between process data and post-delivery product data?
- What is the relationship between projected data and actual data?
- What is the cumulative affect of horizontal data on the post-delivery product?
- What is the combined affect of process activities on the post-delivery product?

Data acquired during the software process and following delivery of the product are subjected to statistical analysis. Analysis of process output employs accepted statistical methods, such as

multiple regression, analysis of variance and covariance, and analysis of categorical data. Multiple regression is employed to determine what type of relationship, if any, exists among process data and between process data and post-delivery data. Data is combined by time period or logical grouping, then compared using analysis of variance techniques to detect significant differences.

Statistical analysis satisfies the need for qualitative process evaluation and is the basis for fact based decision making. The effectiveness activities present in the process is determined using statistical analysis. In addition, the impact of process activities can be shown using statistical techniques. Fact based decision making is based on valid statistical predictions. The quantitative impact of process activities and trends on the delivered product are also used in making fact based decision.

4 Results

Implementation of the integrated assessment methodology is underway within a large military contractor. The contractor produces both hardware and software for a radar-based combat weapons system. This system is one of the longest running military contracts existing today, employing thousands of people and costing billions of dollars.

The investigation step is complete. Dozens of interviews were conducted involving such diverse groups as quality assurance, configuration management, program management, subcontractors, testing, and the customer auditing agency. The input of all these groups is critical because each group has a unique perspective, and distinct and sometimes conflicting goals. An estimated SEI assessment was also performed [HENJ91b].

The modeling step is also complete. The models underwent extensive review throughout the organization's development community. Complete documentation of the development process is based on process models constructed using the modeling technique described in section 3.2. The graphic models are supplemented with task entrance and exit criteria, and specification of the applicable computer program standards, work instructions, and directives.

The data collection step is ongoing [GESD91]. Several versions of the software system, referred to as baselines, are developed concurrently. Process and product data have been collected on different baselines at various points within the process.

Process content analysis and statistical data analysis are also ongoing. Initial process content analysis is complete, and is detailed in [HENJ91b]. Statistical analysis of requirements volatility and defect detection has taken priority because of the impact these two categories have on cost and quality.

4.1 Results of Process Content Analysis

The investigation and modeling steps clarified the actual process used to create computer programs and suggested process improvements. An improvement plan was produced using investigation results and process definitions. The improvement plan uses a global view of the development process to propose improvements. A narrowly focused, question-based improvement plan cannot be successful in a large organization producing a large scale (several million source lines of code) software product.

The improvement plan targets three areas, training, requirements tracing, and project management information. Training software professionals is an ongoing activity within the organization, only two new courses need be added to satisfy SEI maturity level three requirements. Requirements tracing is a labor intensive task at this time. Tools are under review to supplement

manual requirements tracing activities. A project management database is under development, an will include programming, management, metrics, and change control data.

Process content improvements focus on the existing process and the SEI maturity framework. Improvements integrate into normal operating procedure with minimal disruption of daily operations and maximum benefit. Evaluation of improvements and feedback is ongoing.

4.2 Results of Statistical Analysis

Process and product data captured to date display a number of interesting relationships. For example, a contingency table was constructed relating the number of functional upgrades impacting a program unit and the number of errors found in the program unit. The Chi-square value obtained from this contingency table is 30.9, showing a strong association between these data items. This information prioritizes testing and suggests where code walkthoughs are most beneficial.

Analysis of specification change data and the final integration testing of all computer programs comprising the system showed no significant relationships. The number of tests failed and errors detected is very small, leading us to conclude the test phase is not effective in detecting errors. Investigation results support this conclusion.

A predictive equation for the total man-months needed to implement a functional upgrade was produced using multiple regression on upgrade and Preliminary Design Review data. The R-square coefficient of this equation is .92.

Statistical analysis highlights the need for different data and new techniques as well as discovering data relationships. Data and statistical analysis points out additional types of data needed to discover other relationships of interest. We are also considering analysis techniques from the field of artificial intelligence and operations research. While analysis results will be company confidential, the effectiveness of the various analysis techniques employed are applicable to other organizations performing process assessment.

5 Conclusions

The existence of a structured, validated assessment methodology is the foundation for process improvement. In fact, it can be argued that process improvement is only as effective as the assessment it is based on and evaluated by [HUMW89]. The assessment methodology described here is a very effective approach to continuous process assessment and improvement.

The assessment methodology continues to evolve. Process documentation is refined, and the modeling technique updated where appropriate. Analysis of both process content and data indicates additional data needed and modifies existing data definitions. Statistical analysis highlights activities within the process where the greatest leverage can be gained on process and product improvements.

An approach to process assessment integrating TQM and SEI approaches provides insight into both the content and the effectiveness of an organization's software process. Process content improvement places process activities within the software process. Statistically valid relationships among and between the data evaluate the effectiveness of process activities. The results can be used in project management, process management, product quality evaluation and risk assessment. The use of quantitative data in each of these areas will move the field of Software Engineering toward a more advanced stage of technical development.

REFERENCES

[ARTJ86] Arthur, J. D., Nance, R. E., and Henry, S. M., *A Procedural Approach to Evaluating Software Development Methodologies: The Foundation*, Technical Report SRC 86-008, Systems Research Center and Department of Computer Science Virginia Tech, 1986.

[BOLT91] Bolten, T.B. and McGowen, C., "A Critical Look at Software Capability Evaluations," *IEEE Software*, July 1991.

[BOYJ91] Boyle, J., Executive Vice President, Video Lottery Consultants Inc., Bozeman Montana, personal communication, December 7, 1991.

[CHRA87a] Chruscicki, A. J., "Software Reliability Assessment," Rome Air Development Center, Griffiss AFB, 1987.

[DOD88] "Total Quality Management Master Plan," Department of Defense, August 1988.

[GESD91] "AEGIS Computer Program Metrics Plan," General Electric Corporation, Government Electronic System Division, March 1991.

[HENJ91a] Henry, J. E., Keenan, S., Keenan, M. and Henry, S., "An Assessment Method for Small Organizations," *Proceedings of the 1st International Software Quality Assurance Conference*, October 1991.

[HENJ91b] Henry, J. E. and Dublanica, W., "Improving the AEGIS Computer Program Development Process to Achieve SEI Maturity Level 3," General Electric Corporation, Government Electronic System Division, November 1991.

[HENJ92a] Henry, J. E., Dublanica, W., Kubeck, J. and McMullen, B., "A Process Modeling Technique and Application Results," *Proceedings of the Software Tools and Techniques Symposium*, March, 1992.

[HUFK86] Huff, K. E., Sroka, J. V. and Struble, D.D., "Quantitative models for managing software development processes," *Software Engineering Journal*, January 1986.

[HUMW87] Humphrey, W.S. and Sweet, W.L. "A Method for Assessing the Software Engineering Capability of Contractors," Software Engineering Institute, Carnegie Mellon University, September 1987.

[HUMW89] Humphrey, W.S., *Managing the Software Process*, Addison-Wesley, 1989.

[HUMW91] Humphrey, W.S., "Software Process Improvement at Hughes," *IEEE Software*, July 1991.

[JABJ91] Jablonski, J., R., *Implementing Total Quality Management: An Overview*, Pfeiffer, 1991.

[MOGJ90] Mogilensky, J., "Approaches to Upgrading Software Process Maturity," *Washington Ada Symposium*, June 1990.

[OSTL87] Osterwell, S., "Software Processes are Software too," *Proceedings of the 9th International Conference on Software Engineering*, 1987.

[QUIR80] Quinnan, R.E., " A programming process study," *IBM Systems Journal*, Vol. 19, No. 4, April 1980.

[SEI91] "Capability Maturity Model for Software," CMU/SEI-91-TR-24, Software Engineering Institute, Carnegie Mellon University, August 1991.

[SEIS92] Siegal, S., "Why We Need Checks and Balances to Assure Quality," *IEEE Software*, January 1992.

ATTAINING LEVEL 2 & 3 ESTIMATION PROCESSES IN AN R&D ENVIRONMENT

Gordon Wright
Galorath Associates, Inc.
P.O. Box 90579
Los Angeles, CA 90009

Abstract

The charter of the Software Engineering Process Office (SEPO) Naval Command, Control and Ocean Surveillance Center (NCCOSC) R&D Division (NRaD) is to improve the software development processes to a Software Engineering Institute Level 3 and above. Described are SEPO's methods and progress in establishing a software estimation process at NRaD and a brief description of the process.

Biography

Gordon Wright, prior to joining Galorath Associates, Inc., was one of the original members of the Software Engineering Process Office at NRaD. He has over 20 years experience in software specializing in cost and budget forecasting models. Mr. Wright has performed numerous studies and extensive research in the field of software size, cost, and schedule optimization in both the government and industry. He has developed software estimation processes and conducts software estimation workshops. He has developed simulation techniques that perform cost tradeoffs to increase productivity of manufacturing processes. Mr. Wright holds a B.A. in Mathematics from San Jose State University. He is currently Secretary of the REVIC Users Group and the President of the San Diego Chapter of the Society for Software Quality.

Attaining Level 2 & 3 Estimation Processes In An R&D Environment

Establishment of the Software Engineering Process Office - An assessment of the software engineering practices at the Naval Command, Control and Ocean Surveillance Center (NCCOSC) RDT&E Division (NRaD), a U.S. Navy R&D laboratory, was conducted by the Software Engineering Institute (SEI) in early 1988. The purpose of the assessment was to determine NRaD's level of maturity within the five levels of software maturity defined by the SEI. The assessment determined that NRaD was a level 1 organization where level 1 is the least mature and level 5 is the most mature level.

The summary report that SEI submitted to NRaD as a result of the assessment included nine recommendations. The first and second recommendations were for NRaD to

- put into place formal procedures for estimating cost, size, schedule.
- establish a Software Engineering Process Group to serve as a focal point for software process improvement.

As a result of these recommendations the NRaD Software Engineering Process Office (SEPO) was formed in late 1988. SEPO's charter is to improve the software development processes from a level 1 on the SEI Maturity Model to a level 3 and above. SEPO became fully staffed in May 1989 with five full-time people and one rotator. The rotator is a person who transfers into SEPO from a line organization temporarily for a period of three months to one year and works as a resident member of SEPO.

Since its inception, SEPO has concentrated on establishing processes for software estimation, project tracking metrics, formal inspections, and software capability evaluations. SEPO has also established a software engineering resource center which includes an Ada resource center, text and video library, and a repository of Computer Aided Software Engineering (CASE) tool data.

The NRaD Environment - NRaD's primary mission is to develop and evaluate new technologies and implement new and improved C2, C3, and Ocean Surveillance Systems. There are approximately 1600 engineers and scientists and at any one time there are 100 or more software projects. Some of these are small one person, short duration projects while others are several billion dollars and span several years.

One of SEPO's first tasks was to determine how NRaD currently develops software. Any processes SEPO developed would have to address the current development methods. SEPO developed a survey that went to all the departments in NRaD.

It was quickly determined that NRaD projects are conducted under various development approaches, e.g., waterfall, evolutionary acquisition, spiral and prototyping. Many projects utilize a standard waterfall type approach under delivery order or task order contracts. It was also determined that there is considerable emphasis on prototyping.

SEPO's Software Processes - Lack of a software estimation process was identified by SEI as the primary reason NRaD was not a Level 2 organization. However, SEPO conducted its own survey in early 1989 and determined that NRaD was a solid level 1 organization. Since then SEPO has identified 14 software processes to be developed. To date three processes have been completed: 1) Software Size, Cost and Schedule Estimation Process; 2) Project Tracking and Control Process and; 3) the Formal Inspections Process.

These processes are implemented in individual projects by the appropriate SEPO team member, i.e., the SEPO member who has the expertise in Formal Inspections invests time working directly with the project team in conducting Formal Inspections. The SEPO team member becomes less involved as projects become proficient and confident in the process.

Developing the Process - Moving an organization's estimation processes from SEI level 1 to levels 2 and 3 is a multi-faceted process. Improving the estimation processes at NRaD has been through the documenting of a formal estimation process, establishing and chairing the Cost/Size/Schedule Estimation Process Working Group (CEPWG) and formal training which highlights the connection between metrics and estimation.

The first step, documentation of an estimation process, must address the specific needs and circumstances of the individual organization. Generalized processes are now well documented in the literature. However, each organization is different in how they are organized and the type of projects and development models that they use. Some organizations have a team of people dedicated to doing nothing but cost estimating and tracking while other organizations have no such dedicated function and require that individual projects assume the full responsibility of performing and tracking their own estimates. This latter scenario characterizes NRaD.

Software Cost Estimation Practices at NRaD - Some of the questions on the survey dealt with estimation practices currently in place. Most projects admitted that they did not use any formal method or tools to estimate size, cost or schedule. Most project leaders do estimates because they have an immediate requirement. Once the requirement is satisfied, the tool is not used until the next estimate is

absolutely necessary. Then, project managers again want an easy-to-use tool, assuming its results are credible. One of the goals of establishing an estimation process is to move estimates from the realm of "fire drills" to a routine and periodic management function.

The methods used to implement good estimation practices included obtaining representative estimating tools, disseminating information about estimation tools, establishment of the Cost/Size/Schedule Estimation Process Working Group (CEPWG), sponsoring occasional one day, on-site symposiums for estimation tools, and training.

One of the first things SEPO did to introduce NRaD software project personnel to estimation practices and methods was to hold a one day symposium on cost models. Estimation tools were available for hands-on demonstrations - three commercially available tools; a public domain version of COnstructive COst MOdel (COCOMO); and a Navy/Air Force sponsored tool available to DoD agencies. The vendors of the commercial tools also gave presentations on their tools. Because of the high level of interest at the symposium SEPO went ahead and obtained three tools, REVised Intermediate COCOMO (REVIC), System Evaluation and Estimation Resources (SEER), and the Software Architecture, Sizing, and Estimating Tool (SASET).

REVIC is a public domain tool developed by Air Force Major Ray Kile as part of his reserve duties. REVIC was obtained because it was a good implementation of the popular COCOMO. COCOMO is the best documented estimation model and is used extensively throughout the United States and Europe.

SEER was selected because it is a good non-COCOMO, commercially available tool. SEER was developed by Galorath Associates, Inc. and utilizes estimation algorithms developed by Dr. Randall Jensen of Hughes.

SASET was obtained because, in addition to being a good comprehensive tool, it was developed for the DoD and is free to DoD agencies. SASET was developed by Martin Marietta Denver Aerospace Corporation under contract to the Navy Cost Analysis Center.

The Estimation Process - The basic process itself can easily be stated in three basic functions: develop a size estimate based on experience; use the size estimate as input to two independent estimating methods to derive cost and schedule estimates; track the actuals and periodically revise the estimates. However, implementing the process is not as easy. There is always resistance to change as well as inertia. People are slow to adopt new methods and ideas, even when they believe in them. Introducing the process to project personnel has consisted of three primary steps; provide them with a quick overview of the process; give them a quick tutorial of an easy-to-use estimation tool; and provide follow-up support.

The estimation process consists of the three basic steps summarized above. These steps are elaborated upon during the

initial meeting with project personnel. Elaboration of these steps provides project personnel with the basics of how to develop initial estimates and track the estimates vs. actuals. The basic process of developing an estimate is summarized in figure 1.

The process shown in figure 1 is expanded during the initial meeting to include the following points

- Develop a Work Breakdown Structure early
- Estimates should be developed by two or more people
- Two methods of estimation should be used
- Develop a range of estimates, low, most likely and high
- Inspect/review the estimates
- Track and update

Software Estimation File - One of the key elements of the process includes establishment of a Software Estimation File (SEF). This, very simply, is the documentation and retention of the work that goes into and affects the estimates. It is more than just updating the Software Development Plan. The SEF contains any information affecting the estimates as well as the estimates themselves. The SEF contains cost related metrics and records of cost risk analyses. The contents of the SEF consists basically of all estimates and estimate related data. The contents of the SEF and their organization is shown in Table 1.

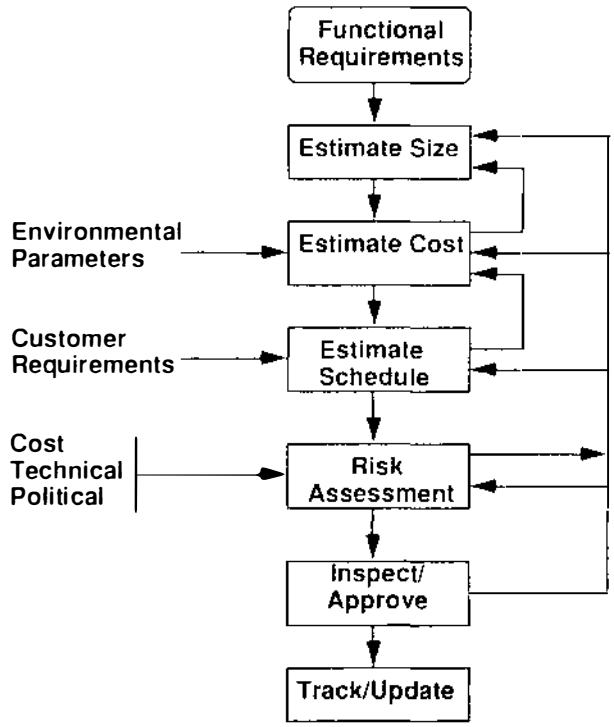


Figure 1. Summary of Software Estimation Process

Project Estimate History Form - The importance of tracking the initial estimates and developing revised estimates is stressed. Initial project estimates are often never seen nor referred to again. This is the result of not documenting the estimates and supporting data in a formal, standardized manner. Initial estimates should be constantly reviewed and revised estimates generated whenever there is any change in direction. New estimates should be developed at least monthly and prior to all major program reviews, i.e., PDRs, CDRs, etc.

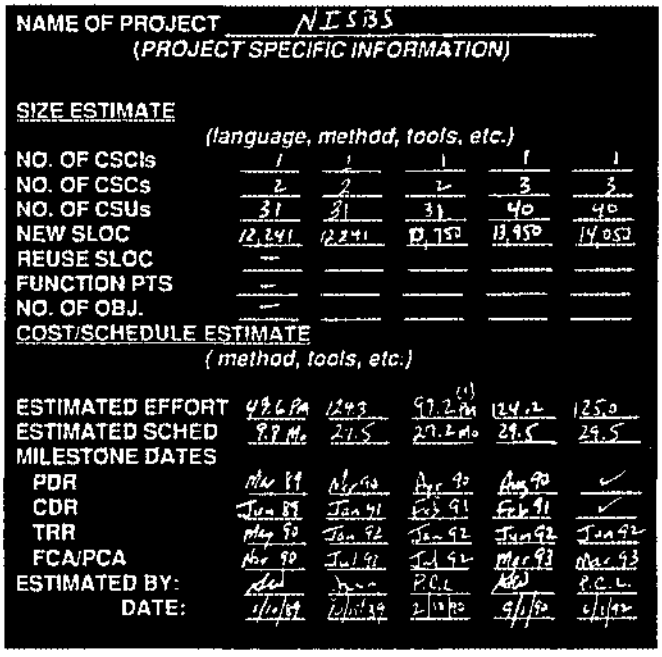
The Project Estimate History Form provides a simplistic format to record top level historical project estimate data. An example of the form filled out for an actual project is shown in figure 2. Each column of the form represents an estimate at a different point in time. The initials of the estimator are entered at the bottom of each column along with the date of the estimate. This provides the project manager a top level summary of how the estimates of size, cost and schedule have changed over time. Because schedule slips are a constant problem, the planned and actual 2167A project review dates are also entered.

TABLE 1

Software Estimate File Format

TABS	CONTENTS
S/W Development Plan	Description of Project - Application, language, sponsor, mode of development, - Schedules
Work Breakdown Structure	Breakout of software tasks
Memos	Copies of relative material - Basis for size estimates - Schedule alterations - Project redirection
Misc. Project Data	Basic assumptions
Summaries	Summary sheets of all completed estimates
Risk Analyses	Comparison data - Descriptions of scenarios - Risks and contingencies
Detail Estimates	Detail outputs from models

The project whose data is shown here is used for the case study in the NRaD SEPO Software Project Management course. The data entered over time effectively illustrate how estimates for size, effort and schedule can vary during development. The constant change in schedule is also highlighted and saved for future reference. Schedule variations of the magnitude shown are the rule rather than the exception.



NAME OF PROJECT NISIS

(PROJECT SPECIFIC INFORMATION)

SIZE ESTIMATE (language, method, tools, etc.)

NO. OF CSCIs	1	1	1	1	1
NO. OF CSCs	2	2	2	3	3
NO. OF CSUs	31	31	31	40	40
NEW SLOC	12,241	12,241	12,750	13,950	14,050
REUSE SLOC	—				
FUNCTION PTS	—				
NO. OF OBJ.	—				

COST/SCHEDULE ESTIMATE (method, tools, etc.)

ESTIMATED EFFORT	92.6 PM	124.3	97.2 PM (1)	134.2	125.0
ESTIMATED SCHED	9.7 Mo.	21.5	29.2 mo	29.5	29.5
MILESTONE DATES					
PDR	Nov 89	Nov 90	Apr 90	Aug 90	✓
CDR	Jun 89	Jan 91	Feb 91	Feb 91	✓
TRR	May 90	Jan 92	Jan 92	Jun 92	Jun 92
FCA/PCA	Nov 90	Jul 92	Jul 92	May 93	Dec 93
ESTIMATED BY:	[illegible]	[illegible]	P.C.L.	[illegible]	P.C.L.
DATE:	1/6/89	10/1/89	2/13/90	9/1/90	6/1/92

Figure 2. Sample Project History Form

Software Estimate Worksheet - In addition to a standardized format for recording the estimate summaries, the SEF also includes a worksheet format to use when doing cost "what if" comparisons. When people start doing "what if" comparisons, they often rapidly go through various scenarios and do not save individual cases. The worksheet, shown in figure 3, will not satisfy all individual needs or preferences for informal notes, but does provide a generic device for historical record keeping purposes. Alternative scenarios typically consider varying key parameters such as size estimates, Requirements Volatility, or Personnel Experience. Saving the alternative scenarios will often prevent redoing some comparisons at a later date.

Cost/Size/Schedule Estimation Working Group (CEPWG) - The role of SEPO is that of facilitator, i.e., to help project personnel acquire the skills and learning necessary to improve the processes within their projects and organizations. The most effective method of disseminating information about estimation methods, practices and tools has

been the working group which in reality has evolved into a regularly scheduled workshop. To date 18 CEPWG meetings have been held. Meetings are held every six weeks with an average attendance of 12-15 people. Since the first meeting in November 1989 over 75 people have attended at least one meeting.

SOFTWARE COST ESTIMATION WORKSHEET
PROJECT:
DATE:

RUN NUMBER	MODEL USED	CSCI	SLOC	MODEL PARAMETERS (NOTE SIGNIFICANT FACTORS USED)	CALENDAR MONTHS	TOTAL COST

Figure 3. Sample Software Estimate Worksheet

Initially the attendance was limited to NRaD personnel. After much discussion however, it was decided to allow NRaD contractors to attend. The attendance is usually divided equally between NRaD personnel and contractors. The meetings usually consist of one or two presenters who address their project estimation experiences. However, the meetings do not always focus on estimation.

CEPWG Project Related Topics of Discussion - Guest presenters usually consist of project personnel who describe their use of the estimation tools on specific projects. They discuss their approaches, assumptions, problems and results. They also discuss their level of confidence, before and after, in the tool(s) that they used. SEPO personnel also provide overviews of how an estimate was developed for specific projects along with demos and discussions of how models may treat some aspect of the software development environment. Summaries of some of the presentations to date follow.

REVIC for Three Small Delivery Order Projects - This presentation addressed use of REVIC for estimating software developed under task order or delivery order contracts. This presentation brought some key issues to the surface. The most prevalent issue was how

government agencies often have a pre-defined amount of money and need a new software package or a modification to an existing package. The contractors sign up for the work for the funding available because if they don't, somebody else will. It was generally agreed that contractors often resort to uncompensated overtime to get the job done. It was also agreed that these types of projects seldom follow formal documentation standards, good configuration management practices or have a quality assurance plan.

REVIC/SEER for Alternative Program Development Options - The application of the REVIC and SEER estimation models to evaluate cost tradeoffs of program options for a next generation wargamming system was presented. Four basic program options were under consideration, with each major option having a couple of sub-options. The options included various degrees of new code development vs. use of existing code on new platforms, and the impact of bringing in a contractor unfamiliar with the project. The presenter expressed concern in getting the models to converge on an estimate. However, he emphasized how the use of the models provided a credible basis of relative comparison of the cost impacts of the various options.

SASET Function Based Estimate - This presentation highlighted the use of two models to derive an estimate for a project in the concept exploration phase. Since this project was in the concept exploration phase, a well defined set of requirements was not available. To establish a rough estimate of size (source lines of code), the SASET model's historical data base was queried for functions similar to those that were to be developed. The resultant size was used to develop estimates with SASET and REVIC. The estimates reflected different levels of Ada programming experience and different levels of requirements volatility. Estimates showed potential costs of $7M to $11M (see table 1) vs. the project manager's original estimate of $700K.

Recode CMS-2 to Ada & Rehost - A contractor described how he had applied REVIC to a prototyping project. His first REVIC estimate was almost 150% higher than the effort actually expended. However, after re-evaluating the values of the environmental parameters and also obtaining a truer picture of the number of personhours per month actually expended per

TABLE 1

Technology Demonstration Project
Data Fusion/Neural Net/Quick Response

PRELIMINARY ESTIMATES

SCENARIO		SASET	REVIC
Ada Exp	Reqs Vol		
Y	N	$8.8M	$7.3M
Y	VH	9.9M	10.0M
VL	N	8.9M	8.3M
VL	VH	10.4M	11.4M

person, REVIC came very close to the actual. Also, the project manager had originally set the REVIC parameter values too conservatively.

An EXCEL spreadsheet program was used to perform risk analysis based on the REVIC parameters. The spreadsheet allowed the user to vary any of the parameters and then observe the resultant sensitivity to cost. The contractor stated that he felt pretty comfortable with REVIC now and plans to use it for future software project estimates. One of the primary benefits he felt was the visibility the model gave him into the effects of the development environment, vs. just the size, on project costs and schedule.

REVIC/SoftCost-Ada for One Man Ada Project - A NRaD software engineer described his use of REVIC and SoftCost-Ada to develop cost and schedule estimates for a small project estimated to be 2,900 Ada lines of code (LOC). He described how he doubled his size estimates to get around SoftCost-Ada's minimum size requirement of 5,000 LOC. Then by applying the power curve, he arrived at an estimate he could divide by two. Since the REVIC results were consistently higher by the same degree than the SoftCost-Ada results, he felt comfortable using the combined results of the two models to arrive at his project's estimates.

Basic Estimation Metrics Tracking - SEPO has presented a format to collect basic cost tracking information. The Project Estimate History Tracking Form, available on e-mail, provides a simple format to record the original cost, size and schedule estimates, and the actuals during the project life cycle. The feedback on the form will help to establish a NOSC

project cost historical data base. The format consists of the following information

- Basic project information
- Estimation method
- Milestone dates
- Estimated CSCIs/CSCs vs. actuals
- Original new SLOC estimates vs. revised estimates and actuals
- Original reused SLOC estimates vs. revised estimates and actuals
- Original cost and schedule estimates vs. revised estimates
- Original estimated page counts vs. revised and actuals

CEPWG Topics Related to Cost Model Parameters - There is as much interest in the generic issues of software estimation as in specific project experience. In addition to project related presentations such as those outlined above, presentations often address cost related issues in general: the differences in specific models; the variation of parameters from model to model; highlights of cost related conferences; demonstrations of models are provided; as well as presentations dealing with theory such as Halstead's metrics.

Some of the specific topics to date have included impact of changing schedules on cost; uncompensated overtime vs. cost/schedule; cost of documentation; impact of design for reuse; basic cost risk tradeoffs; cost of CASE tools; and multiple CSCIs vs. one large CSCI. Many discussions have revolved around the merits and disadvantages of specific tools. The use of estimation tools has been very effective in demonstrating areas of high cost and schedule risk, e.g., the severe impact that code growth can have on a project's development cost. The discussions and demonstrations of tools have shown how the aggregation of two or more erroneous assessments early in the project can have disastrous effects on a project. One recent example demonstrated the cost risk when estimates of size and experience were both overestimated (figure 4.)

Estimation as Part of the Metrics Process - Software estimates need to be revised on a regular basis. SEPO has been assisting projects in specifying metrics in their statements of work. Cost, schedule and size metrics can be tracked from the very beginning of a project all the way through development. The specific metrics desired from the contractor must be stated in the statement of work. Figure 5 shows the metrics related entry from a statement of work recently issued for a small prototype project.

The greatest obstacle to overcome in establishing

processes is to get "buy in" from the people involved. Acceptance cannot be mandated. People will drag their feet or find ways to ignore mandates. Getting people to do

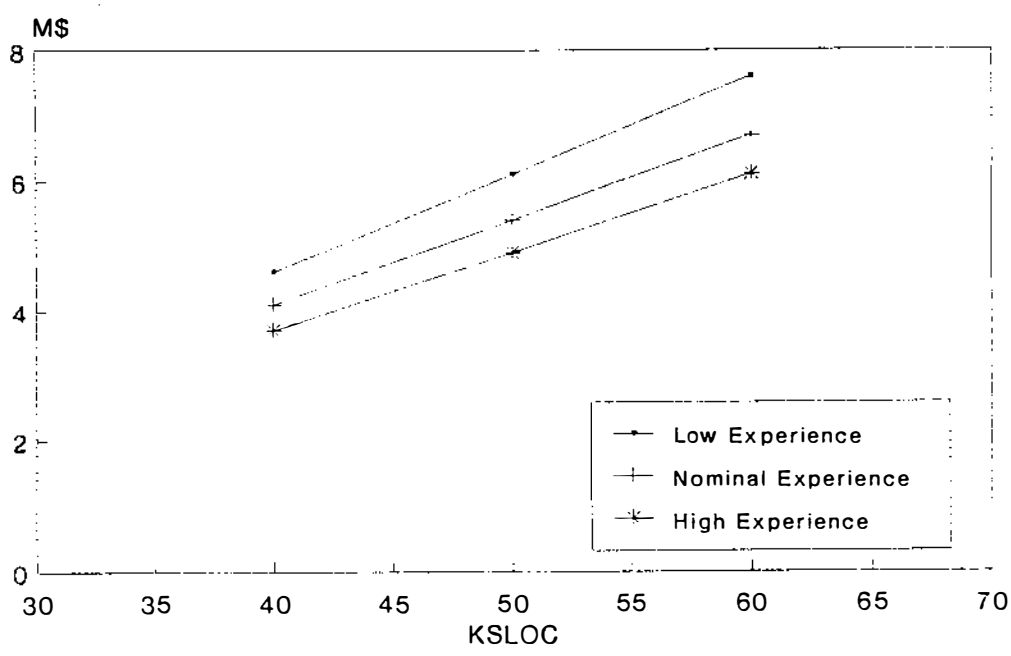


Figure 4. Risk Profile of Cost vs. Size and Experience

3.1 Task Management: The contractor shall provide a detailed tasking plan within 30 days after task initiation. The plan shall include a detailed task element breakdown, labor categories and allocation to task elements, intermediate milestones, and schedule. The task elements shall be defined to a level of effort not exceeding 4 weeks duration. The contractor shall also maintain a management database from which monthly reporting information can be derived. The database shall contain the following information:

a. WBS element status showing progress to date, cost variance and schedule variance.

b. Software size, measured in estimated lines of source code, for each CSCI, CSC, and CSU, as each of these elements is defined. The lines of code shall be categorized as "new," "reused," or "COTS," and "modified."

c. Document information including:

- **Name and Designation**
- **Status (not started, in process/% complete, in review, baselined, etc.)**
- **Date (planned, actual)**
- **Size (pages, disk file size)**
- **Version changes (number of lines changed, added, deleted, as defined by disk file compare).**

d. Engineering Change Proposal/Specification Change Notice Status:....

e. Staffing Metrics by labor category, total labor expended for the reporting period.

Figure 5. Metrics in a Statement of Work

an initial project estimate was easy. They usually came for help when they had to do it. However, getting them to reestimate on a regular basis was not successful. People have plenty of fires to put out and will not relinquish time to something that appears to be a "nice to have" thing. However, when metrics requirements were put into statements of work (SOWs), the necessity for updating estimates became more obvious. SOWs are specifying effort, size, requirements volatility, documentation, and progress metrics. When size is reported and is changing on a monthly basis, the value in updating the project estimate becomes evident.

A recently completed small 1 year project to recode Pascal/FORTRAN code to C and rehost it from a UNIVAC 1100 to an Intel HyperCube (IPSC/860 Touchstone) implemented effort, defects, size and cost metrics. REVIC was used to evaluate the cost of the project. The project used an on-line distributed defect tracking system to track two major categories: How Was Personnel Time Spent; and Which Program Was Worked On.

The project recorded hours per 13 major tasks for all 19 people on the project. The project was considered small because many of the 19 people worked only part time on that project. The 13 tasks were mapped into the six major tasks defined in the REVIC model. The estimate produced by the original REVIC estimate was very close to the actual total effort as shown in figure 6. The total size of the project, however, turned out to be much larger, as shown if figure 7. Even though there is a discrepancy between the original size estimates and the actuals, the project manager is pleased with the results of the estimation and metrics processes and feels confident that this experience can be used to calibrate cost models for the next phase.

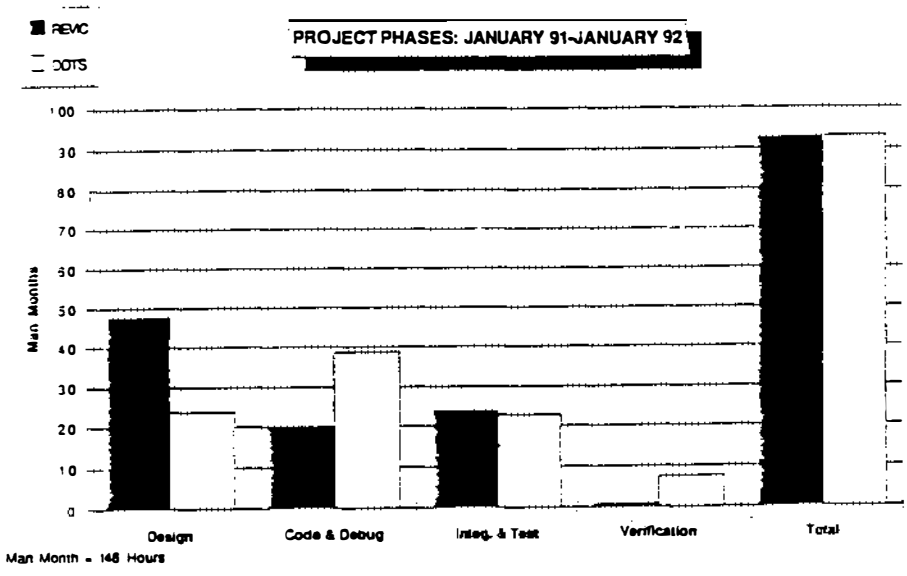


Figure 6. Estimates vs. Actuals by Phase for Small Project

Training - The third step in institutionalizing the process is training. Two SEPO personnel attended the SEI's Software Project Management for Instructors course in early 1991. Subsequently SEPO developed a 1 week Software Project Management (SPM) course which delineates sound software processes from the very early stages of a project (Concept Phase) through the test phase. Over 40% of the course is devoted to exercises.

The purpose of the training is twofold. The SPM course not only addresses the mechanics of good software development processes, i.e., how to do an estimate, formal inspections, use of tools, etc., but also addresses the need and

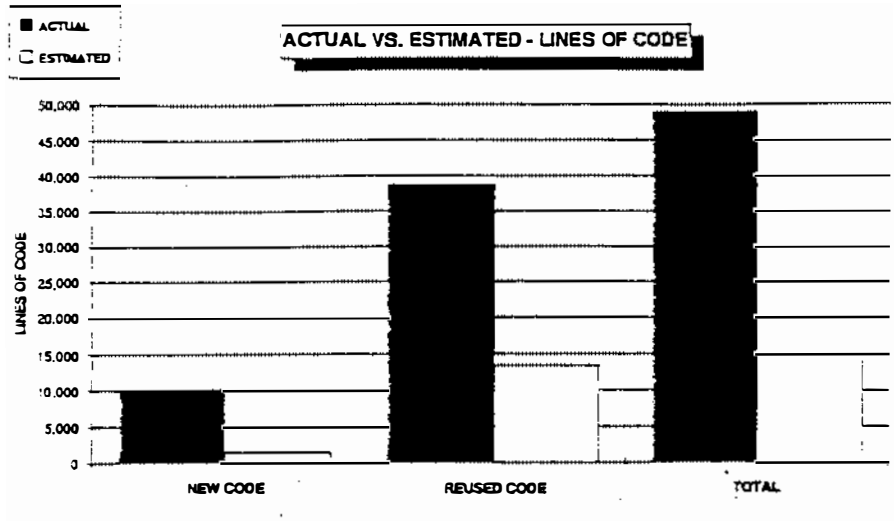


Figure 7. Size Estimates vs. Actuals for Small Project

importance of continual re-estimating, and project tracking and management throughout the life of the project. The five hands-on exercises in the course highlight the importance of using a team approach, using historical data, understanding the estimation method, the usefulness of estimation tools, and the importance of continually re-estimating.

As of March 2, 1992 the course has been given three times. The course is scheduled to be given four more times this calendar year and six times per year thereafter. To date 28 NRaD personnel and 14 SPAWAR personnel have attended the course. The focus of the course is a case study which is an actual, on-going NRaD project. Most of the exercises use data and products from the case study project. Roughly 12% of the course is dedicated to software estimation.

Conclusion - Many NRaD projects could now be considered to be a level 2 for software estimation. They could be considered level 2 because they can now explain how they did their estimates the last time and know how they are going to do their next estimate. Thus, they have a repeatable process for estimating, i.e., how to get a credible size estimate from which to estimate cost and schedule.

The greatest interest and awareness of the importance of good software processes is a direct result of the SPM course. The SPM course highlights the importance and helpfulness of management metrics which include cost estimation and tracking. The course exercises are all built around a real, on-going project which brings the point home. The course also provides hands on opportunity to use an estimation tool.

The working group has been the most effective method of disseminating information on the software estimating models and methods, Major progress has been made in developing credible estimates through the use of estimation tools. People are usually receptive to cost models the first time they use them. They like the ease-of-use afforded by the models and especially how easy it is to play "what if" games. People are usually impressed by the many factors included in the models.

To date, SEPO has provided estimation assistance to over 40 projects. This experience has helped highlight key elements that must be included in a formal estimation process. The initial estimates developed during the tutorials are often disclaimed as being "way too high." However, even if the estimates are much higher than anticipated (cost sometimes 2 to 3 times higher and schedule up to 50% higher) they often admit that they had not considered all of the factors contained in the model's environment.

After people receive an introduction to a model however, they sometimes skew the environmental parameters to get the desired answer. Common errors include overestimated staff capabilities and experience, too much faith in CASE, and underestimating the size. A common error also related to size is the underestimation of the effort to convert existing code. The conversion of existing code is almost never as trivial as initial assessments had assumed. Also, people do not allow for reduced documentation, configuration management and quality assurance requirements when estimating prototype projects.

Progress to date has been substantial but there is still a long way to go in making formal estimation processes an automatic part of every project. The instantiation of the process described here will hopefully contribute to increasing the credibility of proposed project costs and schedules.

3D Function Points:

Scientific and Real-Time Extensions to Function Points

Scott A. Whitmire
Boeing Computer Services
P.O. Box 24346 MS 6C-MA
Seattle, Washington 98124-0346
(206) 234-1643 (P)
(206) 965-0790 (F)
email: dsd048@kbct.profs.boeing.com

Abstract

Measurement is essential to improving the quality of our software products and processes. One basic measure is output size. To be useful, a measure of output must be based on characteristics of the problem being solved. It must also be independent of the influence of the technology used to solve the problem. This paper presents 3D Function Points, a problem-based, technology-independent measure, so called because of it roots in Function Points, first defined by Allan Albrecht in 1979. We discuss the three dimensional nature of an application problem. We identify measurable characteristics from each dimension that contribute to overall problem complexity. We provide rules for counting and assigning a level of complexity for the identified characteristics. Finally, we compare 3D Function Points to four existing problem-based size measures: Function Points, Feature Points, Mark II Function Points, and ASSET Function Points.

Biography

Scott A. Whitmire is a software engineer with over 10 years of applications development experience. He has been involved with software measurement for most of that time, particularly at the engineering level. He is an active participant in measurement activities at both the corporate and industry levels, including serving as a member of the International Function Point Users Group. Mr. Whitmire is also a member of the IEEE and the IEEE Computer Society, where he serves on the Software Engineering Standards Subcommittee.

Mr. Whitmire holds a Master of Software Engineering from Seattle University. His interests include: development methods, especially object-oriented methods; analysis, modeling, and design techniques; measures and metrics for model and design evaluation and selection; software development process engineering; and measurement of software contribution to the business at large. He is currently employed as a software engineer at Boeing Computer Services.

Introduction

We in the software business are learning some hard lessons about quality. We are under pressure to improve our quality and increase our productivity. We are beginning to understand that the quality of our products depends on the processes we use to develop them. From quality initiatives in other businesses, we know that continuous improvement of our processes is critical to our long-term survival; improvement in our products will come only after our processes are changed. Measurement is essential to understanding the current state of our processes, and where beneficial changes might be made. The basic unit for many of these measurements is product size.

Over the years, debate has been long and heated about how best to measure software product size. Lines of code is recognized as such a measure. It is also known to be heavily influenced by the technology used to develop the product. Several alternatives have been proposed. Most measure some aspect of the product. Only recently [15] have people begun to question the basic assumptions behind measuring the software product. An examination of why we need a size metric leads us away from measuring the product and towards measuring the problem itself.

Products vs. Problems

At a high level, software development is a matter of applying technology to a problem [15]. Metrics such as lines of code, Halstead's Software Science [7], and McCabe's Cyclomatic Complexity [11] measure characteristics of the resulting product. Product characteristics cannot be measured until after technology has been applied. Furthermore, the values of these measures are affected more by the implementation, the *way* the technology is applied, than by the problem itself. These *product-based, technology-dependent* metrics have proven to be ineffective for several aspects of software measurement.

Productivity assessment requires isolating the problem size from the effects of technology in order to make meaningful comparisons across environments. For example, did a project take 15 person years because of the scope of the application, or because they wrote it in assembler? In order to isolate the effects of technology, we need to directly measure the scope of the problem, or *problem size*; we need a *problem-based, technology-independent* measure.

Estimating can also benefit from a technology-independent size measure. The size of a problem can be determined very early in the development process. Once the size is known, the unit cost for a particular technology can be applied to derive an estimate for the project. In contrast, a technology-dependent size metric requires first estimating the size of the product. These estimates are difficult under any circumstances. Any variations in the unit cost are compounded by errors in the estimate of product size, which can lead to huge errors in the cost estimate for the project.

In 1979, Allan J. Albrecht introduced Function Points to measure productivity across several development environments. Function Points were the first attempt to measure problem size directly. Since then, Function Points have received wide industry acceptance. Albrecht refined them in 1984 [1], and today they are maintained by the International Function Point User Group (IFPUG) [8].

While Function Points have been extremely successful in the business domain, they have not enjoyed the same success in other domains. The primary complaint is based on a perception that Function Points do not adequately account for the complexity of scientific and real-time problems. In short, they do not measure all they need to measure. During the last decade, several extensions have been proposed, including Feature Points [10] and Mark II Function Points [14]. This paper presents another extension, 3D Function Points, based on the theory that a software application domain has three dimensions, each dimension contains characteristics that contribute to overall problem complexity or size, and these characteristics can be measured directly.

The Three Dimensions of a Problem

It helps to start our analysis by defining a framework to describe the various software domains. Tom DeMarco [5] proposed that all problems, and thus all applications, have three dimensions: the static properties of the retained data and user interface (data dimension), internal processing (function dimension), and dynamic behavior (control dimension). Each of these dimensions contain some of the characteristics that create complexity in a problem. This complexity is the *size* of the problem, which is the sum of the sizes of each dimension. The size of a problem is roughly equivalent to a volume of physical space. Our objective is to identify one or two characteristics from each dimension that can be measured directly and, when taken together, adequately measure the complexity or size of the problem.

We find that Function Points [1], [8] adequately measure the complexity contributed by the data dimension. Function Points measure the *internal data structure* and *external interface structure*. We will use the characteristics in Function Points as our data dimension characteristics.

Using the work of Alan Davis (see [4], p. 261), we expand our framework to describe internal processing as having two parts: the process steps or operations that transform the input directly into output, and the sets of semantic statements that govern these operations. The semantic statements determine when a particular operation is valid. Together, the operation's process steps and its semantics are called a *transformation*. Transformations will be the characteristic we measure from the function dimension.

Dynamic behavior is generally viewed as a set of *states* and *transitions* [4]. In fact, all of the behavior modeling techniques are expressed in terms of states and transitions [3], [4], [5]. Since states and transitions are so important to understand dynamic behavior, we will adopt them as our characteristics from the control dimension.

A Taxonomy of Problems

In some problems, one dimension can be said to dominate. Using this dominant dimension, we can place problems into one of four classes: *data-strong*, *function-strong*, *control-strong*, and *hybrid*. This section discusses the nature and identifying characteristics of problems from each class.

Data-strong (Information System/Business)

The complexity in a data-strong problem derives from the structure of the retained data and the effect it has on the user interface. Several modeling techniques focus primarily on the data structure and result in software solutions that reflect it. In applications that solve data-strong problems, most of the development time and effort are devoted to solving issues of data and interface complexity. Internal processing and behavior issues, while always present, represent a small fraction of the application's activities.

Data-strong applications are often called business or information applications because most business problems are dominated by their retained data and user interface. Such applications generally model or maintain information about real-world entities, both physical and otherwise. Their main purpose is to collect, store, and distribute this information.

However, not all information applications are in the business domain. Any problem that is primarily concerned with the collection, storage, retrieval, distribution, and structure of information describing real-world entities is a data-strong problem, regardless of the application domain.

Function-strong (Scientific/Engineering)

Complexity in function-strong problems derives from the processing required to transform the input data into the output data. The processing has two components, both of which are always present: process steps or algorithms that transform input directly into output, and sets of semantic statements that govern these

transformations [4]. The semantic statements define the relationships, constraints, and rules that constrain or define the use of the algorithm.

Function dominated applications model the processes or phenomena that operate on real-world entities or the behavior of entities in response to various stimuli rather than information about them. Such applications may not contain data *about* the entity so much as knowledge about *how the entity behaves*. In such an application, a car is not an object, but a set of aerodynamic patterns for air flow analysis.

Function-strong applications are often called scientific or engineering applications, depending on the people that use them. Scientific and engineering problems and their solutions tend to be heavily oriented towards internal processing. The data structures may be fairly simple, though the volume may be high, and user interface and control issues may be minimal or non-existent.

Control-strong (Real-time)

Applications that solve control-strong problems are unique in that they don't model anything - they perform real work. These problems are dominated by behavioral or control issues. Such applications are often called real-time because they must monitor a process and react to control or influence that process in a time scale defined by the process, not the computer [12].

A real-time application must control devices that operate within the real world, and do so in such a way that information is not missed or corrupted because it cannot keep up. Such an application must deal with timing, concurrence, and safety issues. They may not have a user interface *per se*, and little or no retained data. Furthermore, they may or may not do any processing of the data they collect other than to pass it on to another application. In fact, many real-time applications are composed of a single process which monitors and controls the devices connected to it, acting like a traffic cop.

Complexity in real-time problems is a function of the number of devices that must be controlled, the speed or time scale of those devices, the consequences of system failure, and the amount of concurrent access to common resources. Other factors, such as the size of the target computer, the operating environment, and the criticality of the problem may also influence the complexity. Because these factors may not be known when describing the problem, or may vary over time, real-time problems are often modeled in terms of the valid states the application's entities can assume, and transitions between those states.

Hybrid

In some problems, there is no clear dominant dimension. Hybrid problems fall into one or both of two categories: the problem is made up of components or subproblems that are dominated by different dimensions, or the entire problem space is truly hybrid. Complexity in a hybrid problem comes from more than one source. Applications that solve hybrid problems will not only model the entities they manage, but may also model their behavior and reactions, and may monitor and control those entities in real time. Decision support, avionics, and command and control applications are examples of hybrid applications [5].

Initial Assumptions and Goals

It is helpful to explicitly state our initial assumptions about, and goals for, a problem-based, technology-independent measure of output size. First, our assumptions:

1. All software is built to solve a problem;

2. We can measure the problem space directly, without regard for the technology used in the solution, and whether or not a solution exists;

3. Implementation of a software application is the process of applying a development technology to the problem space;

4. The *development technology* is a combination of attributes that describe the software solution and the development process. It is the quantification of the methods and techniques; languages; tools; management disciplines; team training and experience, including their level of maturity with the tools, methods, and process; and the goals and targets established for a given development project;

5. The time and effort required to implement a solution is a well behaved function of the problem size and development technology; and

6. The measure of size for a given problem will correlate strongly with the effort required to implement a solution, given a particular development technology.

If an application is a manifestation of a technology applied to a problem, the exclusion of technology from the size measure leaves the size of the problem itself. This value indicates the relative complexity of the problem compared to other problems. With this in mind, a problem-based size metric should meet these goals:

1. The metric measures the size of the problem space directly. That is, the value of the metric can be determined from the problem description;

2. The metric measures those characteristics of the problem that contribute to its inherent complexity;

3. The metric is based on a ratio scale;

4. The metric leads to reasonably accurate estimates for development effort and time, given the necessary historical data for the technology;

5. The metric can be understood and evaluated by non-software personnel, and especially by experts in the application domain;

6. The metric is generally applicable to problems that lie within a domain; and

7. The metric is fairly inexpensive to calculate and use, meaning the benefits derived from using it exceed the costs of calculating it.

3D Function Points

We define each of the characteristics from the three dimensions as element types. We are concerned with individual instances of these element types. 3D Function Points is the name we give the result of applying a process to these instances. The process involves counting the instances, assigning a level of complexity to each, and applying weighting factors in a formula to arrive at a single 3D Function Point Index. This index value is the size of the problem in 3D Function Points, as opposed to Feature Points or lines of code.

This section defines each element type, giving some guidance for indentfying an instance of the element type. It then gives the process of assigning the level of complexity. Finally, we apply the weighting factors and calculate the 3D Function Point Index.

Element Types, Counting Rules, and Levels of Complexity

This section introduces the 3D Function Point element types. The element types are grouped by the dimension in which they are found. For each dimension, the element types are defined, the counting rules are provided, and a process is given to assign the level of complexity.

Data Dimension

The data dimension has two characteristics: the *internal data structure* and the *external interface structure*. The internal data structure has one element type: *internal logical files*. The external interface structure has four element types: *inputs*, *outputs*, *inquiries*, and *external interface files*. The definitions, counting rules, and level of complexity assignment for these five element types are maintained by IFPUG. The material presented here is based on version 3.0 of the *Counting Practices Guide* [8].

The definitions of the five element types depend on three other terms, which are defined first:

Application Boundary - The application boundary is the line that separates the internals of a software application from its external environment. Identification of all of the element types depend on the placement of this line. Normally, the application boundary is the same as the human-machine boundary, but also includes the boundaries separating our application from others. The actual placement of the boundary is dependent upon the judgement of the user and application architect.

File - A file is a collection of data elements that is treated as a single unit in the application domain. A file in this context can have a complex structure, and includes repeating groups of data elements. In data modeling, the entities identified before normalizing correspond to files. Likewise, object classes correspond to files.

In general, there is a one-to-one correspondence between files and entities or classes, with one primary exception. Relationship structures that express aggregation are viewed as a single file. Aggregations are generally expressed as one-to-many relationships and include master to detail relationships, whole to part relationships, assembly to component relationships, and set to member relationships. For example, a customer order has three parts: a header, a group of line items, and some trailer material. In a data model, the header and trailer would be considered one entity, and the line items would be considered another. They would be associated with each other through a one-to-many relationship. Though they are correctly modeled as two entities, they are viewed by the application domain as one: the customer order. This application view is key to identifying the instances of a file.

Temporary or transaction files are not considered files. Temporary files exist due to imperfect implementation technology and are not part of the application domain. Transaction files are transactions that have a time lag between transaction start and transaction end.

Transaction - A transaction is a group of data and operations that are defined by the application domain. The operations must include the property of crossing the application boundary. A transaction can have data flowing one or two ways. A unique transaction is defined by a unique set of data contents, a unique source and/or destination, and a unique set of operations.

We now define each of the element types:

Internal Logical Files - A logical internal file is a file, as defined above, that resides entirely within the application boundary and is maintained through the application's transactions. In this context, maintained means data is added, accessed, or modified by a transaction. The level of complexity is assigned using figure 1. The RET acronym is for record types and represents each grouping of data within the logical file. Our earlier customer order example will have two record types. An aggregation structure will have one record type for each entity or class included in the structure. The DET acronym is for data element types. This is simply the number of fields in all of the file's record types.

	1 - 19 DET	20 - 50 DET	51+ DET
1 RET	L	L	A
2 - 5 RET	L	A	H
6+ RET	A	H	H

Figure 1. Level of Complexity table for Internal Logical Files and External Interface Files.

External Interface Files - An external interface file is a file that is maintained by an external application and is accessed directly by our application for reference. Our application may not change data in an external file. If this is the case, the file is counted as an internal file.

Under modern design and programming practice, and especially object-oriented practice, external files are not used. It is considered bad design to directly access data maintained by another application. This violates the principles of encapsulation and information hiding. It is becoming increasingly rare to encounter an external interface file. One will never encounter such a file in an object-oriented application.

The level of complexity is assigned to an external file using figure 1.

External Inputs - An external input is a transaction in which data passes across the boundary from outside to inside. Inputs are used to add data to, or modify data within, one or more internal files of an application. An input can originate from a human user or any external application. Files created by another application for the express purpose of transferring data to our application are considered external inputs, not interface files.

Primary examples of an external input include data entry screens, batch transaction files, and communication links with other applications. Each of these are merely conduits which carry the actual input transactions. An instance of an element type is identified at the conduit level; we are looking for transaction types, not individual transactions.

The level of complexity for an external input is assigned using figure 2. As with files, the DET acronym is for the data elements or fields contained within the input transaction. The FTR acronym is for file types referenced, and refers to the number of internal and external files updated or accessed during the processing of the transaction.

	1 - 4 DET	5 - 15 DET	16+ DET
0 - 1 FTR	L	L	A
2 FTR	L	A	H
3+ FTR	A	H	H

Figure 2. Level of Complexity table for External Inputs.

External Outputs - An external output is a transaction in which data crosses the application boundary from inside to outside. No internal files are modified, although one or more are accessed. The primary example of an output transaction is a report or display screen. The destination of an output transaction may be a human user or external application.

The level of complexity is assigned to an external output using figure 3.

	1 - 5 DET	6 - 19 DET	20+ DET
0 - 1 FTR	L	L	A
2 - 3 FTR	L	A	H
4+ FTR	A	H	H

Figure 3. Level of Complexity table for External Outputs.

External Inquiries - An external inquiry is a single transaction with both input and output components. An inquiry generally takes the form of a query in which criteria is passed into the application and the matching data is passed back out. A data entry screen may contain one or more inquiries, as well as inputs and outputs. The distinction is that an inquiry will always have data moving in both directions.

The level of complexity is assigned to an inquiry in a two step process. First, the input component is assessed using the process for external inputs and figure 2. Then, the output component is assessed using the process for external outputs and figure 3. The level of complexity for the inquiry is the greater of the input or output component.

Function Dimension

The primary characteristic of the function dimension is the number and complexity of the functions that represent the internal processing required to *transform* the input data into the output data. Processing has two parts [4]. The first part is the process steps required to turn input into output. The second part is the sets of semantic statements that govern or constrain the process. Semantic statements are predicates that must remain invariant throughout the sequence of operations, or the pre- and post-conditions defined for each operation.

Transformations - A transformation is the set of process steps and governing semantic statements required to transform one set of input data into output data. The process steps and semantic statements are derived from the problem space directly. Each unique set of steps and semantic statements is a unique transformation. A transformation must alter the fundamental nature or content of the data, or create new data. Processing that changes only the structure, format, or order of the data is not a transformation.

For example, in image processing, there are several operations that can be used depending on the needs and uses of the output. These operations are independent of each other; that is, an operation can be applied or not, without regard to any other operation. An image processing problem involves deciding which operations are needed and in which order. Each operation's output is used as input to the next. Each operation is counted as a unique transformation.

When solving mathematical equations, a transformation is the set of steps required to solve one set of equations. For matrix or linear programming problems, a transformation is the set of steps to solve one matrix equation or one set of linear equations.

Processing that reads an internal or external file and converts the data into an internal structure is not a transformation. Only the structure or format of the data has changed, not the content, and no new data has been created. Likewise, presenting data to and accepting data from the user, including edits, are not transformations. However, a calculation that uses a unit price and quantity to determine a discount amount is a transformation as new data is created.

Processing that converts numeric, tabular data into graphics data is a transformation because the fundamental nature of the data has changed (from left-brain data to right-brain data).

The overall complexity of a transformation is a function of many factors, including the number and types of process steps, the number and types of semantic statements, the volume of data to be converted, the nature of the mathematics, the required precision, and the required numerical accuracy. However, to simplify matters and make the determinations of complexity more objective, we will use only the number of process steps and the number of semantic statements. There are two reasons for this: these values can be taken directly from the problem space, and the other factors can be represented as semantic statements. The complexity of a transformation can be determined from the matrix in figure 4.

	1 - 5 Sem. Stmts.	6 - 10 Sem. Stmts.	10+ Sem. Stmts.
1 - 10 Steps	L	L	A
11 - 20 Steps	L	A	H
21+ Steps	A	H	H

Figure 4. Level of Complexity table for Transformations.

Control Dimension

Several characteristics have been advanced as possible contributors of complexity for the control dimension [10], [13], [14]. However, the primary emphasis in design methods and techniques for control-strong applications has been the organization and representation of system states and the transitions between them (chapter 4 of [4]), strongly suggesting that these are the prime contributors of complexity. Other factors, such

as the number of devices to be controlled, the number of critical sections of code[1], and the number of operating modes, can be influenced by the implementation and are not inherent characteristics of the problem domain. We want to include only those characteristics that can be measured directly from the problem space and cannot be affected by the approach we take to solve the problem.

States and Transitions - A *state* is a set that contains one and only one value for each condition of interest in the problem. More formally, given a state S_i, a change in the value of any condition within the problem will result in a new state, S_j. A state is normally associated with a single entity within the problem domain; each entity will have a set of states. A *transition* is a valid path (that is, defined within the problem space) from one state to another. There is at most one transition in each direction between two given states, S_i and S_j.[2]

The numbers of states and transitions are taken directly from a Finite State Machine (FSM) representation of the problem. If the FSM is described using *State Transition Diagrams* (STD) [4], then the states and transitions are simply counted. Initial and terminating states are not counted, as they are common to all entities. The transition count is reduced by the number of states, so that only multiple paths leading out of a state are counted, since each state must have at least one transition out of it.

There are other ways to model the behavior of a control-strong problem (again, see chapter 4 in [4]). These methods allow the analyst to specify different levels of detail. Some of the methods provide a higher level of abstraction than the standard STD, but all can be translated into a set of STDs.

For example, [4] discusses the use of *statecharts* and *decision tables* to represent complex problems in a concise manner. A *statechart* can be viewed as a hierarchy of state transition diagrams. States are counted using the diagrams on the lowest level of the statechart hierarchy. Transitions are counted in the same manner. Again, initial and terminal states are not counted, and the number of transitions is reduced by the count of states.

A *decision table* has two parts: a set of rules and a set of actions. The set of rules is a table that contains one row for each condition of interest within the problem. Each column is called a rule. There will be 2^n rules, where n is the number of conditions. The number of states is simply the number of rules. In some tables, some of the rules are impossible situations due to mutually exclusive conditions. In these cases, the number of states is the number of possible rules.

The set of actions is another table aligned by column below the first. This second table contains one row for each potential action. One or more of these actions may be marked as impossible, meaning the rule cannot be used. There may also be actions labeled "do nothing" which means the state does not change. Transitions are actions that are both possible and result in a state change. As always, the initial and terminal states are not counted, and the transition count is reduced by the number of counted states.

One consequence of our method of counting states is that the complexity of all states are equal because a given state in a STD can only represent one state in the problem domain. The same applies to transitions. Therefore, 3D Function Points define only one level of complexity each for states and transitions.

---

[1] A *critical section* is a process or segment of code that must be executed without interruption. The number of critical sections can be influenced by the design process. In other words, one can design a solution to minimize the impact (the number and/or length) of critical sections. Therefore, we will not include this characteristic in our metric.

[2] Some modeling techniques provide for a transition for each stimulus/response or event/action pair that can cause a state change from S_i to S_j. For our purposes, we are concerned only with the conditions that change as we move from state S_i to state S_j. Therefore, mutliple transitions in the same direction between two states S_i and S_j are collapsed into a single transition for counting purposes.

3D Function Point Index Calculation

The general form of the 3D Function Point formula is:

$$\text{3D Function Point Index} = I + O + Q + F + E + T + S + R \quad (1)$$

where I, O, Q, F, E, T, S, and R are the terms for inputs, outputs, inquiries, internal files, external files, transformations, states, and transitions, respectively. Each term in the formula is:

$$\text{Element Type} = W_{el}N_l + W_{ea}N_a + W_{eh}N_h \quad (2)$$

$$(\text{with } I = W_{Il}I_l + W_{Ia}I_a + W_{Ih}I_h, \text{ for example}) \quad (2a)$$

where N_l, N_a, and N_h represent the number of occurrences of an element type at each level of complexity (low, average, or high) and W_{el}, W_{ea}, and W_{eh} are the weights for each rating as determined by using the complexity assessment rules for element type *e*.

The counts for each level of complexity for each element type are entered into a table such as the one in figure 5. Each count is multiplied by the weighting factor shown to determine the weighted value. The weighted values on each row are summed across the table, giving a total value for each element type. These totals are then summed down to arrive at the total 3D Function Point Index.

Element Types	Level of Complexity Low	Average	High	Total
Inputs	___ x 3 = ___	___ x 4 = ___	___ x 6 = ___	___
Outputs	___ x 4 = ___	___ x 5 = ___	___ x 7 = ___	___
Inquiries	___ x 3 = ___	___ x 4 = ___	___ x 6 = ___	___
Internal Files	___ x 7 = ___	___ x 10 = ___	___ x 15 = ___	___
External Files	___ x 5 = ___	___ x 7 = ___	___ x 10 = ___	___
Transformations	___ x 7 = ___	___ x 10 = ___	___ x 15 = ___	___
States	N/A	___ x 1 = ___	N/A	___
Transitions	N/A	___ x 1 = ___	N/A	___
Total 3D Function Point Index				___

Figure 5. Table of Weights for 3D Function Point Index Calculation.

Comparisons to Previous Work

3D Function Points are the results of a project to identify a problem-based measure of output size. Part of that project was a critical analysis of existing problem-based measures against our framework and goals. Four measures were analyzed: Function Points, Feature Points, Mark II Function Points, and ASSET Function Points. This section presents our analysis of the four measures. The weaknesses identified influenced the formation of 3D Function Points.

IFPUG/Albrecht Function Points

Function Points were developed by Allan Albrecht during the mid 1970's as part of an effort to measure productivity across an organization using many different technologies. Albrecht sought a measure of output size that was independent of the technology used to develop the product. He was one of the first to measure the functionality delivered to the user, rather than the volume of product required to deliver it.

The original Function Points, published in 1979, counted external inputs, outputs, and inquiries - data that passes across the human-machine boundary; master files - data retained by the application; and data that is passed between applications. The number of instances of each element type was multiplied by a weighting factor and then summed to produce an Unadjusted Function Point total. The weights were derived from empirical observation. The Unadjusted Function Point total was then adjusted for overall system complexity by ±25%. The magnitude of the adjustment was arbitrary, leading to many of the early complaints about Function Points [14].

Early users of Function Points discovered that the definitions of the element types (inputs, outputs, inquiries, and files) left substantial leeway when it came to actual counting. Albrecht published a major revision to Function Points in 1984 [1]. The revision provided more guidance for identifying and counting the various element types, and for classifying the complexity of an element type instance as low, average, or high. This allowed the Unadjusted Function Point count to vary over a much greater range.

Albrecht also provided more guidance for calculating the adjustment factor. Rather than selecting the number from thin air, the user ranks a set of 14 system-wide factors, called General Systems Characteristics, from 0, having little or no influence on the application, to 5, having significant influence throughout. The rankings are summed, multiplied by .01 to convert to a percentage, then added to .65. The result is an adjustment factor ranging from .65 to 1.35 (±35%), enhancing both its range and objectivity.

Function Points were developed in a business system environment. They have been shown to work well in that domain, but have not worked well in other software domains such as scientific and real-time. When Function Points are applied in these domains, the results are too coarse - the effort for problems of the same size varies over too large a range.

Function Points enjoy wide industry support. However, many people have criticized Function Points, including Jones [10], Symons [14], and Verner, *et al* [15], mostly regarding the inadequacy described above. Our analysis indicates this weakness is due entirely to the fact that Function Points measure only one dimension. The General Systems Characteristics are intended to account for the other dimensions but are not adequate.

Feature Points

Feature Points were developed by T. Capers Jones in 1986 [10] as a better measure of size than Function Points for non-business problems. Such problems, according to Jones, are characterized by complex internal processing.

Feature Points grew out of a 1985 modification made to Function Points. Jones's modification eliminated the classification of the complexity of each element type instance, using the average weights from Albrecht, and significantly changed the calculation of the adjustment factor. The adjustment factor is calculated by rating, on a scale of 1 to 5, the overall complexity of the logic and data structures, summing them, and then looking up the result in a table. The adjustment factor ranges from .6 to 1.4 (±40%). This method of calculating function points was incorporated into Jones's SPQR/20 product.

As an experiment for counting telephone switching software, Jones added an algorithm parameter to his function point definition, changed the weight on the internal files from 10 to 4[3], and called the result Feature Points. He defines an algorithm as "the set of rules which must be completely expressed in order to solve a significant computational problem [10]." As examples, he gives a Julian date conversion routine and a square root function.

Feature Points showed the most promise of the four metric proposals. They are the first attempt to address the function and control dimensions. However, like the other metrics, they do not measure all that needs to be

[3]The current Feature Point weights are 7 for internal files and 3 for algorithms [2].

measured. Specifically, the algorithm parameter, as defined, does not account for both the procedural and semantic aspects of the function dimension, nor do recent modifications to the parameter[4]. In addition, Feature Points do not address the control dimension at all.

Furthermore, the definition of algorithms and the examples given by Jones are not sufficient for counting purposes. No guidance is provided to identify algorithms at the desired level of abstraction. A Julian date conversion, for example, is almost always part of a date calculation. The algorithm of interest is the date calculation itself, not its component parts, and then only if it is not part of a larger transformation.

Some algorithms may be used more than once in a high level transformation. For example, two Julian date conversions are required for each date calculation. Are both conversions counted as separate algorithms? Are they counted each time a date calculation is made? Feature Points give no insight into how to count reused algorithms.

Feature Points will undersize data-strong problems due to the reduction of the weighting factor for internal files. In such problems, the structure of the internal data is the major driver for the overall problem complexity.

The complexity adjustment for Feature Points is based on two aspects of the overall problem: the overall complexity of the data and logic structures. For problems where the complexity is consistent across the problem domain, this is fine. In most problems, however, complexity is not evenly distributed. Some portions of the problem may be very simple, while others are very complex. In such cases, the entire problem is rated as average, making the adjustment insensitive to the distribution of complexity over the problem domain.

In addition, the complexity adjustment amounts to double consideration for algorithms and internal files. It appears that this was unintentional since the complexity adjustment is unchanged from Jones's modified Function Point calculation.

In summary, Feature Points are a step toward measuring all three dimensions, but they are not enough. They do not consider the control dimension, and fail to consider the semantic aspects of the function dimension. Also, some aspects of the problem domain are considered twice.

Mark II Function Points

Mark II Function Points were developed by Charles Symons [14] in 1988 as an improvement to Function Points. Symons reasoned that the effort required to develop an information system is the product of three factors:

1. The *information processing size*, a direct measure of the information processed and provided by the application;

2. A *technical complexity factor* that quantifies the influence of the various technical factors involved in implementing the product; and

3. *Environmental factors*, a combination of factors arising from the project environment, including tools, management practices, and team skill. This factor is equivalent to our *development technology*.

The information processing size and technical complexity factor comprise the *intrinsic size* of the problem being solved, and are needed for productivity studies. The environmental factors must be added for estimating purposes.

---

[4]Recently, Jones modified the algorithm parameter to include a complexity matrix based on rules and factors per rule [2]. Rules, in this use, are equivalent to process steps; there is no coverage for the semantics part of our framework.

Like Albrecht, Symons focuses on measuring the problem size. He characterizes all processing in an information system as a set of transactions, each having an input, processing, and output component. The inputs and outputs can be counted directly, and sized according to the number of data elements involved. Symons does not apply this reasoning to scientific and real-time domains. Our experience indicates that this line of reasoning is invalid in those other domains where identifiable transactions may not exist.

To measure the processing, Symons relies on the work of Michael Jackson [9], which states that a well structured function should match the logical structure of the data it uses. Symons postulates that the structure of the data is represented by the path a transaction takes through the entity model during processing. He claims that each step in the path through the model generally involves a branch, and that steps involving many-to-many relationships require a loop. Based on this line of reasoning, Symons defines processing size as the number of entities on this path, summed across all transaction types. As with transactions, our experience indicates that this line of reasoning does not apply to function-strong and control-strong problems, where all of the interaction may be with a single entity.

The unadjusted Mark II Function Points are calculated as:

$$UFP = W_i N_i + W_e N_e + W_o N_o \tag{3}$$

where N_i, N_e, and N_o represent the counts for input, entity, and output types, respectively, and W_i, W_e, and W_o represent the respective weights.

Symons devotes significant effort to the calculation of appropriate weights. Using a small sample of empirical data, he calculates $W_i = 1.56$, $W_e = 5.9$, and $W_o = 1.36$. These weights are the average hours devoted to each component based on the sample data. For some unstated reason, Symons wanted the Mark II and Albrecht Function Point counts to be the same for systems up to 500 Function Points. To accomplish this, he changes the weights to $W_i = 0.44$, $W_e = 1.67$, and $W_o = 0.38$. Note that the unadjusted Mark II Function Points only measure the information processing size; the technical complexity factor has not yet been applied. Note also the assumption that the weights are in units of hours implies a specific development technology.

Symons feels that the factors used to calculate the technical complexity factor in Function Points have several deficiencies. The inclusion of only 14 factors may be inappropriate in the typical development environment used today. He proposes 20, including many of the original 14, plus:

1. The interfacing required with other applications;

2. Special security requirements;

3. Access provisions for third parties;

4. Documentation requirements; and

5. The requirement for special user training facilities, e.g. the need for a training subsystem.

Symons performed two analyses to assess the validity of Albrecht's technical complexity adjustment. The first compared the technical complexity factor as calculated according to Function Point rules to the actual factor determined from estimates of relative work effort data. The second analysis attempted to correlate the degree of influence for each factor in the Function Point adjustment to the "estimated actual percentage development effort." Symons felt that more data was needed to get clear results from this second analysis. His preliminary findings are:

1. The distinctions between some of Albrecht's factors, particularly those relating to performance and on-line dialogs, are not clear, and some combining of these factors is warranted;

2. The factors relating to the ease of installation, operation, change, and security requirements do not require the same effort per degree of influence as other factors;

3. Symons's factor regarding documentation may require double the effort per degree of influence as other factors; and

4. The factor measuring the complexity of the internal processing is no longer appropriate, given the direct measurement of processing for each transaction type.

It turns out that Symons never develops a new technical complexity factor in [14]. The examples he gives using Mark II Function Points do not include any form of adjustment. In the end, Mark II Function Points measure only one of the three factors that contribute to the size of a problem: information processing. Technical complexity is not addressed.

In Function Points, the weights measure a component's relative contribution to problem size. The Mark II weights implicitly measure a component's relative contribution to work effort required to solve the problem. Thus, the Mark II weights imply a development technology. Symons confirms this by asserting the weights must change when the technology changes.

Symons's method of using estimates and gut feel to distinguish work effort due to information processing size versus technical complexity raises serious questions about the validity of his conclusions. We know from experience that people are not good at remembering the effort due directly to the problem versus the effort due to indirect concerns.

Mark II Function Points do not measure characteristics from the function and control dimensions. This, plus the implicit dependence on a technology, and his questionable analysis techniques lead us to reject them as an extension to Function Points.

Asset Function Points

Donald J. Reifer [13] developed what we will call ASSET Function Points as part of his Analytical Software Size Estimation Technique (ASSET). He did not set out to redefine Function Points, but was looking for a better way to estimate source lines of code (SLOC) given certain information about a product. Reifer primarily concentrated on scientific and real-time software, which is why we include his proposal.

Reifer's formula for SLOC is given in [13] as:

$$SLOC = (ARCH)(EXPF)((LANG * FP_A) + MV_N)^a \quad (4)$$

where ARCH is an architectural constant, EXPF is an expansion factor based on a set of size modifiers, LANG is the expansion factor for the target language, FP_A is the adjusted Function Point total (using Reifer's definition of Function Points), MV_N is a normalized measure of the math volume, and a is the reuse factor.

For function-strong problems, Reifer's Function Points count inputs, outputs, master files, modes, inquiries, and interfaces. The term *modes* is not defined, nor is its meaning clear from [13]. According to Reifer, his empirical data showed that better results were obtained without weights, so the function point count is left unadjusted.

For control-strong problems, the function point formula adds the number of stimulus/response relationships and the number of rendezvous. Again, *modes* is undefined, as are *stimulus/response relationships* and *rendezvous*, and no weights are used.

The architectural constant (ARCH) is determined from the overall architecture of the system. The values are derived empirically, and range from 0.9 to 2.1.

The expansion factor (EXPF) is calculated by multiplying a calibration constant by the sum of a set of multipliers. The multipliers rank such factors as: requirements volatility, database size, degree of real-time processing, use of modern programming techniques, use of software tools, analyst capabilities, and the experience of the team members with the application, environment, and language. Reifer does not give any guidance on how to determine a particular value for any of the multipliers (presumably, you need to buy his product).

While the calculation of ARCH and EXPF have nothing to do with calculating the size of a problem, they are interesting in that they are among the first attempts to quantify a development technology.

For our analysis, we focused on the function point definition. Given that *modes* is undefined, it is impossible to determine what Reifer intended to measure. It seems clear from his context, however, that it is some characteristic of the solution rather than the problem. His inclusion of stimulus/response relationships for control-strong problems is valid, as these are taken directly from the problem space, and can be mapped directly onto transitions. Rendezvous, like critical sections, can be designed out of a solution and are thus characteristics of the solution, not the problem.

The proposal of different formulas for function-strong and control-strong problems makes measuring hybrid systems difficult. In addition, Reifer offers no help in classifying a problem as function-strong versus control-strong. This limits the usefulness of the function point formulas. As a whole, the ASSET technique is significantly influenced by the development technology. While this is intended for Reifer's purposes, it violates several of our goals.

Summary and Conclusion

All software is developed to solve a problem. We can measure certain characteristics of the problem space directly, ignoring any solution for that problem. A software solution is the result of applying a development technology to a problem. Since we can choose from among many potential technologies, every problem has many solutions. We need a size measure that is independent of technology to make comparisons between and decisions about development technologies. We also need it to assess the efficiency and effectiveness of our ability to solve a problem.

Problems have three dimensions: data, function, and control. Problems can be classified according to the relative influence of these dimensions as being data-strong, function-strong, control-strong, or hybrid. Data-strong problems are dominated by characteristics from the data dimension. This dimension describes the *retained data and user interaction requirements*. The data dimension is measured by counting *internal and external files, external inputs, external outputs,* and *external inquiries*. Function Points, as published by IFPUG [8], measure this dimension very well.

Function-strong problems are dominated by characteristics from the function dimension. This dimension describes the *internal processing required to transform input data into output data*. This internal processing has two parts: the actual process steps and the semantic statements or rules under which those steps must operate. The combination of the two parts is called a *transformation*. The function dimension is measured by counting transformations.

Control-strong problems are dominated by characteristics from the control dimension. This dimension describes the *valid combinations of conditions of interest* that may exist within the problem domain. Each combination represents one of the possible *states* that the problem may be in at any point in time. The control dimension also describes the allowable operations that may be performed while in one state to *transition* to another. The control dimension is measured by counting states and transitions.

Hybrid problems are not dominated by any one dimension; the characteristics that create the complexity come from multiple dimensions. Modern applications often exhibit hybrid characteristics, either at the problem level, or through different subproblems that are dominated by different dimensions.

All software has some aspect of each dimension. The difference between classes of problems is not whether a given dimension is present, but which dimension dominates.

3D Function Points measures characteristics from all three dimensions. Other existing problem-based measures, including Function Points, Feature Points, Mark II Function Points, and ASSET Function Points measure characteristics from only one or two dimensions.

Acknowledgements

The research for this paper was conducted as part of a company-wide software metrics effort, the Software Metrics Application Expansion (SMAX) Project. The author wishes to thank the members of the SMAX team for providing assistance with reviews and research. The author wishes to thank the members of the research project team, Michael Crow, Steve Galea, Russ Gold, and Jerry Tollar, for their work in gathering and analyzing the information used to write this paper. The author particularly wishes to thank Judith Carson for volunteering to proofread and edit this paper.

We also wish to acknowledge the support and encouragement from IFPUG, especially Eldonna Williamson and Allan Albrecht, and from others within the software business. No product can be built without the help of many people. Ours is no exception. We thank the many colleagues that helped us through discussion, debate, and research contributions.

References

1. A. J. Albrecht, *IBM CIS&A Guideline 313, AD/M Productivity Measurement and Estimate Validation*, IBM, 1984.

2. A. J. Albrecht, Personal Communication, May, 1991.

3. P. Coad and E. Yourdon, *Object-Oriented Design, Yourdon Press*, Englewood Cliffs, New Jersey, 1991.

4. A. M. Davis, *Software Requirements Analysis and Specification*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

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

6. D. Embley, B. Kurtz, and S. Woodfield, *Object-Oriented Systems Analysis - A Model-Based Approach*, Yourdon Press, Englewood Cliffs, New Jersey, 1992.

7. M. H. Halstead, *Elements of Software Science*, Elsevier North-Holland, New York, 1977.

8. *Function Point Counting Practices, Release 3.0*, International Function Point Users Group, 1990.

9. M. Jackson, *Principles of Program Design*, Academic Press, London, 1975.

10. T. C. Jones, *A Short History of Function Points and Feature Points*, Software Productivity Research, Inc., Cambridge, Mass., 1988.

11. T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol SE-2, no. 4, pp. 308-320, 1976.

12. D. A. Mellichamp, Ed., *Real-Time Computing With Applications to Data Acquisition and Control*, Van Nostrand Reinhold, New York, 1983.

13. D. J. Reifer, "Asset-R: A Function Point Sizing Tool for Scientific and Real-time Systems", *Journal of Systems and Software*, draft prepared Aug. 1988 (date of publication unknown).

14. C. R. Symons, "Function Point Analysis: Difficulties and Improvements," *IEEE Transactions on Software Engineering*, vol. SE-14, no. 1, pp. 2-11, Jan. 1988.

15. J. M. Verner, G. Tate, B. Jackson, and R. G. Hayward, "Technology Dependence in Function Point Analysis: A Case Study and Critical Review", *Proceedings of the 11th International Conference on Software Engineering*, ACM/IEEE, May 1989.

16. E. Yourdon, *Modern Structured Analysis*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

An Indicator of Information Hiding

Linda S. Rising and Frank W. Calliss

Department of Computer Science and Engineering
College of Engineering and Applied Sciences
Arizona State University
Tempe, AZ 85287-5406, U.S.A.

Abstract

Developers and maintainers of large software systems must face the dilemma of overwhelming complexity. The most powerful mechanism for controlling complexity is abstraction — limiting consideration to those details that are most relevant. In a programming language, constructs that support high level abstraction allow encapsulation and enforce information hiding. These constructs are called modules, packages, or classes. This paper describes the creation of a metric for information hiding at the module level based on measurement theory. The relationship between the information hiding metric and the maintainability of programs written in modular programming languages is then validated using subjective expert opinion and a case study involving several versions of three subsystems of a large Ada program.

Biography

Linda Rising has a B.A. in chemistry, a M.A. in mathematics, a M.S. in computer science, and a Ph.D. in computer science. Her interests include all software development and maintenance issues, especially quality improvement and metrics. She is a member of ACM and the IEEE Computer Society.

Frank W. Calliss is an assistant professor at Arizona State University. Dr. Calliss received his Ph.D. in Computer Science from the University of Durham in England in 1990. He research interests are in software metrics and software maintenance. He is a member of the ACM, IEEE and the British Computer Society.

1 Introduction

The development and maintenance of large software systems are overwhelmingly complex tasks. An important weapon against this complexity is abstraction, that is, ignoring unimportant details and concentrating on important ones. An important form of abstraction is *information hiding*. The concept of information hiding was introduced by Parnas [10]. According to this precept, each module hides a single design decision from other modules.

Module interfaces or connections should reveal as little information as possible. Since interfaces represent the assumptions that modules make about one another, changes to a module should not involve changes to the interface. Parnas [11] states that the interface is created by identifying a set of operations that cannot be efficiently performed without knowing the hidden design decision.

The module construct considered in this research appears in object-based languages like Ada and Modula-2. The module construct in the object-oriented programming languages presents issues that this research has not considered, i.e., inheritance. The issue of inheritance complicates information hiding as we need to determine when an entty should be inherited, imported of declared locally in a package. To avoid this complication, only object-based languages were considered in this research.

2 Information Hiding Metrics

The metrics produced in this research were derived using measurement theory [7, 8, 16, 20]. According to the principles of measurement theory, a measure is the assignment of a number to an entity to characterize an attribute of that entity. Therefore, a measure is not a number but a mapping between the entity and attribute.

There must be a clearly identified set of entities and a clearly defined attribute before measurement is done. The set of empirical relations $\mathcal{R}$ that characterize the attribute, together with the set of entities $\mathcal{E}$, is called an empirical relation system. Creation of the empirical relation system involves identification of properties or axioms that capture the intuitive understanding and empirical observations about the attribute.

A number system is then determined into which to map the entities. The numerical relation system consists of a set $\mathcal{N}$ together with a set of relations $\mathcal{P}$ over $\mathcal{N}$. For each relation in the empirical relation system for a given attribute, a corresponding relation is needed in the numerical relation system.

The representation theorem requires showing that the attribute can be represented in an appropriate number system by a mapping from the set of entities possessing the attribute to the number system. Relations induced by the attribute on the set of entities should imply and be implied by the relations between their images in the number system, that is, all empirical relations must be preserved in the numerical relation system. This latter requirement is known as the representation condition.

The uniqueness theorem requires showing that any two functions that are defined from the set

of entities to the set of numbers and that represent the attribute are related in certain ways. This limits the kind of mathematical operations that can be performed on the measurement values.

The concept of information hiding was introduced by Parnas [10]. According to this precept, each module hides a single design decision from other modules. This definition exceeds that of merely concealing data. Parnas [12] equates information hiding with encapsulation or abstraction where the unimportant details that are hidden from the client are all related to a single design decision.

Module interfaces or connections should reveal as little information as possible. Since interfaces represent the assumptions that modules make about one another, changes to a module should not involve changes to the interface. Parnas [11] states that the interface is created by identifying a set of operations that cannot be efficiently performed without knowing the hidden design decision. This definition is the basis for the development of a metric for information hiding at the module level.

Based on the Parnas definition, there are two considerations in the determination of information hiding at the module level: (1) whether or not the module implements a single design decision and (2) whether or not the interface is minimal. Intuitive notions regarding these considerations and the application of measurement theory [15] result in the following metric for information hiding at the module level

IH(Module) = Number of Design Decisions + Number of Extraneous Entities

From the viewpoint of measurement theory, there are two levels of validation that may be performed for software measures [7]. The first ensures that the measure is a proper numerical characterization of the specified property. For example, if the measure is intended to define modularity of designs, then the measure should preserve any intuitively understood relations on designs imposed by the notation of modularity.

The second type of validation to be considered in measurement theory, often considered to be the only type of validation for software measures, is to demonstrate that a measure of some internal attribute, like modularity, is related to some externally observed attribute of the process or product, like reliability.

3 Subjective Validation

Subjective validation using expert opinion was performed to satisfy the first kind of validation. Subjective validation is part of the methodology proposed by Baker et al. [2] for defining a measure. Relations induced by the quality attribute on the set of entities should imply and be implied by relations between their images under the measure in the number system. This is the representation condition described in the previous section. Changes in the measure are compared with similar changes in the attribute. This determination can be made by a single expert or a consensus. Applying subjective validation supplements the theoretical verification of applying the definition and axioms.

The programs used for the subjective evaluations are: (1) Flight, an Ada program used to determine flight routes; (2) m2dep, a program in Modula-2, which reports the dependencies between modules; (3) DASC, an Ada style checker. The value for the module level information hiding metric was computed for each module in the three programs and the modules ranked according to their value for the information hiding metric. This ranking was compared with that prepared by experts on the three programs. The Spearman correlation coefficient was used, since this statistic measures the extent to which the ranks of the two variables are in agreement. This is a non-parametric test, appropriate for ordinal or nominal data, since the information hiding metric is at best ordinal data.

3.1 Validation with Respect to Three Programs

3.1.1 Flight

Computation of the Spearman rank correlation coefficient produced a value of 0.750, which indicates a moderate linear relationship between the subjective ordering and that of the original version of the information hiding metric. During the case study, a revised form of the metric was developed. (The revised metric is given in Section 4.2.) The rank correlation computation for the revised metric was 0.875, which indicates a strong linear relationship between the subjective ordering and that of the revised version of the information hiding metric.

3.1.2 m2dep

Computation of the Spearman rank correlation coefficient produced a value of 0.717, which indicates a moderate linear relation between the subjective ordering and that of the original version of the information hiding metric. The value of the information hiding metric was also computed using the revised metric and produced a value of 0.883, which indicates a strong linear relation between the subjective ordering and that of the revised version of the information hiding metric.

3.1.3 DASC

Computation of the Spearman rank correlation coefficient produced a value of 0.857, which indicates a strong linear relationship between the subjective ordering and that of the original version of the information hiding metric. The value of the revised metric was also computed and produced a value of 0.804, which indicates a strong linear relation between the subjective ordering and that revised version of the information hiding metric.

The results of subjective validation with the original form of the metric were sufficient to carry out the case study. They also suggest that the information hiding metric is appropriate for more than one object based language, since both Ada and Modula-2 programs were examined. The metric does not consider the effects of inheritance, so no true object-oriented languages were considered.

4 Case Study Validation

This case study involved three subsystems of Project $\mathcal{X}$, a large Ada program that contains approximately one-million lines of code and has been through several releases. To validate the information hiding metric, an external attribute was needed to correlate with the values of the metric. It seems intuitively obvious that modifying a package with poor information hiding should cause ripple effect, while a package with good information hiding would not. Ripple effect is observed when changes in one section of a program necessitate changes in other sections of the program [19].

Two releases were examined; each release comprised one or more versions of each package in each subsystem. 495 packages were examined in subsystem $\mathcal{A}$, 357 in subsystem $\mathcal{B}$, and 432 in subsystem $\mathcal{C}$. These packages had all undergone some change to produce a new version of the package. The value of the information hiding metric was determined for each package. The history of each package was then examined to see what kinds of changes were made and whether a measure of these changes correlated with the information hiding metric. The history of each package consisted of the code changes that were made between the different version of each subsystem.

4.1 Defining the Number of Design Decisions

In the subjective validation the computation of the information hiding metric had been carried out by examining the programs and computing the sum of the number of design decisions and the number of extraneous entities in the visible part.

Determining the number of design decisions was easy when the programs were small and each module could be studied. Trying to determine the number of design decisions in a large number of packages was more difficult. The determination had to be done semi-automatically to avoid spending an inordinate amount of time on this part of the analysis. The examination required several passes. The first pass was made with AdaMAT, a sophisticated metrics tool for Ada programs. Second and third passes, if needed, were made with two tools developed for the case study.

A list of definitions for single decision decisions was created after examining several packages by hand and considering the information that could be gained from the tools. These definitions were based on descriptions of packages with model cohesion [13]. Examples of these packages include: the implementation of one abstract data type or abstract state machine; a set of related constants; a set of related operations; the interface to an external device; a set of system parameters (related constants, types, and operations, but not variables).

AdaMAT reported the number of entities but did not relate their types nor the relationships among types and subprograms. Some packages contained an enormous number of types (over 300) and it was not easy to decide whether these were related in any way at all.

Some package specifications contained constants that were undoubtedly related to the design decision in the package but these constants were not usually all of the same type nor of related types. Since it was impossible to solve these problems automatically, constants were counted as

extraneous entities but were not used to determine the number of design decisions. Subprograms are not counted as extraneous entities but are considered in determining the number of design decisions.

It was shown in the statistical analysis [14] that the number of constants is not a significant factor in considering the likelihood of significant change for Project $\mathcal{X}$ (this result may not hold for other large projects). The packages that contained very large numbers of constants, in fact, were shown not to undergo significant change. One very good reason for this is that these packages were probably imported by a large number of clients and any significant change would have meant ripple effects through a large number of users. The developers obviously avoided this problem by never changing these packages.

Kafura and Reddy observed in their study of a medium size software system [9] that maintainers will avoid very complex procedures, even when the maintainer knows that restructuring of the complex procedure is called for to implement required enhancements. The maintainer will make inappropriate changes to other parts of the system to avoid the complex procedure. Continuation of this kind of behavior eventually produces unmaintainable systems.

One kind of design decision not included in the list is the abstract state machine. Since the bodies were not analyzed, it could not be determined if it contained a global variable accessed by the subprograms. The item in the list that lies closest to this design decision is the collection of subprograms that operate on a common data type or are parameterless.

The following are definitions of single design decisions whose determination could be made automatically: (1) one private type; no variables; any number of constants; all subprograms operate on the private type, (ADT); (2) one visible structured type; one or two additional simple types in the same type family as the structured type [4]; any number of constants; all subprograms operate on the type family, (ADT); (3) no variables; no types; any number of constants; all subprograms operate on the same type or have no parameters, set of related operations; (4) a collection of constants only, set of related constants; (5) one variable; any number of constants; any number of subprograms, not model cohesion, poor design decision since variable will be global to all client packages.

It is difficult to determine the number of design decisions automatically so it was decided to use a binary evaluation, one decision or more than one decision: now give the form of the expressions with (0 | 1).

4.2 The Revised Metric

Initially, a count was kept of all constants, simple and structured variables, simple and structured types. After the Project $\mathcal{X}$ data was collected, a simple trial-and-error approach was used to try other forms of the metric, using various combinations of the entities with various coefficients. The result was the following revised form of the information hiding metric.

IH(Module) = 2 * (0 | 1) + Visible Variables (Simple and Structured)

The revised metric was also used in the statistical validation tests described in the following section, and the results were better than those with the original form of the metric.

The revised form of the metric seems to indicate that the number of design decisions is of greater importance in determining the information hiding of a package and is therefore the important factor in determining whether or not a package will undergo significant change. Next in importance is the number of visible variables. This is an old issue in programming [18]. There is undoubtedly universal agreement among users of all programming languages that the use of global variables is bad programming practice. Some programming languages allow control over variable access, by allowing the programmer to specify public entities. In languages like Ada, Modula-2, and Oberon the module comprises a separate specification and implementation parts, where all the public entities are found in the specification. In C++, CLU, and Eiffel, modules contain only one part but public entities may be listed and any entity not listed is assumed to be private [4].

The following validation was performed on the revised metric. The original form did not produce significant results in the Mann-Whitney test or the χ^2 test. Although the results are not quite as good as the revised form, simply using the number of design decisions gives fairly good results. Early intuition seemed to indicate that the number of extraneous entities was a more important component of the metric than the number of design decisions. It is intuitively appealing to conclude that the more objects in the interface, the more problems will arise during a modification. Further investigation into the relative effects of these two components of information hiding should be carried out.

4.3 Statistical Validation

The purpose of metrics validation is to show that metrics can predict quality factor values. If metrics are to be useful, they must accurately indicate whether quality requirements have been met or are likely to be met in the future. It is important that metrics be validated, otherwise, metrics may be misapplied or metrics may be used that have little or no relationship to the desired quality characteristics.

According to Schneidewind [17], validation should use nonparametric statistical techniques because of the nature of metrics data. The values are typically not normally distributed, usually skewed toward zero, have large variability, and include several very large values [6]. There are several advantages of nonparametric statistics over parametric methods, which are important for metrics validation. The ranks of random variables can be used instead of the values themselves, thus relaxing the assumptions about data relationships, while providing a measure of quality. The fact that the data is not well-behaved does not necessarily mean that it is less valuable. In fact, many useful applications of metrics can be derived from the ability to classify products as better or worse, high quality or low quality, acceptable or unacceptable. The information provided by nonparametric analysis supports this approach.

To validate the information hiding metric (IH), an external attribute was needed to correlate with the values of the metric. It seems intuitively obvious that modifying a package with poor information hiding should cause ripple effect through all user packages while a package with good

information hiding would not. Ripple effect is observed when changes in one section of a program necessitate changes in other sections of the program [19].

4.4 Defining Significant Change

The potential ripple effect caused by the changes to a package is related to the number of extraneous entities found in the module. Changes in the body of a package would only result in the recompilation of that package body and would not cause ripple effect. Changes to the private part of a package specification might require changes to the package body and, in addition, all client packages would have to be recompiled, even though no changes would be required. Changes to the visible part of the specification, on the other hand, have the potential for causing ripple effect, since these changes would probably result in changes to client packages as well as the body of the package.

A package specification contains visible entities whose change is likely to cause ripple effect, as a result, any addition, deletion, or change in any visible entity will be referred to as a significant change. A significant change is considered likely to cause ripple effect. The following are examples of changes that are not significant: addition, deletion, or change in documentation or style or addition, deletion, or change to the private part of the specification or to the body of the package. These changes will not cause ripple effect.

A good definition of significant change should enable an evaluation to be made semi-automatically. Examples could be found, when examining package specifications by hand, where a subprogram had been included as a test driver and then deleted. This was clearly not a change that would cause ripple effect but it would be regarded as a significant change following the definition. Examples could also be found, however, where a package specification contained only one or two subprograms and their removal clearly represented a significant change in the resources provided by that package. The examination and analysis of the packages in Project $\mathcal{X}$ was a long, tedious process, requiring several months. Since not every package specification could be examined by hand, an objective decision had to be made as to whether a given change would cause ripple effect or not. The simplest solution is to assume the worst case and consider any significant change as having the potential for causing ripple effect.

Another problem to be solved was determining whether or not to count the total number of significant changes to a given package. In examining some package specifications by hand, it could be seen that a visible entity had been added and its predecessor simply left in the specification. In other packages, the predecessor had been deleted. The former would have a smaller count than the latter but, obviously, they were both making the same kind of change. Some specifications were so large that unless hard-copy listings were made of the "differences" file and the two versions, it was impossible to determine exactly how many changes had been made. The easiest and simplest solution to this problem was to let change be measured as binary, either a significant change had been made or not. It retrospect it can been seen that using some kind of enumeration would have allowed a linear correlation with the metric. As results of the statistical validation show in Section 4.2, the binary approach produced a metric that is a good discriminator but shows only a

small correlation with change.

Since only subsystems were available, it was impossible to determine the effect of the changes. If a package were imported by only a few clients, obviously, a change would not have the impact of a change in a package that was imported by a large number of clients.

The tests described in the IEEE draft standard [1] were performed, except for tracking and predictability. Tracking would have been very difficult to do automatically and would have taken too long to do by hand. Since only one system was examined, predictability could not be considered.

The following statistical tests are prescribed for checking the following relationships: correlation - to determine whether there is a sufficiently strong linear relationship between the quality factor and the metric; consistency - to determine whether increasing/decreasing values in the quality factor are accompanied by appropriate increasing/decreasing values of the metric; discriminative power - to determine whether the metric can discriminate between high and low quality components.

The results [14] show that the metric fails the correlation and consistency tests. The metric does not correlate well with change, although there is a low but definite correlation. One reason for this is probably the binary measurement of change and number of design decisions. Fenton [6] argues against looking for a simple linear correlation and states that validation should identify an optimal level of the metric with regard to the quality factor. This advice comes close to that of Card's recommendation of an indicator described in the nex section.

Results of the Mann-Whitney and χ^2 tests [14] show that the revised metric is a discriminator for level of significance of change. The average ranks of the metric for packages exhibiting significant change are much higher than the average ranks for packages that have not exhibited significant change, with $p < 0.0005$. Results of the χ^2 test show a high value of χ^2 and a very small significant level (0.000). This indicates that a value of the revised metric (IH = 1) can discriminate between packages that have undergone significant change and those that have not.

5 Conclusions

Subjective validation applied to the metric produced satisfactory results. This indicates that the metric appears to be measuring what experts would agree is information hiding.

A case study involving several versions of three subsystems of a large Ada program was carried out to determine how a revised version metric was related to the probability of change in modules in succeeding versions. Results showed that there is no linear correlation between the information hiding metric and change but that the metric does discriminate between packages likely to undergo significant change and packages that are not. Those packages with better values for the information hiding metric were less likely to undergo significant change. This kind of change would ultimately produce system deterioration and an increase in system entropy. Appropriate action based on early use of the information hiding metrics would provide some insurance against the insidious deterioration observed by Belady and Lehman [3] in large systems.

These results support the view that the revised information hiding metric satisfies Card's criteria

for an indicator. Card claims that what is needed in an industrial setting is not a measure but an indicator of whether action should be taken [5]. The revised information hiding metric has been field tested in the case study; it can be produced from products and processes that currently exist in an industrial environment; since it depends only on the specification, its value can be determined during high level design. This early determination makes it a high leverage indicator since it can identify a quality attribute or its absence when it is easiest to take appropriate action to improve its value. This early action makes it effective in improving cost, schedule, and quality. Thus, the information hiding metric satisfies all of Card's criteria for an indicator.

References

[1] "Standard for a Software Quality Metrics Methodology," Tech. Rep., IEEE Computer Society, Washington, D.C., 1990. P-1061/D21.

[2] Baker, A.L., Bieman, J.M., Fenton, N., Gustafson, D.A., Melton, A., and Whitty, R., "A Philosophy for Software Measurement," *The Journal of Systems and Software*, vol. 12, no. 3, pp. 277–281, July 1990.

[3] Belady, L. and Lehman, M., "A Model of Large Program Development," *IBM Systems Journal*, vol. 15, no. 3, no. 3, pp. 225–252, 1976.

[4] Calliss, F.W., *Inter-Module Code Analysis Techniques for Software Maintenance.* PhD thesis, University of Durham, Computer Science, 1989.

[5] Card, D.N., "What Makes a Software Measure Successful," *American Programmer*, vol. 4, no. 9, pp. 2–8, September 1991.

[6] Fenton, N., "Software Metrics: Theory, Tools and Validation," *Software Engineering Journal*, vol. 5, no. 1, pp. 65–78, January 1990.

[7] Fenton, N., *Deriving Structurally Based Software Measures.* London, England: Chapman and Hall, 1991.

[8] Finkelstein, L. and Leaning, M.S., "A Review of the Fundamental Concepts of Measurement," *Measurement*, vol. 2, no. 1, pp. 25–34, January-March 1984.

[9] Kafura, D. and Reddy, G.R., "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 3, pp. 335–343, March 1987.

[10] Parnas, D.L., "On the Criteria to be used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, December 1972.

[11] Parnas, D.L., "Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems," Tech. Rep., Naval Research Laboratory, Washington, D.C., June 1977. NRL Report 8047.

[12] Parnas, D.L., "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 2, pp. 128–137, March 1979.

[13] Rising, L.S. and Calliss, F.W., "Problems in Determining Package Cohesion and Coupling," *Software — Practice and Experience.* To appear in.

[14] Rising, L. and Calliss, F.W., "Experiences Validating an Infomation hiding Metric," Tech. Rep., Department of Computer Science and Engineering, Arizona State University, 1992.

[15] Rising, L. and Calliss, F.W., "Using Measurement Theory to Develop an Information Hiding Metric," Tech. Rep., Department of Computer Science and Engineering, Arizona State University, 1992. Technical Report TR–92–011.

[16] Roberts, F.S., *Measurement Theory with Applications to Decision Making, Utility, and the Social Sciences.* Reading, Massachussetts: Addison-Wesley Publishing Company, 1979.

[17] Schneidewind, N.F., "Validating Software Metrics," Tech. Rep., Naval Postgraduate School, Monterey, CA, September 1990. NPS-54-90-019.

[18] Wulf, W. and Shaw, M., "Global Variable Considered Harmful," *SIGPLAN Notices*, vol. 8, pp. 28–34, February 1973.

[19] Yau, S.S., Collofello, J.S., and MacGregor, T., "Ripple Effect Analysis of Software Maintenance," in *Proceedings of the IEEE COMPSAC*, (Chicago, Illinois), pp. 60–65, IEEE Computer Society, November 1978.

[20] Zuse, H. and Bollmann, P., "Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics," *SIGPLAN Notices*, vol. 24, no. 8, pp. 23–33, August 1989.

Through the Interface: A Survey of Usability Testing Methods

Jean Scholtz
Computer Science Department
Portland State University
Portland, OR 97207-0751
jean@cs.pdx.edu

Abstract

Today an abundance of methods are available to usability testing personnel. In order to conduct testing in a timely and cost effective fashion usability evaluators must match the testing necessitated by their particular software product to the types of problems that the various methods are best at discerning. This paper is intended as a survey of the various method presently available and a discussion of the advantages and disadvantages of each.

Dr. Jean Scholtz

Dr. Scholtz is currently an Assistant Professor of Computer Science at Portland State University where she teaches courses on Human-Computer Interaction. She is also the Director of the Center for Software Quality Research. She has worked with many local companies in setting up and conducting usability testing. In 1991 and 1992 she received a NASA/ASEE summer faculty fellowship to look at HCI issues in software being developed at NASA - Kennedy Space Center. Her research interests include issues in user interface design and empirical studies of programmers in the areas of transfer of skill between programming languages and object oriented design.

Through the Interface: A Survey of Usability Testing Methods

Jean Scholtz
Computer Science Department
Portland State University
Portland, OR 97207-0751
jean@cs.pdx.edu

Many software products today claim that they are "user friendly'". While this makes good press release material, the basis for this claim most often is left unspecified. This paper examines possible criteria that could be used to support this claim and presents a survey of the literature on methods to measure the adherence of software to these criteria.

What Constitutes Usability?

The basic human-computer interaction process involves three entities: an interface designer, a system and an end user. The interface designer's task is to turn software specifications into a model which in turn is implemented via interface code. The system then becomes the medium through which this model is portrayed to the end user. The end user's task is to develop a model of the system based on information gathered through use of the system and system documentation. The concern then is how closely the end user's model of the system resembles the designer's model (assuming that the software designer's model is indeed correct). In this respect, the usability of the system is dependent on the capability of the interface to accurately portray system usage and functionality. This in return, assumes that system usage and functionality has been developed in response to an analysis of the user's tasks. Don Norman(1986) uses the terms "gulf of execution" and "gulf of evaluation" to talk about gaps between the system model and the model of the end user. The gulf of execution results from a mismatch of what the end user believes must be done in order to accomplish a specified task and what the system actually expects. The gulf of evaluation measures the distance between the user's perception of what has occurred and what the actual status of the task is. The goal of any sort of usability testing

is to assess these gulfs and make recommendations that will narrow them.

Shackel (1986) states that for a system to be usable the following must be achieved: the range of required tasks must be accomplished at better than some required level of performance, by some required percentage of users, within some specified time, based on some specified relearning time, with flexibility allowing adaptation to some specified percentage of tasks, and with some acceptable level of frustration. Usability design and testing, therefore, involves specifying the required percentages and times, then testing to see that the system conforms to these specifications.

What is it that users want in order to feel that a given system is indeed usable? A study by Holcomb and Tharp (1991) asked users to rate the following seven basic principles as to their importance in a word processing system: functionality, consistency, naturalness and intuitiveness, need for minimal memorization, feedback, user help and user control. Users of word processing systems believed that all principles were important as no principle received an average of less than 74 points out of a possible 100. The functionality principle was rated the highest with an average rank of 94. Consistency was second with an average rank of 81.61. The systems used in this study were word processing systems so it is likely that the ranking of the attributes would change somewhat for other types of systems. However, it should be noted that users considered all seven principles to play a vital part in contributing to software usability.

In addition to the issue of what to test with respect to usability, another issue is when in the software development process testing should be done. Many people perceive usability testing to be a final testing of the software and documentation just prior to release. While some usability testing is done at this time, the vast majority of usability testing should be carried out well in advance; some in fact prior to any system development. The time at which usability defects are discovered and addressed is directly proportional to the cost required to fix them. Therefore, one of the more recent advances in usability testing involves employing theoretical models early in the design phase in order to predict potential usability problems. This paper will give an overview of theory based usability assessments, usability evaluations and classic usability testing. Each type of assessment will be discussed with respect to when in the development process it can be done, what types of problems can be discovered, ease of use, and relative costs to carry out.

Theory Based Usability Testing

This class of testing techniques includes formalizable models of user knowledge. These models range from device level models such as the Keystroke model developed by Card, Moran and Newell (1980) to the models that take the user task into consideration such as ETIT-External Task, Internal Task (Moran, 1983), TAG -Task Action Grammar (Payne and Green, 1986), PUMs - Programmable User Models (Young and Whittington, 1990), and GOMS - Goals, Operators, Methods and Selection Rules (Card et. al., 1983). These models involve formalizing a description of the user interface in order to assess it for learnability, usability and transfer issues. Although these models differ in many aspects, they all have one unifying theme: representation of how users map tasks into actions to accomplish this within the interface.

The GOMS Model

Of these models, the GOMS model (of which the Keystroke model represents one level of analysis) has been most commonly used. The GOMS model can be specified at several different granularities of analysis: a unit task level, a functional level, an argument level and a keystroke level. This model takes into account the user's goal, the methods available for achieving this, a set of operators and a set of selection rules used to choose one of many methods available. The GOMS model has recently been extended from its original use in predicting behavior in text editing to predicting user behavior in a browsing task (Peck and John, 1992). In addition, GOMS has been used to model behavior in tasks in real time and with auditory actions included. (Gray, John, and Atwood, 1992). The GOMS model has proven very useful for modelling routine cognitive behavior. However, it assumes error free behavior and does not factor in potential sources of confusion that might exist in the interface.

Programmable User Models

PUMS (programmable user models) is capable of dealing with errors of procedure. In this model the detailed procedure of the user is not specified beforehand. The PUM uses information about local behavior and properties of the task and interface to predict conceptual errors on the part of the user. An error by the PUM is an indication that users will encounter difficulties in this area.

Encoding an interface in any of these models is a time consuming and often creative task. That is, designers find that many times a one

to one mapping between their interface and the formalism does not exist. Therefore, much of their effort is in finding a way of expressing the interface in the formalism. Most of the formalisms are not well defined. No hard and fast rules exist for encoding an interface. However, the activity of attempting to specify various portions of an interface in a formalism can alert a designer to potential problems.

Use of Theory Based Models

What sort of problems can be uncovered using these models and how? Basically, designers can discern possible learnability problems and usage problems. Inconsistent rules can be identified from an overabundance of rule forms. Increased training times for new systems can be identified by noting that two systems have many distinct productions. Conceptual errors which cause users to follow an ill-fated course of action can be discovered. For example, a GOMS analysis is based on the hierarchy of procedures that a user must learn in order to accomplish a task. This model can predict learning time and execution time and aids in examining the consistency of the procedures. The evaluation for the GOMS analysis is done using values that the model predicts given the design specifications. These models can be applied prior to the existence of any system. Therefore, usability problems can be detected during design, long before system implementation.

A designer would be ill advised to attempt to encode an entire system in any of these formalisms. However, by picking an appropriate formalism and a questionable portion of an interface, this technique could result in uncovering potential problems early in the design stage. Moreover, using analytical models rather than empirical data allows carry over from one design to the next. Since what is being evaluated are design specifications and task procedures rather than a more specific screen design, there is a much greater possibility of taking results from one evaluation and applying them to other project designs. Usability tests of specifics tend to have weak generalizations, if any.

Cognitive Walkthrough

Another theoretical method of early evaluation is the cognitive walkthrough technique developed by Lewis and Polson (1990). This method is used by developers of an interface to step through tasks that a user would typically accomplish. The actions and feedback specified are examined to see if they facilitate the user in achieving his goal. At each step of the walk through, specialists identify the knowledge the user must have and possible sources for obtaining this

knowledge. Again, this technique can be used early in the developmental cycle and can be accomplished using only a design. Unlike the previously discussed formalisms, there is no time consuming encoding of an interface. A computerized version of this system has been developed to aid in asking questions at each step and evaluating the final results (Reiman et. al., 1991). The time required to do this technique is considerable although the computerized version should help somewhat. The cognitive walkthrough was originally developed for walk up and use situations but it has been extended recently to more complex interfaces (Wharton, Bradford, Jeffries, Franzke, 1991). Several important issues concerning the use of the cognitive walkthrough technique are addressed in this paper.

1. Due to the time consuming nature of the evaluation, the tasks to be evaluated have to be limited. The walkthrough method does not specify how these task should be selected. It is up to the evaluators to select a representative set of tasks. Issues such as how many tasks and at what granularity the evaluations should be done must be addressed by the evaluators.

2. Evaluators need to have some knowledge of cognitive science in order to perform the walkthrough successfully. In addition, this method is more successful when carried out by a group of evaluators.

3. Interpretation of the results of the cognitive walkthrough needs to be done with respect to the larger picture of the entire interface rather than one particular task.

This study concludes that a basic problem with the cognitive walkthrough is its focus on lower level interface issues.

Karat et. al. (1992) found that walkthrough evaluators favored using scenarios to conduct their evaluations rather than using a self-exploratory method. Scenarios constructed by actual users can help to ensure that evaluation will identify problems that might be encountered in actual use.

The Cognitive Jogthrough

A faster paced version of the cognitive walkthrough,the cognitive jogthrough (Rowley, 1992), utilizes video recording and an interactive evaluation session. This method could be used to evaluate several proposed alternative interface designs. Although still time consuming, more of the interface can be evaluated in the same time frame using this technique rather than the walkthrough.

Usability Evaluations

An evaluation differs from classic usability testing in that it does not involve actual users carrying out tasks on the specified system.

Rather, trained User Interface personnel or others using sets of guidelines evaluate a system with respect to principles of user interface design. Two techniques, heuristic evaluation and use of guidelines, will be discussed here.

Heuristic Evaluation

Heuristic evaluation is done by an expert in usability issues. It is done keeping in mind a product's users and tasks as well as general human factors guidelines. Jakob Nielson (1990) terms this " heuristic evaluation of interfaces". People perform this evaluation on the basis of their own common sense and intuition. Studies run by Nielson show that for heuristic evaluation to be useful, a group of evaluators is needed. Data collected suggests that five evaluators can determine a large proportion of the problems. Little planning is required to do this and therefore, the cost is low. However, an interface (or at least a prototype) needs to exist.

One issue is what amount of expertise the evaluators should have in order to be effective. In Nielson's experiment heuristic evaluations were done by nonprofessional evaluators who had knowledge of basic principles for usability. A study by Jeffries et al. (1991) used usability experts as evaluators. Nielson (1992) conducted a study of heuristic evaluations done by three groups of evaluators: novices with no experience, usability experts, and usability experts who also had domain experience. As could be expected, usability experts with domain experience were much more proficient at finding usability problems then the other two groups. If one has this kind of expertise available, the size of the group can be limited to only two or three evaluators. Using this technique, the study showed that major problems in usability have a higher probability of being found than do minor problems. Particular problems that are difficult to find are those of missing interface elements and lack of clearly marked exits.

Guideline Evaluation

The use of guidelines for assessment is another evaluation technique. This involves using published guidelines such as those by Smith and Mosier (1986) and evaluating the interface for conformance. Again, a prototype of a rather complete nature needs to exist in order to use this technique. Using guidelines as a means of assessing usability is difficult for three reasons. First, the number of guidelines is overwhelming. In fact, many guidelines conflict with one or more other guidelines. It is difficult to interpret many of the guidelines to specific cases. Therefore, even if one knows all of the guidelines, it is difficult to assess if a specific interface has correctly

incorporated it. Secondly, even if guidelines have been followed, the usability of the system is not ensured. Many guidelines address different levels of user experience. If the guidelines followed have not been correctly matched to the user population, the interface will not be usable - at least by its intended users. Thirdly, the sets of guidelines are large and an untrained developer could easily overlook problems due to an unawareness of certain guidelines.

A study by Tetzlaff and Schwartz (1991) on the use of guidelines during design showed that designers missed many concepts. There is no reason to believe that better performance during evaluation would be expected. Therefore, again a group evaluation is recommended. Furthermore, this evaluation technique must be used by skilled User Interface personnel rather than developers who are not knowledgeable in interface design guidelines. When carried out in this fashion, guidelines can be used in a timely fashion to discover many problems. Guidelines can be used when only a prototype, although a rather developed one, exists. The cost is that of the group of UI specialists selected to do the evaluation.

Usability Testing

Usability testing is testing in which actual users are involved in the testing procedure. Although most envision usability testing as an activity done near the end of the product development, usability testing encompasses a wide range of testing from focus groups prior to design to testing on prototypes to beta testing and questionnaires after release. At each stage of development different types of information can be obtained. The table below summarizes the kind of information that can be obtained for each stage of development and specifies several methods for obtaining this.

This table is by no means a complete list of what can be obtained and how. It is intended as a guide to testing at each stage of development. Using prototypes of differing levels of development provides a vehicle for doing usability testing earlier in the development process.

when	how	what
early development	focus group	how users think about tasks
	surveys	problems with current system
	field studies	tasks users do
mid development	prototype	ambiguities, confusion
late development	beta test	ease of use
	lab test	learnability
after release	questionnaires	how users feel about product, additions for next release
	performance comparisons	time to do tasks, errors how product compares to others

table 1: Usability Testing in the Design Cycle

Usability testing can also be used to determine if predetermined usability criteria have been met. In this case, the usability criteria should also specify data that will determine if the criteria has indeed been met by the implemented system.

Data from usability testing can be classified as either quantitative or qualitative. Examples of quantitative data usually collected are number of errors made in specified tasks, time to complete specified tasks, or number of tasks correctly done. Qualitative data can be obtained from verbalizations users make while they use the system and attitude questionnaires. Data on errors made can be used both quantitatively (number made) and qualitatively (kind of errors made and why). Additionally, later usability testing can be carried out either in a controlled laboratory or in a more realistic field testing environment. Use of a controlled study ensures that the desired portion of the system does indeed get tested. The disadvantage is that there is no way to be certain that this particular task will get carried out in this particular method in the user's normal work setting. On the other hand, field testing tests the system in the user's normal environment. The disadvantage of field testing is that only the portion of the system that the given user normally uses is tested.

The main drawback to any of these usability tests is the planning required. Although usability testing experience helps in designing and conducting new tests, every test is unique and therefore, constitutes a large amount of effort to design. Gathering representative groups of users together for a valid test is a lengthy and costly process. The time delay often precludes use of some of these techniques. Except for some of the larger corporations, the expertise needed to design and conduct usability tests is not readily available. That is not to say that this type of usability testing should never be used. In order to answer specific concerns about ease of use of a given product the most definitive answer can only be given by the actual user. Moreover, usability testing has an advantage over the other types of evaluations in that it can encompass the entire product. That is, documentation, on-line help, tutorials, and the environment in which the product is used are assessed during full scale usability testing. But this large scale comes with a high cost.

Developing questionnaires, surveys and scenarios for usability testing also requires considerable expertise. A special issue of IEEE Transactions on Professional Communications edited by Ramey (1989) is an excellent source of information on usability testing.

A Comparison of the Techniques

Jeffries et al., (1991) conducted a study of four usability evaluation methods: heuristic evaluation, guidelines, cognitive walkthrough, and usability testing. Four groups, each using one of the methods, evaluated a pre-release of some software. A total of 223 problems were discovered. Of these the heuristic evaluation found 121, usability testing 32, guidelines 35 and the cognitive walkthrough 35. In all but the heuristic evaluation technique some percentage of problems were found either by side effects or prior experience (3% for usability testing, 14% for cognitive walkthrough, and 57% for guidelines evaluation). The benefit/cost ratios were measured as the number of problems found per person-hour. Heuristic evaluation had a four to one advantage. Cognitive walkthroughs were next, followed by guidelines and then usability testing. The evaluations techniques also differed on the type and severity of problems they found. The cognitive walkthrough missed general and reoccurring problems. Heuristic evaluation identified by far the most problems, however many of these were very low priority problems. This technique does not help evaluators rate the priority of problems. Guideline evaluation also missed a large number of serious problems. Usability testing was good at identifying serious problems, reoccurring problems and general problems. Moreover, low priority problems were ignored

by subjects in usability tests. However, it cost the most of the four techniques used.

Desurvivre (1992) compared usability testing with a cognitive walkthrough and a heuristic evaluation technique. This study found that the heuristic evaluation done by experts found more problems than were actually found in the laboratory usability test. Two other groups of evaluators, system designers and non-experts, were quite low in the number of problems they found in both heuristic evaluation and in the cognitive walkthrough. Not only did experts identify more problems than discovered in the lab, they also listed problems that could be classified as improvements as well as potential problems.

A study by Wong et al. (1992) showed that usability testing done on a task by task basis failed to identify as many errors as when an entire scenario test was conducted. This suggests that doing usability testing on only a small portion of the interface may fail to identify problems that occur in the larger context of the whole system.

Finding the problem is just the beginning: the problem must then be eliminated. How do HCI personnel or software designers go about making recommendations for changes that will improve the interface? If empirical testing is being done, users may suggest changes that they feel would help them. Often the problem suggests the change. This is the case when the sequence provided by the interface does not match the sequence which the user wants to perform. Karat et al (1992) points out that if many problems are discovered in a particular area of an interface, designers will have a greater source of information to use in suggesting changes.

Types of Software Used in Usability Evaluations

One natural question to ask is "what types of software have been used in comparing usability testing methods"? The types of software used in usability comparisons and in illustrations of a sole technique are quite varied. The cognitive walkthrough in the Wharton et al. study was carried out on three systems: HP-VUE, a visual interface to the Unix operating system; REPS, an information system for sales representatives; and BCA, a CAD tool for electrical engineers. Cognitive walkthroughs have also been carried out on graphical programming languages. The cognitive jogthrough was applied to a proposed user interface for a chromatography workstation. Individual and team comparisons in a cognitive walkthrough (Karat et al., 1992) were based on two GUI office environments which included spreadsheets, graphics applications and text applications. The GOMS model has been used to study such diverse systems as help browsers in programming languages, the behavior of NASA test directors during shuttle

countdowns, and the behavior of telephone operators. Heuristic evaluations have also been carried out on several computer information systems accessed through touch-tone telephones. The study involving the use of guidelines during design (Tetzlaff, 1991) used specifications for software to manage a company's recreational sports teams.

Summary

How does one decide what kind of usability testing to do? Of course, the best of all possible worlds would be to do all of it. This is usually prohibited by time and budget constraints. Each product must be examined to determine what aspect of testing is most needed. Products that are modifications or revisions of earlier releases will most likely have feedback from the prior release available. In this case, testing at later stages of development would provide the most needed feedback. Under these conditions problems have already been identified and the concern is if the modifications have sufficiently improved those. A newly developed product, on the other hand, will benefit more from earlier theory based testing to address overall design concerns.

In addition, the cost benefit analysis aspect of testing must be factored in. In addition to the Jeffries et al. (1991) study discussed earlier, papers on cost benefit analysis by Mantei and Tory (1988), Karat (1990a, 1990b), and Mayhew (1990) can provide a framework on how to estimate the potential benefits and weigh those against the calculated cost of testing. Costs of each type of testing can be weighted along with a prioritized list of the information needed.

Virzi (1990) showed that in usability testing 80% of the problems were discovered by between four and five subjects. This suggests that even a small number of subjects can identify any major problems.

A small amount of usability assessment of some sort far exceeds no testing. The bottom line is that usability testing always gets conducted. The question is whether the developer or the user conducts the testing.

References

Card, S., Moran, T. and Newell, A. (1980). The Keystroke-level Model for User Performance Time with Interactive Systems. *Communications of the ACM,* 23 (7), 396-410.

Card, S., Moran, T. and Newell, A. (1983). *The Psychology of Human-Computer interaction.* Hillsdale, NJ: Erlbaum.

Desurvire, H. , Kondziela, J., and Atwood, M., (1992), What is Gained and Lost When Using Methods Other Than Empirical Testing, Presented as a short talk at CHI'92.

Gray, W., John, B., and Atwood, M. (1992). The Precis of Project Ernestine or An Overview of a Validation of GOMS. In P. Bauersfeld, J. Bennett, & G. Lynch (Eds) *Proceedings of CHI'92,* New York: ACM Press, 307-312.

Holcomb, R. and Tharp, R. (1991). What Users Say about Software Usability. *International Journal of Human-Computer Interaction.* 3(1), 49-78.

Howes, A. and Payne, S.J. (1990). Display-Based Competence: Towards user models for menu driven interfaces. *International Journal of Man-Machine Studies,* 33, 637-655.

Jeffries, R., Miller, J., Wharton, C. and Uyeda, K. (1991). Evaluation in the Real World: A Comparison of Four Techniques. In S. Robertson, G. Olson & J. Olson (Eds.), *Proceedings of CHI '91,* New York: ACM Press, 119-124.

Karat, C. (1990). Cost-Benefit Analysis of Usability Engineering Techniques, *Proceedings of the Human factors Society 34th Annual Meeting,* 839-843.

Karat, C. (1990). Cost-Benefit Analysis of Iterative Usability Testing. In *Proceedings of the Third IFIP Conference on Human-Computer interaction: INTERACT '90,* Cambridge, England.

Karat, C., Campbell, R, and Fiegel, T. (1992). Comparison of Empirical Testing and Walthrough methods in User Interface Evaluation. In P. Bauersfeld, J. Bennett, & G. Lynch (Eds) *Proceedings of CHI'92,* New York: ACM Press, 397-404.

Larkin, J. (1989). Display-based problem solving. In Klahr, D. & Kilovsky, K. (Eds) *Complex Information Processing,* Hillsdale, NJ: Lawrence Erlbaum. 319-341.

Mantei, M. and Teorey, T. (1988). Cost/Benefit Analysis for Incorporating Human Factors in the Software Lifecycle. *Communications of the ACM,* 31(4), 428-439.

Mayhew, D. (1990). Cost-Justifying Human Factors Support - A Framework, *Proceedings of the Human factors Society 34th Annual Meeting,* 834-838.

Nielson, J. (1992). Finding Usability Problems Through Heuristic Evaluation, In P. Bauersfeld, J. Bennett, & G. Lynch (Eds) *Proceedings of CHI'92,* New York: ACM Press, 373-380.

Nielson, J. and Molich, R. (1990). Heuristic Evaluation of User Interfaces. In J. Chew and J. Whiteside (Eds), *Proceedings of CHI'90,* New York: ACM Press, 249-256.

Norman, D. (1986). Cognitive Engineering. In D. Norman and S. Draper (Eds), *User Centered System Design: New Perspectives on Human-Computer Interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Payne, S. and Green, T.R. G. (1986). Task-action grammars: a model of the mental representation of task languages. *Human-Computer Interaction.* 2(2), 93-133.

Peck, V. and John, B. (1992). Browser-Soar: A Computational Model of a Highly Interactive Task, In P. Bauersfeld, J. Bennett, & G. Lynch (Eds) *Proceedings of CHI'92,* New York: ACM Press, 165-172.

Polson, P. and Lewis, C. (1990). Theory Based Design for Easily Learned Interfaces. *Human-Computer Interaction.* 5, 191-220.

Ramey, J. (Ed). (1989). special issue: Usability Testing. *IEEE Transactions on Professional Communications.* 32(4).

Rowley, D. and Rhoades, D. (1992). The Cognitive Jogthrough: A fast-paced user interface evaluation procedure, In P. Bauersfeld, J. Bennett, & G. Lynch (Eds) *Proceedings of CHI'92,* New York: ACM Press, 389-395,

Reiman, J., Davies, S., Haer, D., Esemplare, M., Polson, P. and Lewis, C. (1991). An Automated Cognitive Walkthrough. In S. Robertson, G. Olson & J. Olson, (Eds). *Proceedings of CHI'91,* New York: ACM Press, 427-428.

Shackel, B. (1986). Ergonomics in design for usability. In M.D. Harrison & A.F. Monk (Eds.), *People & Computers: Designing for Usability. Proceedings of the Second Conference of the BCS HCI Specialist Group.* Cambridge: Cambridge University Press.

Smith, S. and Mosier, J. (1986). Guidelines for Designing User Interface Software, Report MTR-10090, The MITRE Corp, Bedford, MA.

Tetzlaff, L. and Schwartz, D. (1991). The use of Guidelines in Interface Design. In S. Robertson, G. Olson & J. Olson (Eds). *Proceedings of CHI'91,* New York: ACM Press, 329-333.

Virzi, R. (1990). Streamlining the Design Process: Running Fewer Subjects, *Proceedings of the Human Factors Society 34th Annual Meeting,* 291-294.

Wharton, C., Bradford, J., Jeffries, R. and Franzke, M. (1992). Applying Cognitive Walkthrough to More Complex User interfaces: Experiences, Issues and Recommendations. In P. Bauersfeld, J. Bennett, & G. Lynch (Eds) *Proceedings of CHI'92,* New York: ACM Press, 381-388.

Wong, A, Donkers, A., Dillon, R. and Tombaugh, J. (1992) Is the Whole Test Greater Than the Sum of its Parts? Presented as a poster at CHI'92.

Young, R., Green, T.R.G. and Simon, T. (1989) Programmable User Models for Predictive Evaluation of Interface Designs. In K. Bice and C. Lewis (Eds) *Proceedings of CHI'89,* New York: ACM Press, 15-20.

Young, R. and Whittington, J. (1990). Using a Knowledge Analysis to Predict Conceptual Errors in Text -Editor Usage. In J. Chew & J. Whiteside (Eds). *Proceedings of CHI'90,* New York: ACM Press, 91-98.

Debugging Parallel Code using a Line Profiler

Wayne E. McDaniel
Intel Corporation
Supercomputer Systems Division C01-01
15201 N.W. Greenbrier Parkway
Beaverton, OR 97006

e-mail: wayne@ssd.intel.com

Abstract

Traditionally, profiling has been used to identify the code that executes the most often. However, profiling can also be used to identify the code that executes during a program failure. This paper presents profiling as a debugging technique, which uses the frequency counts produced by line profiling program failures to understand what went wrong.

Biography

Wayne McDaniel is a senior software systems engineer in the Supercomputer Systems Division of Intel Corporation. He is part of a team that is responsible for sustaining the system software for Intel's iPSC/860® parallel supercomputer. He is primarily responsible for sustaining the NX/2 parallel operating system but also works on the Concurrent File System. He is currently working on a code inspection process for bug fixes, which he hopes will reduce the chances of side effects, find additional bugs in related code, and serve as an educational experience for each member of the sustaining team.

Before joining Intel, Wayne developed real-time operating systems at Texas Instruments and worked on remote access automation at Boeing Computer Services, Richland. He is also author of an automation tool that he distributes as shareware.

Debugging Parallel Code using a Line Profiler

Wayne E. McDaniel

Intel Corporation
Supercomputer Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, OR 97006

1. Introduction

The NX/2 operating system is the system software that supports message passing and process control on the Intel iPSC/860® parallel supercomputer. The NX/2 operating system executes independently on each of the up to 128 compute nodes. Each compute node has an Intel i860® microprocessor, its own memory, and an interface to the hypercube message passing network. The iPSC/860®, along with the NX/2 operating system, provides an environment for development of large scale parallel applications.

Debugging distributed memory parallel operating systems is a difficult task. For the NX/2 operating system, the difficult task is eased somewhat because of the built-in debugger that provides the basic debugging operations, such as, displaying memory contents, setting breakpoints, disassembling instructions, and displaying message queues. However, even with a good debugger, debugging parallel code is still difficult. Setting breakpoints and single stepping code alters the execution timing between nodes to a point where the circumstances required to reveal the bug cannot be guaranteed. Therefore, to successfully debug parallel code, a debugging method that allows each node to execute at full speed needs to be supported.

I decided to use a debugging method that worked for me back in 1986 when I worked for Boeing Computer Services, Richland. I found that by simply knowing the number of times each source statement executes, it was possible to find and fix bugs. At Boeing, I was responsible for sustaining a large terminal emulation and file transfer program, which I did not write. Line profiling this program saved hours (and hours) of debugging time. To solve a bug, a profiled version of the program was executed at full speed in an environment designed to reveal the bug. Then, after the program finished executing, a special printout was made showing the source code next to the frequency counts. It was possible to study the source code knowing exactly how many times each source statement executed, which always seemed to provide an answer. This paper will show that it is

possible to solve very difficult bugs by studying the frequency counts produced when line profiling program failures.

2. Solving Bugs in the NX/2 Operating System

A big problem area in the NX/2 operating system was supporting interrupt messages. Interrupt messages, when received by a node, will cause normal processing to be interrupted and a special interrupt handler to be invoked. Unfortunately, due to bugs, the applications programmer was not allowed to make calls to *printf* from within these handlers. And, since it was not understood why *printf* failed, it was a common practice to limit the functionality of interrupt handlers to simple statements and avoid making system calls. Clearly, this was not an acceptable limitation.

This bug presented a fair challenge for me because I was not involved in the original development. I was new to the NX/2 operating system and to the iPSC/860® parallel supercomputer. And, to make things worse, setting breakpoints and single stepping code altered the execution timing and spoiled the environment required to create this bug. If I had been involved in the original development, I would have gained detailed knowledge of the software and would be able to "execute" the code in my thoughts. Original authors use their experience gained during development to find and fix bugs during sustaining. The advantage of this experience is very significant. Original authors can visualize program execution and automatically filter out areas that are least likely to cause the bug or are known not to execute. When they review code, detailed knowledge of what executes implicitly guides them to the "interesting areas." Since I could not visualize program execution in the NX/2 operating system, I decided to used a line profiler to guide me to the "interesting areas."

Since it was not clear where to start looking for the interrupt message bug, every source file that appeared even remotely related to this bug was instrumented with profile statements. (This totaled about 100 pages of code.) Once the code was instrumented, two profiles were done. The first profile was made using a test program that was known to work. The second profile was made using a test program that was known to fail. The frequency counts from the successful execution was compared with the frequency counts from the failed execution. The following code segment taken from this set of experiments shows the number of times each node decrements an important variable.

```
 Node 1    Node 0    Node 1    Node 0
(failed)  (failed)  (worked)  (worked)

2         | 3       | 5       | 5       |     PF_ON(68);
          |         |         |         |     iod = _fio_iod[sel];
          |         |         |         |     iod->busy--;
```

The two columns next to the code are the frequency counts for the experiment that worked. The other two columns show the frequency counts for the experiment that failed. The PF_ON(68) statement is from the profiling tool, which expands into 'profil_buf[68]++' by the C preprocessor. The frequency counts show an important difference between the experiment that worked and the experiment that failed.

Profiles reveal the "interesting areas." Out of 100 pages of source code, more than half was ignored. It was possible to ignore most of the code that did not execute and the code that matched frequency counts for both experiments. The frequency counts reveal the interesting areas by showing unexpected or unusual results. The following profile, for example, shows that node 1 gets stuck in an endless loop.

```
 Node 1    Node 0    Node 1    Node 0
(failed)  (failed)  (worked)  (worked)

3         | 3       | 5       | 5       |     PF_ON(37);
          |         |         |         |     if (!iod->async) {
3         | 3       | 5       | 5       |         PF_ON(38);
          |         |         |         |         while (iod->busy || iod->count != 0) {
58381     | 3       | 5       | 5       |             PF_ON(39);
          |         |         |         |             if (_sync_handler) {
          |         |         |         |                 void (*hdlr)();
          |         |         |         |                 int trapsave;
          |         |         |         |
2         | 3       | 5       | 5       |                 PF_ON(40);
          |         |         |         |                 trapsave = masktrap(1);
          |         |         |         |                 hdlr = _sync_handler;
2         | 3       | 5       | 5       |                 PF_ON(41);
          |         |         |         |                 if (hdlr) {
2         |         |         |         |                     PF_ON(42);
          |         |         |         |                     _sync_handler = 0;
          |         |         |         |                     hldr(0, _sync_handler_count);
2         | 3       | 5       | 5       |                     PF_ON(43);
          |         |         |         |                 }
2         | 3       | 5       | 5       |                 PF_ON(44);
          |         |         |         |                 masktrap(trapsave);
          |         |         |         |             } else {
58379     | 0       | 0       | 0       |                 PF_ON(45);
          |         |         |         |                 flick();
          |         |         |         |             }
          |         |         |         |         }
          |         |         |         |     }
```

When the experiment works correctly, each node executes this code 5 times and the *flick()* routine is never called. This is not true for the experiment that failed. Node 1 enters an endless loop calling the *flick()* routine over and over. Node 0 is blocked waiting for node 1 to do something useful.

This profile clearly showed that node 1 was in an endless loop. (This was easy to identify.) However, it was not easy to identify why. Understanding why required hours of analysis. The analysis starts by identifying the code most likely involved by tagging a printed listing of the profiled source with page markers. Each page marker identifies code that has unusual or unexplainable frequency counts. The unusual or unexplainable frequency counts serve as a starting point for the investigation. Other pages are added to the investigation as the code is reviewed and understood. Also, as part of the investigation, questions, concerns, and observations about the source code and the frequency counts are written down on the listing to help understand the code and to help think through what executed during the failure.

The interrupt message bug was found using this technique. As it turns out, the *_sync_handler* variable, shown in the code segment above, was defined globally and was not properly protected for reentrant code. The *_sync_handler* variable gets cleared by an unrelated execution thread causing the *flick()* routine to be continuously called. This is what happens: While node 1 is in the middle of calling *printf*, node 0 sends an interrupt message. When node 1 receives the interrupt message, the *printf* execution is suspended and the interrupt handler is invoked. The interrupt handler then calls *printf*, which successfully executes from start to finish. However, the global variable *_sync_handler* was clobbered in the process. When execution resumes with the original *printf*, the *_sync_handler* variable was incorrectly set to zero. Since *_sync_handler* is zero, node 1 enters an endless loop.

Essentially, by using this debugging technique, bugs are found by inspection. The frequency counts are used to focus the inspection and provide proof about what executes. The inspection is fueled by constant questions about why the counts ended up as they did. The inspection is also fueled by comparing frequency counts of experiments that worked with experiments that failed.

3. Using Line Profilers to Debug Code

There are a lot of great stories about line profilers finding performance bugs (BENTL82, BENTL87,

KNUTH71, SCHRI85, and many others). However, it was difficult to find references where line profilers were being used to solve bugs, especially difficult bugs involving race conditions and starvation. Satterthwaite (SATTE72), Knuth (KNUTH71), Graham, Kessler, and McKusick (GRAHA83) mention some of the benefits of using profilers to debug code. However, in most cases, finding bugs was merely a side effect of using profiles to identify areas for performance improvements. On the other hand, Jon Bentley did provide a concrete example of using line profilers to solve difficult bugs in BENTL87. From BENTL87 (Copyright 1987 ACM):

> "A friend at Bell Northern Research implemented statement counts one weekend in a real-time phone switching software system with multiple asynchronous tasks. By looking at the unusual counts, he found six bugs in the field-installed code, all of which involved interactions between different tasks. One of the six they had been trying (unsuccessfully) to track down via conventional debugging techniques, and the others were not yet identified as problems (i.e., they may have occurred, but nobody could attribute the error syndrome to a specific software bug)."

Here, a profile was made with the intention of identifying bugs, instead of, identifying code for performance improvements.

Improving the performance of programs is a natural for profilers, since profilers identify the code that executes the most often. And, since profilers do such a nice job improving the performance of programs, they are quite popular and are supported directly by many compilers. After a profiled program executes, a new source file meant only for viewing is automatically created with the frequency counts collated next to the source statements they represent. However, compiler supported profilers are not well suited for debugging code. Most require the profiled program to terminate normally. To support debugging, a profiler must allow access to the frequency counts at anytime during program execution and allow access regardless of how the program terminates. Without this access, many bugs would be impossible to resolve.

Getting the frequency counts written to a file in an appropriate format can be difficult. It depends on the bug and the computing environment. For example, the bug may cause the program to not terminate normally, like the interrupt handler bug that was discussed earlier. Or, even if the program terminates normally, the computing environment may not support a file system to dump the frequency counts at the exit points. Also, even if dumping the frequency counts at the exit points is possible, it may not be the best way to isolate a bug.

Many times, knowing the frequency counts between two breakpoints reveals important information about a bug. At the first breakpoint, the frequency counts are set to zero. Then, at the second breakpoint, the frequency counts are saved. Again, the best collection method depends on the bug and the computing environment.

4. A Simple Profiler

I wrote the profiler used to debug the NX/2 operating system. To make profiling possible, three programs were written. The first program, which is used only once, preprocesses a source file by adding profile statements after each *begin-block* character, i.e., each '{' character. Once the profile statements have been added to the source file, they become a permanent addition to the source. The next program assigns a unique number to each of the profile statements. These numbers index into an array that will keep track of the frequency counts as the profiled program executes. The last program combines the frequency counts collected in the array with the source code to produce a new source file meant only for viewing. This new source file has the frequency counts listed in columns next to the source statements they profiled.

The following program, which counts the number of lines in a file, will be used to illustrate the steps taken to profile code.

```
main()
{
int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF) {
        if (c == '\n') {
            nl++;
        }
    }

    printf("%d\n", nl);
}
```

Step 1: Add the profile statements to the source using the command: 'prepf -b line.c'. The line.c program will now look like the following.

```
main()
{
PF_ON();
int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF) {
        PF_ON();
        if (c == '\n') {
            PF_ON();
            nl++;
        }
    }

    printf("%d\n", nl);
}
```

The *prepf* program is used to save edit time. It uses a simple algorithm that adds a profile statement after each '{' character. Therefore, additional editing of the source file may be required. For the above example, the first "PF_ON()" statement would need to be moved down one line.

Step 2: Assign integers to each of the profile statements using the command: 'pset line.c'. After executing the *pset* command, the line.c program would be as shown. (The *pset* program is used when PF_ON() statements are added, deleted, or moved around.)

```
main()
{
int c, nl;

    PF_ON(1);
    nl = 0;
    while ((c = getchar()) != EOF) {
        PF_ON(2);
        if (c == '\n') {
            PF_ON(3);
            nl++;
        }
    }

    printf("%d\n", nl);
}
```

Step 3: Compile and execute line.c. As the line.c program executes, the frequency counts are tallied in the global array *profil_buf*. (The C preprocessor expands each PF_ON statement to profil_buf[0]++, profil_buf[1]++, profil_buf[2]++, and so on.)

Step 4: Use an appropriate method of storing the frequency counts to a file.

Step 5: Create a new file that lists the frequency counts next to the source statements using the command: 'profile line.c'. Here is what *line.p* will look like.

```
        | main()
        | {
        | int c, nl;
        |
1       |    PF_ON(1);
        |    nl = 0;
        |    while ((c = getchar()) != EOF) {
675     |        PF_ON(2);
        |        if (c == '\n') {
28      |            PF_ON(3);
        |            nl++;
        |        }
        |    }
        |
        |    printf("%d\n", nl);
        |}
```

The *profile* program creates the multiple column feature used to profile nodes on the iPSC/860 parallel supercomputer using a command line switch. This switch can also be used to compare different profiles of a sequential program. A configuration file can also be specified from the *profile* command, which allows profiling multiple file programs and selectively turning off profiling.

Profilers, like this one, are easy to write. This one was written in about two weeks.

5. Conclusion

Most everyone that has successfully used a profiler concludes that a program's profile is extremely valuable. My conclusion is no different. I have found that by simply studying the frequency counts produced by a line profiler, it is possible to solve extremely difficult bugs. The key is to focus on profiling the bug and not necessarily the entire program. Pick the modules that most likely contain the error, then add enough profile statements to get a good profile of what happened. The goal is to cause the bug by executing the minimum number of statements, which in turn, give the frequency counts the most meaning. If possible, compare the frequency counts of a successful execution with one that fails. The frequency counts document exactly what executed when the bug occurred. This serves as a map that will guide the debugging effort. Time is not wasted studying code that does not execute or executes the expected number of times.

Sometimes the frequency counts may not reveal the problem. The important thing, however, is the analysis

is almost guaranteed to make other debugging techniques be more effective. The frequency counts will identify places for breakpoints, variables to trace, code to research, additional code to profile, and so on. The frequency counts will also identify places to add debug code, which in turn can be profiled.

Actually, debugging with frequency counts is a type of postmortem analysis. Postmortem debugging techniques were once very popular. Bugs were solved by analyzing dumps that captured the state of the system when the bug occurred. Hopefully, future computing environments will support highly configurable postmortem tools that will present information in terms of the source code. Source level postmortem debugging tools will provide sustaining engineers with an important method for solving customer problems. Customers will be able to collect the raw debug information for later analysis. This is important because often times problems only occur at the customer site and any attempts to recreate the problem locally fail. Their system may be used or configured in different ways exposing stress, race, and starvation bugs for the first time. Postmortem debugging techniques will not (and should not) replace interactive debugging techniques but simply provide an alternative.

Using a line profiler as a postmortem debugging tool is easy and extremely valuable. Virtually any piece of software can benefit by frequency count analysis, especially unfamiliar code or code that doesn't fail when using interactive debugging methods.

6. References

(ARAL88) Aral, Z. and Gertner, I. "Parasight: A high-level debugger/profiler architecture for shared-memory multiprocessor," *Conference Proceedings. 1988 International Conference on Supercomputing*, July 1988, pp. 131-139, ACM.

(BENTL82) Bentley, J. L. *Writing Efficient Programs*, Prentice-Hall, Englewood Cliffs, N. J., 1982.

(BENTL87) Bentley, J. L. "Profilers—Programming pearls," *Communications of the ACM*, July 1987, pp. 587-592, Volume 30, Number 7.

(GRAHA83) Graham, S. L., Kessler, P. B., and McKusick, M. K. "An execution profiler for modular programs," *Software—Practice and Experience 13*, 1983, pp. 671-685.

(KERNI76) Kernighan, B. W., Plauger, P. J. *Software Tools*, Addison-Wesley, 1976.

(KNUTH71) Knuth, D. E. "An empirical study of FORTRAN programs," *Software—Practice and Experience 1*, April-June 1971, pp. 105-133.

(RAGSD91) Ragsdale, S. (editor), Asbury, R., Bain, W. L., Barton, M., Brandenburg, J. E., Jackson, J. V., Kushner, E. J., and Scott, D. S. *Parallel Programming*, Intel/McGraw-Hill Series, 1991.

(SATTE72) Satterthwaite, E. H. "Debugging tools for high level Languages," *Software—Practice and Experience 2*, July-September 1972, pp. 197-217.

(SCHRI85) Schrijver, H. "Profiling and debugging in modern programming languages," Computer Physics Communications 38, North-Holland Physics Publishing Division, Elsevier Science Publishers B. V. 1985, pp. 289-293.

(UNIX89) *Unix System V/386 Programmer's Guide, Vol II, Release 3.2*, Profiling Documentation, Prentice-Hall, Englewood Cliffs, N. J., 1989, pp. 18-29 through 18-60.

Toward Automatic Localization of Software Faults*

Hsin Pan *Eugene H. Spafford*

Software Engineering Research Center
Purdue University
West Lafayette, IN 47907–1398
(317) 494–7825
{pan, spaf}@cs.purdue.edu

Abstract

Developing effective debugging strategies to guarantee the reliability of software is important. By analyzing the debugging process used by experienced programmers, four distinct tasks are found to be consistently performed: (1) determining statements involved in program failures; (2) selecting suspicious statements that might contain faults; (3) making hypotheses about suspicious faults (variables and locations); and (4) restoring program state to a specific statement for verification. If all four tasks could be performed with direct assistance from a debugging tool, the debugging effort would become much easier.

We have built a prototype debugging tool, *Spyder*, to assist users in conducting the first and last tasks. *Spyder* executes the first task by using *dynamic program slicing* and the fourth task by *backward execution*. This research focuses on the second task, reducing the search domain containing faults, referred to as *fault localization*.

Several heuristics are presented here based on dynamic program slices and information obtained from testing. A family tree of the heuristics is constructed to study effective application of the heuristics. The relationships among the heuristics and the potential order of using them are also explored. A preliminary study was conducted to examine the effectiveness of the heuristics proposed. Results of our study show the promise of fault localization based on these heuristics as well as suggest criteria for precise application of the heuristics (e.g., the standard of selecting thresholds). A new debugging paradigm equipped with these heuristics is expected to reduce human interaction time significantly and aid in the debugging of complex software.

Biography

Hsin Pan is currently a Ph.D. candidate in the Department of Computer Sciences at Purdue University and is expected to graduate in 1992. His thesis work focuses on developing new techniques to enhance the process of fault localization. His research interests include software quality assurance (testing and debugging), program development environments, and parallel processing.

Gene Spafford is an Assistant Professor with the Department of Computer Sciences at Purdue University and the SERC. He does research on issues relating to increasing the reliability of computer systems, and the consequences of computer failures. This includes work with software testing and debugging, computer security, and issues of liability and professional ethics. Spaf is a Senior Member of IEEE and the IEEE Computer Society, and a member of ACM and Usenix.

*This research was supported, in part, by a grant from the Software Engineering Research Center at Purdue University, a National Science Foundation Industry/University Cooperative Research Center (NSF Grant ECD–8913133), and by National Science Foundation Grant CCR–8910306.

1 Introduction

In the software life cycle, more than 50% of the total cost may be expended in the testing and debugging phases to ensure the quality of the software [30, 35]. Developing effective and efficient testing and debugging strategies is thus important.

In standards terminology [7] *errors* are defined as inappropriate actions committed by a programmer or designer. *Faults* or *bugs* are the manifestations and results of errors during the coding of a program. A program *failure* occurs when an unexpected result is obtained while executing the program on a certain input because of the existence of *errors* and *faults*.

Testing explores the input space of a program that causes the program to fail, while debugging tries to locate and fix faults or bugs after failures are detected during test or use. Although testing and debugging are closely related, none of the existing debugging tools attempt to interface with testing tools. Conventional debugging tools (e.g., ADB and DBX [15]) are command–driven and tend to be stand–alone. Many fault localization techniques used in current debugging tools (e.g., setting breakpoints) were developed in the 1960s and have changed little[5]. Users have to discover by themselves useful information for debugging.

Two major steps involved in the debugging process are locating and correcting faults. Previous studies [30, 41] found that locating faults is the most difficult and important task in debugging. The focus of our work is to develop methods that automatically localize faults and thus enhance the debugging process as well as reduce human interaction time. The result of this study will support a new debugging paradigm proposed in [34].

By surveying how experienced programmers debug software, we found that four distinct tasks are consistently performed: (1) determining statements involved in program failures (e.g., execution paths) so that the exact behavior of program failures and the influence among statements are understood, (2) selecting suspicious statements that might contain faults, (3) making hypotheses about suspicious faults (variables and locations), and (4) restoring program state to a specific statement for verification. The first task can be achieved by executing a program step–by–step using some debugging tools. However, if a debugging tool can automatically highlight the execution path of a program for a given input, the first task will be accomplished more efficiently. The second and third tasks are currently performed by manually examining the code and program failures without the assistance of debugging tools. As to the fourth task, some debugging tools (e.g., DBX and GDB [40]) support facilities allowing users to set breakpoints, to reexecute the code, and to verify the values of variables. Obviously, the debugging effort would become much easier if all four tasks could be performed with direct assistance from a debugging tool.

We have built a prototype debugging tool, *Spyder* [1, 2, 4, 6], to assist users in conducting the first and last tasks. *Spyder* performs the first task by using *Dynamic Program Slicing* [1, 3], and can automatically find the dynamic slice of a program for any given variables, locations, and test cases in terms of data and control dependency analysis. A dynamic slice is denoted as $Dyn(P, v, l, t)$, where P is the target program, v is a given variable, l is the location of v, and t is a given test case[6]. $Dyn(P, v, l, t)$ contains statements of P actually affecting the value of v at location l when P is executed against test case t. Execution paths of the program with given inputs are special cases of Dynamic Program Slicing.

In conducting the fourth debugging task, *Spyder* can restore the program state to a desired location by *backtracking* the program execution to that location and need not reexecute the program from the beginning.[1] The work described in this paper focuses on the second task — to reduce the search domain containing faults — referred to as *fault localization*. Several heuristics based on both dynamic program slices provided by *Spyder* and information obtained from testing are proposed.

The rest of this paper is organized as follows. The heuristics proposed by us are illustrated in the second section. In Section 3, a family tree of the proposed heuristics is constructed to study effective application.

[1]Readers are referred to [1, 3] and [2, 4] for more details on dynamic program slicing and backtracking, respectively.

Examples and results of preliminary studies of these heuristics are presented in Section 4. A brief survey of related work is given in Section 5. Finally, the contributions and extensions to this research are suggested.

2 Our Approach

From the history of the development of fault localization, we find that techniques in some prototype systems work only for programs with restricted structure and solve only limited problems. An efficient debugging paradigm that deals with a broader scope of faults is needed.

2.1 Background

The tasks performed in the process of debugging are to collect valuable information for locating faults. The information may come from program specification, program behavior during execution, results of program execution, test cases after testing, etc. Because deterministic decisions (the general approach that systematically analyzes complicated information to benefit debugging) do not yet exist, we adopt a heuristic approach to gather useful information for different cases. We believe that heuristics can cover varied situations and help us localize faults.

Dynamic Program Slicing can determine statements actually affecting program failures so that the search domain for faults will be reduced. Although it is not guaranteed that dynamic slices always contain the faults (e.g., missing statement or specification faults), to investigate statements actually affecting program failures is a reasonable strategy in debugging. By analyzing semantics and values of variables in suspicious statements of dynamic slices, we might discover valuable information for debugging. Therefore, we choose dynamic slices as the search domain to locate faults.

We have developed a family of heuristics to further reduce the search domain based on the dynamic slices provided by *Spyder* and on information obtained from testing. Each heuristic will suggest a set of suspicious statements whose size is usually smaller than the size of the dynamic slices. Although each of our proposed heuristics is only suitable for some specific kinds of faults, the overall debugging power from uniting these heuristics is expected to surpass that of currently used debugging tools.

In the testing phase, multiple test cases running against program P present different kinds of information. Our goal is to extract that information as much as possible for debugging. The more test cases we get, the better results we might have by investigating the information obtained from testing. Therefore, we prefer a *thorough test* — to finish the whole testing process to satisfy as many criteria of a selected testing method as possible. After a thorough test, if the existence of faults in program P is detected, then at least one test case can cause P to fail. These test cases are called *error–revealing test cases*, T_d. Likewise, the test cases on which P generates correct results are called *non–error–revealing test cases*, T_u. Analyzing the results of program failures will help identify suspicious variables (e.g., output variables) that contain unexpected values. Dynamic slices with respect to these suspicious variables and corresponding test cases are then constructed for the heuristics.

A dynamic slice $Dyn(P, v, l, t)$ contains four parameters: the target program P, a given variable v, the location of v (l), and a given test case t. We define two metrics, *inclusion frequency* and *influence frequency*. *Inclusion frequency* of a statement is the number of distinct dynamic slices containing the statement; *influence frequency* of a statement in a $Dyn(P, v, l, t)$ is the number of times the statement was visited in $Dyn(P, v, l, t)$. By varying either or both of the test case (t) and the variable (v) parameters, we can get different kinds of dynamic slices that are used to determine corresponding metrics. Then, heuristics are applied based on the dynamic slices and metrics obtained. At this moment, the effectiveness of heuristics for automatic fault localization under different program (P) or location (l) parameters is not clear to us, but is the subject of ongoing research. [34]

2.2 Heuristics

A family of heuristics for fault localization based on dynamic slices are proposed in our technical report [33]. In this paper, at the suggestion of the referees, we only discuss representative heuristics (but list many of them) without mathematical notation so as to simplify the presentation. Labels of heuristics illustrated in this section are the same as those in the technical report. Readers are referred to [33] for details.

Heuristics constructed using different test case parameters are similar to those constructed using different variable parameters. However, to vary the variable parameter we must be able to verify the value of a given variable with regard to the given location and test case. Because the error–revealing and non–error–revealing test case sets are obtained directly from a thorough test, it is preferred to first employ heuristics based on different test case parameters.

In order to explain the heuristics clearly, we first illustrate heuristics using the notations of different test case parameters (Heuristics 1 to 16). In this paper, the set of dynamic slices with respect to error–revealing test cases (T_d) is called the "failure set" and the ones with respect to non–error–revealing test cases (T_u) is called the "success set."

For clarity and simplicity, the proposed heuristics are primarily developed by assuming only one fault in a failed program. However, many heuristics are still suitable for the case of multiple faults.

Heuristic 1 Indicate statements present in all available dynamic slices. □

This heuristic covers all statements in all dynamic slices of both failure and success sets.

Heuristic 2 Indicate statements present in all dynamic slices of the success set. □

Faulty statements could be in these statements, if the fault is not triggered or not propagated to the result. These statements will be used by other heuristics.

Heuristic 3 Indicate statements in all dynamic slices with low inclusion frequency in the success set. □

A statement frequently involved with correct results has less chance to be faulty. When there exist many non–error–revealing test cases and very few error–revealing test cases, this heuristic can be employed to find faulty statements in the success set. The faulty statements, if any exist[2], would not lead P to wrong results when executing non–error–revealing test cases.

Heuristic 4 Indicate statements in the failure set but not in the success set, plus statements with low inclusion frequency in dynamic slices of the success set. □

Heuristic 5 Indicate statements appearing in every success slice, if any (i.e., statements in the intersection of all dynamic slices in the success set.) Then, study the necessity of the statements for correct results. □

Heuristic 6 Indicate all statements in dynamic slices of the failure set. □

This heuristic makes users focus on all statements in dynamic slices with respect to the error–revealing test cases — failure slices.

Heuristic 7 Indicate statements with high inclusion frequency in the failure set. □

Heuristic 8 Indicate statements appearing in every dynamic slice of the failure set, if any. □

[2]Failures may also be caused by missing statements. This issue is briefly discussed in Section 4.3.

A statement frequently involved with incorrect results has more chance to be faulty, and errors are confined to the statements executed. Heuristic 8 will indicate statements with the highest inclusion frequency of Heuristic 7, if there exists at least one statement appearing in every failure slice. When there exist many error–revealing test cases (failure slices) and very few non–error–revealing test cases (success slices), Heuristics 7 and 8 can be employed to find the faulty statements in the failure set. If P has a few faults that always cause program failures, then this heuristic could locate the faulty statements quickly, especially when P has only one faulty statement (single fault).

Heuristic 9 Indicate statements in the difference set of the failure set and the success set (i.e., statements in the failure set – statements in the success set). □

Statements involved in the execution of program failures but never tested by non–error–revealing test cases (i.e., statements only appearing in failure slices) are highly likely to contain faults. If there are many test cases in both error–revealing and non–error–revealing test set, this method is worth trying. Statements indicated by this heuristic are in the difference set between the results of Heuristic 6 and Heuristic 2 (H6 – H2). A few heuristics investigating statements in the difference between two dynamic slice sets are also available (e.g, Heuristics 10, 11, and 12).

Heuristic 10 Indicate statements appearing in every failure slice but not in any success slice. □

Heuristic 11 Indicate statements in the difference set of the success set and the failure set (i.e., statements in the success set but not in the failure set). Then, study how these statements contribute to correct results, or focus on other statements. □

Heuristic 12 Indicate statements appearing in every success slice but not in any failure slice. Then, study the necessity of these statements for correct results. □

Heuristic 13 Indicate statements in all dynamic slices with high inclusion frequency in the failure set and low inclusion frequency in the success set. □

This heuristic is a combination of Heuristics 3 and 7. It is suitable when many statements are involved in both failure and success sets, and many elements are in the error–revealing and non–error–revealing sets after a thorough test. Statements leading to incorrect results and less involved in correct program execution are highly likely to contain bugs. For such statements, the ratio of the corresponding inclusion frequency in the failure set to the corresponding inclusion frequency in the success set is a useful indicator. The higher ratio a statement has, the higher chance the statement contains faults.

Heuristic 14 If a set of statements B_1 located by the above heuristics, especially by those indicating statements with low inclusion frequency in the success set (e.g., Heuristics 3, 4, and 13), does not contain faults and belongs to a branch of a decision block such as `if` (exp) `then` $\{\ldots B_1 \ldots\}$ `else` B_2, or `while` (exp) $\{\ldots B_1 \ldots\}$, then the logical expression exp should be examined. □

Because the logical expression exp is always executed to decide whether B_1 should be executed, inclusion frequency of the predicate statement (`if`–statement or `while`–statement) is always equal to or greater than that of $B1$. The indication of the predicate statement based on inclusion frequency is thus not as effective as B_1. This heuristic reminds users to examine logical expression exp in the predicate statement.

Heuristic 15 Indicate statements with high influence frequency in a selected failure slice. □

The influence frequency measures the effect of statements being executed more than once (e.g., in a loop) but not counted in the inclusion frequency. The logic behind this heuristic is the same as that behind Heuristic 7 — statements often contributing to incorrect results are more likely to be faulty.

Heuristic 16 Indicate statements with low influence frequency in a selected success slice. □

A similar set of heuristics can be obtained by only varying the variable parameter (instead of different test case parameters) with respect to the given P, l, and t. In these cases, we must be able to verify the value of suspicious variables with respect to the given location (l) and test case (t) in order to construct the "failure" variable set (having incorrect value) and the "success" variable set (having correct value). By combing heuristics based on varying test cases and variables, we can develop another set of heuristics such as Heuristics 18, 19, and 20 in [33].

More specific criteria for applying these heuristics have yet to be studied. For example, the threshold for statements with high inclusion frequency in the failure set could be suggested as the highest twenty percent of statements in the union sorted by inclusion frequency. Users are allowed to set their own threshold for different purposes. Our preliminary study suggests a better way to decide the threshold. The suggestion is discussed in Section 4.3.

Heuristics and experiments according to relational (decision–to–decision) path analysis on execution paths were studied by Collofello and Cousins [12]. A few of their approaches are similar to ours, such as the concept behind Heuristic 13, the most useful one among theirs. As our current approach allows users to vary two out of four parameters (variable and test case) of dynamic slices that contain statements actually affecting program failures, possible faulty statements suggested by our heuristics should be more precise than those suggested by theirs and other approaches. Also, more information is provided by our heuristics.

With the aid of our heuristics, a reduced search domain for faults (e.g., a smaller set of suspicious statements) is anticipated. Other functions of dynamic instrumentation provided by *Spyder* will then help us manage further examination. For instance, the reaching definition, which shows the latest definition of a variable and is a subset of the def–use chain in program dependency analysis, enables us to trace back to find the place where a suspicious variable went wrong. Then, the backtrack function effectively "executes" a program in *reverse* until a preset breakpoint is reached, just like forward program execution being suspended at breakpoints. In short, an efficient debugging session is conducted by locating faults from a reduced domain via our heuristics as well as by using effective functions of dynamic instrumentation.

3 Further Analysis

We have constructed a family of heuristics to study effective algorithms for applying the heuristics. In this family, relationships among the heuristics and the potential order of using them were explored.

We first examined heuristics with different test case parameters (Heuristics 1 to 16) by constructing the family tree in Figure 1. The same argument based on Figure 1 can then be applied to heuristics with different variable parameters. In Figure 1, each node represents one heuristic. A solid line connects an upper (parent) and a lower (child) nodes with a superset–subset relationship that statements indicated by the parent (upper) node contain those of the child (lower) node. A dotted line links an upper node and a lower node that is derived from the upper node but without a superset–subset relationship. Bold nodes are heuristics that require thresholds, and dot (intermediate) nodes are heuristics derived from others to construct a complete family tree. Statements highlighted by heuristics within the intermediate nodes only give us basic information for debugging. By contrast, their descendant heuristics will provide more helpful information. Among the intermediate nodes, Heuristic 1 (H1) is the root of the family tree, Heuristic 2 (H2) is the root of a subtree with respect to non–error–revealing test cases, and Heuristic 6 (H6) is the root of a

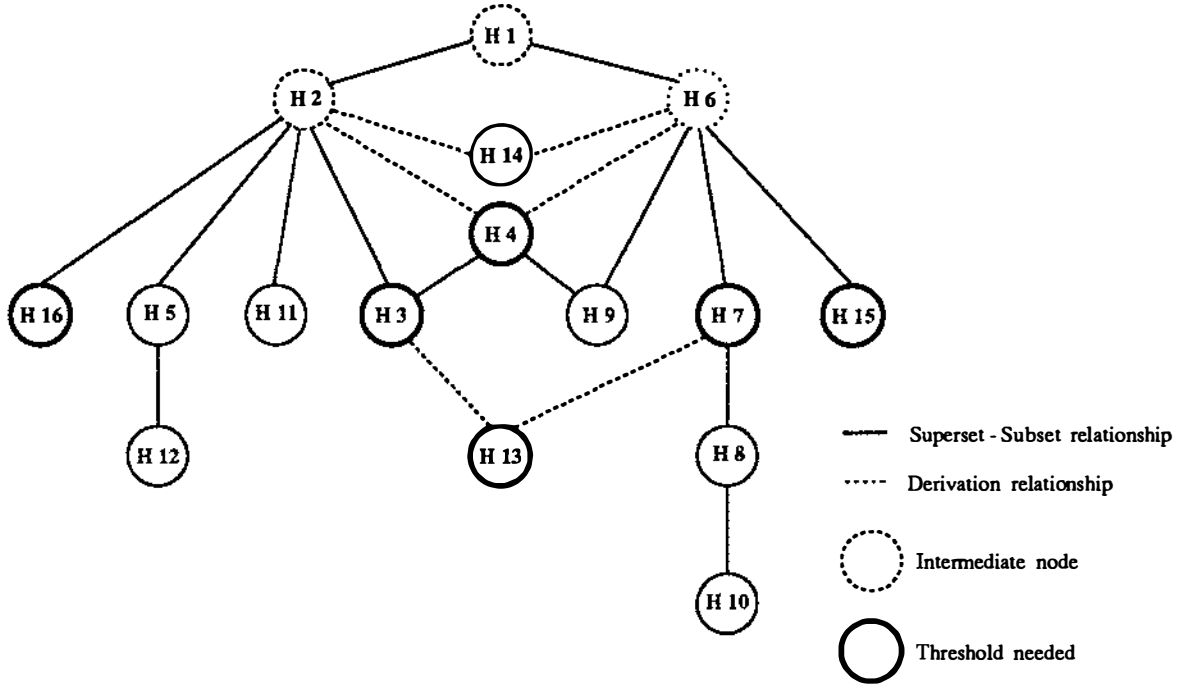


Figure 1: The family tree of the proposed Heuristics 1 to 16

subtree with respect to error–revealing test cases. H14 can be applied after traversing other heuristics in the tree except H15 and H16.

Heuristics of the subtree rooted by H2 are based on dynamic slices of the success set (with respect to non–error–revealing test cases). These heuristics, especially H5, H12, and H11, are different from others because they highlight statements that often lead to correct results and suggest studying the necessity of the statements for correct results. However, we can ignore these statements and focus on other statements involved in the family tree (i.e., statements suggested by Heuristic 1). In this case, $\overline{H5}$, $\overline{H12}$, and $\overline{H11}$ represent the corresponding compliment heuristics.

As heuristics H1 to H14 are based on all dynamic slices with respect to error–revealing and/or non–error–revealing test cases, their functions are interpreted as *global analysis*. On the other hand, H15 and H16 conduct *local analysis* because they are based on one dynamic slice at a time. We prefer to perform global analysis first. Local analysis is then conducted to reduce the search domain further to locate faulty statements.

According to the family tree in Figure 1, heuristics without threshold requirements (especially for the nonintermediate nodes) are preferred before those with threshold requirements. This is because the former suggest a precise set of statements and the latter indicate a different set of statements for different thresholds. For a pair of nodes with a superset–subset relationship, the parent heuristic can be evaluated first because the child heuristic conducts analysis based on the result of the parent heuristic analysis (e.g., H5/H12 and H8/H10).

From the above discussion, global analysis can be summarized as:

Group 1: applying heuristics in nonintermediate nodes — H8/H10, H9, H5/H12, H11, $\overline{H5}/\overline{H12}$, and $\overline{H11}$.

Group 2: applying heuristics with threshold requirements — H4, H3, H7, and H13. Search domains provided by heuristics in this group contain less statements than those provided by heuristics in

Group 1. The top–down order of employing these heuristics are preferred because the results of upper nodes will be used by lower nodes. For each heuristic, we should try the strictest threshold first. If faulty statements are not in the suggested region, the threshold will be increased gradually and the search domain will be expanded accordingly.

Group 3: applying H14 to recheck some suspicious statements that are ignored by the above heuristics.

While traversing the family tree, we interpret the top–down steps as refining suspicious statements and the bottom–up steps as extending the search domain. Users can make a guess first to get a set of suspicious statements for examination. Then, our tool will help them refine or extend the search domain by traversing the family tree of heuristics.

The result of global analysis is a set of suspicious statements (a reduced search domain) based on dynamic slices. To further verify faulty statements, heuristics performing local analysis (e.g., H15 and H16) are employed based on one dynamic slice at a time. Statements indicated by both global and local analyses are first examined, then the other highlighted statements.

Techniques and steps discussed above can be applied to heuristics with different variable parameters as well as to heuristics with different test case and variable paramenters.

After a reduced search domain for faults is presented by our proposed heuristics, further analysis is needed to identify whether faulty statements are in the highlighted suspicious region. Ongoing research will provide automated decision support to do verification. [42]

4 Results of A Preliminary Study

A simple trial was conducted to examine the effectiveness of the heuristics proposed. Because the major goal of fault localization is to reduce the search domain containing faults, the effectiveness was analyzed by comparing the known faulty statements and the suspicious domains suggested by heuristics for each tested program. In this section, we discuss the selection of tested programs, the results obtained from applying the heuristics presented in Section 2.

4.1 Tested Programs

Eleven test programs were selected and constructed based on seven programs. Most of these programs were collected from previous studies and are well–known experimental programs with previously studied faults.[3]

Table 1 gives the size and complexity of each tested program. The first column lists the number of executable statements, showing the size of a tested program. Columns 3 to 5 are obtained from a data flow coverage testing tool — *Atac* (Automatic Test Analysis for C programs) [22], developed at Bellcore. The following definitions are quoted from the man pages of *Atac*.

> Column *blocks* represents the number of code fragments not containing control flow branching. Column *decisions* shows the number of pairs of blocks for which the first block ends at a control flow branch and the second block is a target of one of these branches. Column *p–uses* (predicate uses) indicates the number of triples of blocks for which the first block contains an assignment to a variable, the second block ends at a control flow branch based on a predicate containing that variable, and the third block is a target of one of these branches. Column *all–uses* is the sum of *p–uses* and pairs of blocks for which the first block contains an assignment to a variable and the second block contains a use of that variable that is not contained in a predicate.

[3]The programs are described in Appendix A.

Program	# of executable statements	blocks	decisions	p-uses	fault types
aveg	35	36	18	40	wrong logical expression
calend	29	22	10	16	wrong logical operator
find1	33	32	18	80	wrong variable reference
find2	33	32	18	80	wrong variable reference
find3	32	32	18	80	missing a statement & faults of find1 and find2
gcd	57	57	36	124	wrong initialization (value)
naur1	37	28	18	48	missing simple logical expression
naur2	37	28	18	50	missing simple logical expression
naur3	36	28	18	46	missing predicate statement
transp	155	156	73	135	wrong initialization (value)
trityp	37	47	39	99	wrong logical operator

Table 1: Tested programs

The data flow coverage criteria (columns 3 to 5) help us understand the complexity of a tested program. Fault types of each tested program are presented in Column 6. Most of the tested programs have only one fault so that we can easily examine the effectiveness of our proposed heuristics for fault localization.

4.2 Results

As mentioned in Section 2.1, a thorough test is preferred before applying our proposed heuristics. *Atac* was used to conduct the thorough test and to construct two test case sets, non–error–revealing test case set T_u and error–revealing test case set T_d. A set of data–flow criteria of a selected program is provided after the tested program is analyzed by *Atac* (e.g., blocks, decisions, p–uses, and all–uses). Each test case satisfies the criteria to a certain degree when executed against the tested program. Summary of the satisfaction degree presented in Table 2 consists of both percentage and counts of all four criteria to show the adequacy of selected test cases. *Atac* was employed to satisfy the coverage of criteria as much as possible and to guarantee adequacy. In our experiment, test cases in T_u were added to improve the satisfaction degree without causing program failure. Test cases in T_d were added to improve the satisfaction degree under program failure. Information presented in Table 2 contains the highest percentage that were reached by the selected test cases.

Our heuristics were then applied based on T_u, T_d, and output variables of tested programs using *Spyder*. A set of suspicious statements was suggested by each heuristic for a selected program. To measure the effectiveness of a proposed heuristic, we first examined whether the known faulty statements were contained by the suggested statements. Then, we compared the reduced domain with the size of the original tested program (i.e., the number of executable statements) and with the domain suggested by the root of the heuristic family in Figure 1 (i.e., H1). These results are presented in Tables 3 and 4. Comparing the reduction in lines of code from the original program (LoC) to the number highlighted by each heuristic shows the degree of effectiveness for the reduced search domain. The star superscript in an entry indicates the suggested statements containing the known faulty statements. Thus, we would probably decide to first apply those heuristics showing a significant reduction in the lines of code to be examined by the user.

As discussed in Section 3, heuristics without threshold requirements should be applied before heuristics with the requirements. Results of applying heuristics without threshold requirements are presented in Table 3. Heuristics 1 and 6 are considered separately because they are intermediate nodes in Figure 1, and

Prog.	t.c. types	% blocks	% decisions	% p-uses	% all-uses
aveg	T_u	100 (36)	100 (18)	73 (29/40)	81 (64/79)
	T_d	100 (36)	94 (17/18)	70 (28/40)	77 (61/79)
calend	T_u	100 (22)	90 (9/10)	94 (15/16)	97 (30/31)
	T_d	91 (20/22)	60 (6/10)	69 (11/16)	74 (23/31)
find1	T_u	100 (32)	100 (18)	79 (63/80)	84 (104/124)
	T_d	97 (31/32)	94 (17/18)	76 (61/80)	82 (102/124)
find2	T_u	100 (32)	100 (18)	79 (63/80)	84 (104/124)
	T_d	97 (31/32)	94 (17/18)	74 (59/80)	81 (100/124)
find3	T_u	100 (32)	100 (18)	80 (64/80)	85 (104/122)
	T_d	97 (31/32)	94 (17/18)	75 (60/80)	80 (98/122)
gcd	T_u	100 (57)	89 (32/36)	69 (85/124)	71 (163/230)
	T_d	95 (54/57)	81 (29/36)	64 (79/124)	67 (153/230)
naur1	T_u	100 (28)	100 (18)	65 (31/48)	66 (53/80)
	T_d	96 (27/28)	89 (16/18)	56 (27/48)	61 (49/80)
naur2	T_u	100 (28)	100 (18)	66 (33/50)	67 (55/82)
	T_d	100 (28)	100 (18)	62 (31/50)	65 (53/82)
naur3	T_u	100 (28)	100 (18)	70 (32/46)	69 (54/78)
	T_d	100 (28)	100 (18)	78 (36/46)	79 (62/78)
transp	T_u	94 (146/156)	89 (65/73)	81 (110/135)	85 (307/361)
	T_d	96 (150/156)	90 (66/73)	79 (107/135)	84 (304/361)
trityp	T_u	98 (46/47)	97 (38/39)	76 (75/99)	78 (88/113)
	T_d	66 (31/47)	51 (20/39)	37 (37/99)	39 (44/113)

Table 2: *Atac*'s measurement of test case adequacy

search domains highlighted by H6 are not significantly reduced from the domain provided by H1. We are more interested in other heuristics in Table 3. For those entries with a zero, the related heuristics do not indicate any suspicious statements for fault localization and can be ignored.

Heuristics with threshold requirements are presented in Table 4. Statements involved in a heuristic with threshold requirements are first ranked according to the metric used by the heuristic (e.g., inclusion frequency) and are then grouped based on the ranks (i.e., statements with the same rank are in one group). Among different groups, the rank associated with a group that contains the fault is referred to as the *critical level*. The threshold of the heuristic should at least be set at the critical level to assure the suggested domain containing faults. This minimal threshold is referred to as the *critical threshold*. After the critical level is decided, suspicious statements are thus highlighted by the heuristic.

In Table 4, Row b shows the critical thresholds: the ratio of the rank of the critical level to the number of ranked levels. If faulty statements do not belong to any statements involved in the heuristic, the critical threshold will be 100% (e.g., the two values are the same).

We use the entry of H3 (indicating statements with low inclusion frequency in the success set) on program calend as an example for further illustration. Five groups with different inclusion frequency are obtained from the success set on calend: the first group (ranked 1) has the lowest inclusion frequency 1 and contains eight statements; the second group (ranked 2) contains nine statements with inclusion frequency 2; the third group (ranked 3) contains one statement with inclusion frequency 3; the fourth group (ranked 4) contains one

Prog.	LoC	H1	H6	H8	H9
aveg	35	28*	28*	23*	0
calend	29	23*	22*	7*	0
find1	33	25*	25*	24*	0
find2	33	25*	25*	22*	0
find3	32	24*	24*	21*	0
gcd	57	39*	36*	11*	1*
naur1	37	31*	31*	25*	0
naur2	37	31*	31*	25*	0
naur3	36	30	30	27	0
transp	155	45*	31*	29*	1*
trityp	37	21*	15*	10*	0

*: the highlighted statements contain faulty statements

Table 3: Effectiveness analysis for heuristics without threshold requirements

statement with inclusion frequency 4; and the fifth group (ranked 5) contains four statement with inclusion frequency 5. The faulty statement (Statement 30) is in the second group. Thus, the critical level of H3 on calend is 2 (i.e., associated with the second group), and those seventeen statements (including the faulty one) in the first (having eight statements) and second (having nine statements) groups are highlighted. We then decide the critical threshold in Row b based on the ranks (the ratio of the rank of the critical level to the total number of ranked levels — 2/5).

Heuristic 14 is designed to enhance the location of faults in the predicate expression when other heuristics are not effective enough. Because H14 would consider results of other heuristics, a precise critical threshold is hard to define for it. In Table 4, Row a for H14 indicates the number of predicate statements in a tested program. Row b tells the effectiveness of using H14 to locate faulty predicate statements. "N/A" means that this heuristic is not applicable because faults of the corresponding tested program are not in predicate statements. "*" means that the faulty predicate statements can be located by using this heuristic combined with other heuristics. If this heuristic does not provide more effective information than other heuristics do for locating faulty predicate statements, "–" is marked in the entry.

4.3 Discussion

The determination of thresholds for heuristics with this requirement (e.g., H3, H4, H7, and H13) will affect the effectiveness and size of suggested domains. A unique threshold, which makes the suggested domain reasonably small and consistently contain faults, is highly desirable. In Table 4, general critical thresholds for various heuristics in Row a range from 1% to 91%. A standard threshold cannot be decided within this wide scope. On the other hand, we find that most critical thresholds in Row b are below 50%. Moreover, thresholds based on ranked levels are easy to use (e.g., from the first to the last ranked level, gradually). Therefore, we conclude to choose thresholds based on ranked levels and using the first 50% of ranked levels as a standard threshold for the first–time criterion when employing these heuristics.

By comparing the size and percentage of search domains suggested by heuristics for each tested program, we found that for most cases the domains provided by heuristics with threshold requirement are more precise than those provided by heuristics without this requirement. Therefore, we prefer to examine the search domains suggested by heuristics with threshold requirements first (i.e., H4, H3, H7, and H13), although extra effort is needed to decide the critical thresholds. From Tables 3 and 4 it is hard to conclude which heuristic is the most effective one. Also it is not appropriate to make this conclusion because these heuristics are

Prog.		LoC	H3	H13	H14
aveg	a	35	28*	25*	9*
	b		3/3*	2/3*	–
calend	a	29	17*	7*	3*
	b		2/5*	2/7*	*
find1	a	33	10*	9*	9*
	b		2/3*	2/4*	–
find2	a	33	4*	4*	9*
	b		2/4*	1/4*	–
find3	a	32	4*	4*	9*
	b		2/4*	2/5*	–
gcd	a	57	38	1*	17
	b		5/5	1/9*	N/A
naur1	a	37	28*	23*	6*
	b		3/4*	3/5*	*
naur2	a	37	28*	22*	6*
	b		3/4*	1/5*	*
naur3	a	36	3	3	5
	b		2/4	2/4	*
transp	a	155	45	1*	36
	b		3/3	1/5*	N/A
trityp	a	37	10*	4*	11*
	b		3/8*	2/8*	*

*: the highlighted statements contain faulty statements
a: # of highlighted statements
b: rank of the critical level / # of ranked levels
– : not effective enough

Table 4: Effectiveness analysis for heuristics with critical threshold requirements

proposed to handle different situations. Program behavior and type of faults would affect the effectiveness of these heuristics. We can only provide a general approach to apply these heuristics.

For some types of faults (e.g., missing assignment), our heuristics cannot directly cover the faults in suggested domains. However, analyzing the statements highlighted by our heuristics (e.g., semantics of suspicious variables in the statements, the analysis approaches mentioned in [6]) can lead to the identification of the faults. For instance, the missing predicate statement in program naur3 can be located by analyzing the semantics of statements suggested by Heuristics in Table 4. If the faults of wrong initialization in program gcd and transp are changed to missing initialization statements, we still can indirectly locate the faults of missing initialization statements by analyzing the semantic of variables in suspicious statements highlighted by our heuristics. Other approaches to fault localization based on test knowledge (e.g., error–revealing mutations derived from mutation–based testing [34]) and dynamic program slices are currently under development by the authors. These approaches further will help fault localization.

5 Related Work

In this section, a brief survey of typical fault localization techniques is presented.

Traditional debugging techniques such as dumping memory, scattering print statements, setting breakpoints by users, and tracing program execution only provide utilities to examine a *snapshot* of program execution. Users have to use their own strategies to do fault localization.

Shapiro [39] proposed an interactive fault diagnosis algorithm, the *Divide–and–Query* algorithm, for debugging programs represented well by a computation tree (e.g., the logic programs written in Prolog). The computation tree (the target program) is recursively searched until bugs are located and fixed. Renner [37] applied this approach to locating faults in programs written in Pascal. With this method, users can only point out procedures that contain bugs; other debugging tools are needed to debug the faulty procedures. The similar result is obtained in [17].

The knowledge–based approach attempts to automate the debugging process by using techniques of artificial intelligence and knowledge engineering. Knowledge about both the classified faults and the nature of program behavior is usually required in this approach. Many prototype debugging systems have been developed based on this approach since the early 1980s. [14, 38] However, knowledge about programs in the real world is complicated. These prototype systems can only handle restricted fault classes and very simple programs.

Program slicing proposed by Weiser [43, 44] is another approach to debugging. This method decomposes a program by statically analyzing the data–flow and control–flow of the program — referred to as *static program slicing*. Program dicing, proposed by Lyle and Weiser [27, 45], attempts to collect debugging information according to the correctness of suspicious variables involved in static program slices. *Focus* [28] is a debugging tool based on program dicing to find the likely location of a fault. Because static program slices contain many irrelevant statements that make fault localization inefficient, studying program slicing based on dynamic cases to get the exact execution path is warranted. Dynamic Program Slicing [1, 3, 6, 25] is a powerful facility for debugging and dependency analysis. Nevertheless, it has not been systematically applied to fault localization. In Agrawal's dissertation [6], he briefly alluded to the idea of combining dynamic program slices and data slices for fault localization. Our heuristics are based on dynamic slices that are collected by varying test cases, variables, and location of variables.

Current testing and debugging tools are separate. Even if they are presented in one tool, the two functions are not well integrated to benefit each other. Osterweil [32] tried to integrate testing, analysis, and debugging, but gave no solid conclusion about how to transform information between testing and debugging to benefit each other. Clark and Richardson [11] were the first to suggest that certain test strategies (based on the symbolic evaluation) and classified failure types could be used for debugging purposes. However, only one example is given to describe their idea, and no further research has been conducted.

STAD (System for Testing and Debugging) [24] is the first tool to successfully integrate debugging with testing. As mentioned above, its testing and debugging parts do not share much information except for implementation purposes (e.g., they share the results of data flow analysis). The debugging part of STAD will be invoked once a fault is detected during a testing session, and leads users to focus on the possible erroneous part of the program rather than locate the fault precisely. PELAS (Program Error–Locating Assistant System) [23] is an implementation of the debugging tool in STAD. Korel and Laski proposed an algorithm based on hypothesis–and–test cycles and knowledge obtained from STAD to localize faults interactively [26]. However, STAD and PELAS only supported a subset of Pascal, and limited program errors are considered.

Collofello and Cousins [12] proposed many heuristics to locate suspicious statement blocks after testing. A program is first partitioned into many decision–to–decision paths (DD–paths), which are straight–line codes existing between two consecutive predicates of the program. Two test data sets are obtained after testing: one detects the existence of faults and the other does not. Then, heuristics are employed to predict

possible DD–paths containing bugs based on the number of times that DD-paths are involved in those two test data sets. Their ideas are related to some of the heuristic fault localization strategies proposed by us. The deficiency of their method is that only execution paths (DD-paths), a special case of dynamic program slicing, are examined. After reducing the search domain to a few statement blocks (DD-paths), no further suggestion is provided for locating bugs.

Unlike the approaches proposed by Lyle–Weiser and Collofello–Cousins, which are only based on suspicious variables and test cases respectively, our heuristics are developed by considering test cases, variables, and location of variables together. We believe our methods will obtain more helpful information and reduce the search domain effectively.

6 Concluding Remarks

With the support of *Spyder* and the proposed heuristics, a new debugging scenario can be described as follows: (1) users find program failures by using the testing methodology provided by an integrated testing tool; (2) the debugging tool interactively helps users reduce the search domain for faults by dynamic instrumentation (e.g., dynamic program slicing and backtracking) and the information obtained from the testing phase; (3) the tool supports fault prediction strategies based on the reduced domain and test–based information; and (4) users can retest the program to assure that the program failure has been prevented after faults are located and fixed. The tool to support this new debugging scenario should be integrated with a testing environment and can conduct program dependency analysis, monitor execution history for backtracking, and provide fault prediction strategies based on information obtained from failure analysis and fault classification. [34]

In this paper, a set of heuristics is proposed to confine the search domain for bugs to a small region. The heuristics are based on dynamic program slices that are collected by varying test cases, variables, and location of variables. Preliminary results of our studies indicate the effectiveness and feasibility of the proposed heuristics. Although it is not guaranteed that faults can be found in the domains suggested by the proposed heuristics, a confined small region containing faults or information leading to fault discovery is provided for further analysis.

We continue to study the nature of program failures/faults as well as information obtained from testing methodology to further develop the foundation of our heuristics and to develop other promising approaches. We expect that our new debugging paradigm equipped with these heuristics will significantly reduce human interaction time and aid in the debugging of complex software.

Acknowledgements

The authors would like to thank Richard DeMillo for suggesting the idea of constructing the family tree for proposed heuristics in Section 3, Hiralal Agrawal for building the prototype debugging tool – *Spyder*, and Shi-Miin Liu and R. J. Martin for reviewing early drafts of this article. We also thank Bellcore for making *Atac* available for research use at Purdue. The 10th Pacific Northwest Software Quality Conference reviewers provided valuable comments and suggestions.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the ACM SIGSOFT '91 Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pages 60–73, Victoria, British Columbia, Canada, October 8–10 1991.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution backtracking approach to program debugging. *IEEE Software*, 8(3):21–26, May 1991.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, White Plains, New York, June 1990. (*ACM SIGPLAN Notices*, 25(6), June 1990).

[4] H. Agrawal and E. H. Spafford. An execution backtracking approach to program debugging. In *Proceedings of the 6th Pacific Northwest Software Quality Conference*, pages 283–299, Portland, Oregon, September 19–20 1988.

[5] H. Agrawal and E. H. Spafford. A bibliography on debugging and backtracking. *ACM Software Engineering Notes*, 14(2):49–56, April 1989.

[6] Hiralal Agrawal. *Towards Automatic Debugging of Computer Programs.* PhD thesis, Purdue University, West Lafayette, Indiana, August 1991. (Also released as Technical Report SERC-TR-103-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, September 1991).

[7] ANSI/IEEE. *IEEE Standard Glossary of Software Enginnering Terminology.* IEEE Std 729–1983. IEEE, New York, 1983.

[8] R. S. Boyer, E. Elspas, and K. N. Levitt. SELECT — a system for testing and debugging programs by symbolic execution. In *Proceedings of International Conference on Reliable Software*, pages 234–245, 1975. (*ACM SIGPLAN Notices*, 10(6), June 1990).

[9] Gordon H. Bradley. Algorithm and bound for the greatest common divisor of n integers. *Communications of the ACM*, 13(7):433–436, July 1970.

[10] T. A. Budd. *Mutation Analysis of Program Test Data.* PhD thesis, Yale University, New Haven, Connecticut, 1980.

[11] Lori A. Clarke and Debra J. Richardson. The application of error–sensitive testing strategies to debugging. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 45–52, Pacific Grove, California, March 1983. (*ACM Software Engineering Notes*, 8(4), August 1983; *ACM SIGPLAN Notices*, 18(8), August 1983).

[12] James S. Collofello and Larry Cousins. Towards automatic software fault location through decision–to–decision path analysis. In *AFIPS Proceedings of 1987 National Computer Conference*, pages 539–544, Chicago, Illinois, June 1987.

[13] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–43, April 1978.

[14] Mireille Ducasse and Anna-Maria Emde. A review of automated debugging systems: Knowledge, strategies, and techniques. In *Proceedings of the 10th International Conference on Software Engineering*, pages 162–171, Singapore, April 1988.

[15] Kevin J. Dunlap. Debugging with DBX. In *UNIX Programmers Manual, Supplementary Documents 1, 4.3 Berkeley Software Distribution.* Computer Science Division, University of California, Berkeley, California, April 1986.

[16] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of the all–uses and all–edges adequacy criteria. In *Proceedings of the ACM SIGSOFT '91 Fourth Symposium on Testing, Analysis and Verification (TAV4)*, pages 154–164, Victoria, British Columbia, Canada, October 8–10 1991.

[17] Peter Fritzson, Tibor Gyimothy, Mariam Kamkar, and Nahid Shahmehri. Generalized algorithmic debugging and testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 317–326, Toronto, Canada, June 26–28 1991.

[18] M. Geller. Test data as an aid in proving program correctness. *Communications of the ACM*, 21(5):368–375, May 1978.

[19] J. B. Goodenough and S. L. Gerhart. Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[20] F. Gustavson. Remark on algorithm 408. *ACM Transactions on Mathematical Software*, 4:295, 1978.

[21] C. Hoare. Algorithm 65: FIND. *Communications of the ACM*, 4(1):321, April 1961.

[22] J. R. Horgan and S. London. Data flow coverage and the C language. In *Proceedings of the ACM SIGSOFT '91 Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pages 87–97, Victoria, British Columbia, Canada, October 8–10 1991.

[23] Bogdan Korel. PELAS – program error-locating assistant system. *IEEE Transactions on Software Engineering*, SE-14(9):1253–1260, September 1988.

[24] Bogdan Korel and Janusz Laski. STAD – a system for testing and debugging: User perspective. In *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification*, pages 13–20, Banff, Canada, July 1988.

[25] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *The Journal of Systems and Software*, 13(3):187–195, November 1990.

[26] Bogdan Korel and Janusz Laski. Algorithmic software fault localization. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, pages 246–252, Hawaii, January 1991.

[27] James R. Lyle. *Evaluating Variations on Program Slicing for Debugging.* PhD thesis, University of Maryland, College Park, Maryland, December 1984.

[28] James R. Lyle and Mark Weiser. Automatic program bug location by program slicing. In *Proceedings of the 2nd International Conference on Computers and Applications*, pages 877–883, Beijing, PRC, June 1987.

[29] J. M. McNamee. Algorithm 408: A sparse matrix package (part I) [f4]. *Communications of the ACM*, 14(4):265–273, April 1971.

[30] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, Inc., 1979.

[31] P. Naur. Programming by action clusters. *BIT*, 9:250–258, 1969.

[32] Leon Osterweil. Integrating the testing, analysis, and debugging of programs. In H. L. Hausen, editor, *Software Validation*, pages 73–102. Elsevier Science Publishers B. V., North–Holland, 1984.

[33] H. Pan and E. H. Spafford. Heuristics for automatic localization of software faults. Technical Report SERC-TR-116-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, July 1992.

[34] Hsin Pan. Debugging with dynamic instrumentation and test–based knowledge. Technical Report SERC-TR-105-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, September 1991.

[35] Roger S. Pressman. *Software Enginnering: A Practitioner's Approach.* McGraw–Hill, Inc., second edition, 1987.

[36] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, December 1976.

[37] Scott Renner. Location of logical errors on Pascal programs with an appendix on implementation problems in Waterloo PROLOG/C. Technical Report UIUCDCS-F-82-896, Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, Illinois, April 1982. (Also with No. UIUC-ENG 82 1710.).

[38] Rudolph E. Seviora. Knowledge–based program debugging systems. *IEEE Software*, 4(3):20–32, May 1987.

[39] Ehud Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, Cambridge, Massachusetts, 1983. (PhD thesis, Yale University, New Haven, Connecticut, 1982).

[40] Richard M. Stallman. *GDB Manual, third edition, GDB version 3.4.* Free Software Foundation, Cambridge, Massachusetts, October 1989.

[41] Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man–Machine Studies*, 23(5):459–494, November 1985.

[42] Chonchanok Viravan. Fault investigation and trial. Technical Report SERC-TR-104-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, September 1991.

[43] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[44] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[45] Mark Weiser and Jim Lyle. Experiments on slicing–based debugging aids. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers*, pages 187–197. Ablex Publishing Corp., Norwood, New Jersey, 1986. (Presented at *the First Workshop on Empirical Studies of Programmers*, Washington DC, June 5–6 1986.).

Appendix A: Tested Programs

Source code of all tested programs (written in the C programming language) may be obtained from the authors.

Program **aveg** first calculates the mean of a set of input integers. Then, percentages of the inputs above, below, and equal to the mean (i.e., the number of inputs above, below, and equal to the mean divided by the total number of inputs) are reported. Our version is directly translated from a Pascal version, and a fault was accidentally introduced during the transformation. The fault is an incorrect logical expression in an `if`–statement. If the working variable for the mean of the given integers has the value zero during calculation, then the result is incorrect.

Geller's calendar program **calend** [18], which was analyzed by Budd [10], tries to calculate the number of days between two given days in the same year. A wrong logical operator (== instead of !=) is placed in a compound logical expression of an `if`–statement. This fault causes errors in leap years.

The **find** program of Hoare [21] deals with an input integer array a with size $n \geq 1$ and an input array index f, $1 \leq f \leq n$. After its execution, all elements to the left of $a[f]$ are less than or equal to $a[f]$, and all elements to the right of $a[f]$ are greater than or equal to $a[f]$. The faulty version of **find**, called **buggyfind**, has been extensively analyzed by SELECT [8], DeMillo–Liption–Sayward [13], and Frankl–Weiss [16]. In our experiment, **find3** is the C version of **buggyfind**, which includes one missing statement fault and two wrong variable references (in logical expressions). The two wrong variable references were placed in **find1** and **find2** respectively.

Bradley's **gcd** program [9], which was also analyzed by Budd [10], calculates the greatest common divisor for elements in an input integer array a. In our experiment, a missing initialization fault of **gcd** was changed to a wrong initialization with an erroneous constant.

Gerhart and Goodenough [19] analyzed an erroneous text formatting program (originally due to Naur [31]). Minor modification of this program was made for our experiment. The specification of the program is as follows:

> Given a text consisting of words separated by BLANKs or by NL (New Line) characters, convert it to a line–by–line form in accordance with the following rules: (1) line breaks must be made only when the given text has a BLANK or NL; (2) each line is filled as far as possible, as long as (3) no lines contain more than MAXPOS characters.

Program **naur1** has a missing path fault (e.g., a simple logical expression in a compound logical expression is missing). With this fault, a blank will appear before the first word on the first line except when the first word has the exact length of MAXPOS characters. The form of the first line is, thus, incorrect as judged by rule (2). Program **naur2** also has a missing path fault (e.g., a simple logical expression in a compound logical expression is missing). This fault causes the last word of an input text to be ignored unless the last word is followed by a BLANK or NL. Program **naur3** contains a missing predicate statement fault (e.g., an `if`–statement is missing). In this case, no provision is made to process successive line breaks (e.g., two BLANKs, three NLs).

Program **transp** [29], which was adopted for experiment by Frankl and Weiss [16], generates the transpose of a sparse matrix whose density does not exceed 66%. Two faults were identified in the original FORTRAN program. [20] We translated the correct version to C and reintroduced one of the faults. The other fault happens because of features of the FORTRAN language and cannot be reproduced in C. The fault present is a wrong initialization with an erroneous constant.

The last tested program, **trityp**, is a well–known experimental program.[36] It takes three input integers as the length of three sides of a triangle, and decides the type of the triangle (scalene, isosceles, equilateral, or illegal). The program contains three faulty statements with the same fault type, wrong logical operator ($\geq$ instead of $>$).

Software Metrics in a Process Framework

Shari Lawrence Pfleeger
Pacific Northwest Quality Conference
Portland, Oregon
October 1992

MITRE

Overview

- Need for context for metrics
- Process maturity framework
- Combining goal/question/metric with process maturity
- Example: Derivation from Maturity Questionnaire
- Example: MITRE Metrics Advisor tool

MITRE

What a Mess!

How do I keep a project from

- **being late?**
- **costing too much?**
- **producing poor quality products?**
- **producing the wrong product?**

How do I find out as soon as possible that I have problems?

MITRE

Software Engineering: A Means to an End

Software engineering offers:

- **An organized structured way of understanding and controlling software development and maintenance**
- **A way to define a development or maintenance process and use it to make problems more visible -- especially early in the process**
- **A way to instrument a process, so that probes and indicators can evaluate the status of products and activities, measuring their quality and effectiveness**

MITRE

Steps to Understanding and Control

1. Set goals for inputs, resources, products and process.
2. Define a process, and check it for
   - realism
   - consistency
   - completeness
3. Determine what questions are implied by the goals: what does each goal mean in terms of what you have to know about process and product (and when)?
4. Measure what is visible in the process.
5. Evaluate process and measurement for sufficient answers to questions. If insufficient, return to step 2 to improve the process.

MITRE

Need for Context for Metrics

- Metrics are useful only if they address a GOAL and answer an important QUESTION
- Goals can be related to:

  PROJECT: Will we meet our schedule?

  PROCESS: Are the design reviews effective?

  PRODUCT: Is the software reliable?

- The goals and questions suggest a CONTEXT for the metrics.

MITRE

Example of Goal/Question/Metric for CIM (From Strassmann Briefing)

Information Technology Mission:

80+% reusable code and 100% reusable data

This goal translates into several questions:

1. **How do we define reusability?**
2. **How do we measure reusability?**
3. **How do we design reusability in?**
4. **What are the tradeoffs between**
   - **reusability and performance?**
   - **reusability and cost?**
   - **reusability and maintainability?**

... these questions tell us what we need to *measure* and *control* !

MITRE

Process Maturity Framework

For many organizations, a major goal is PROCESS IMPROVEMENT

Process Maturity can provide a framework for process improvement and so for metrics.

MITRE

Metrics in a Process Maturity Framework

Level	Characteristics	Measurement
5. Optimizing	Improvement fed back to process	Process + feed-back for changing process
4. Managed	Measured process (quantitative)	Process + feedback for control
3. Defined	Process defined, institutionalized	Product
2. Repeatable	Process dependent on individuals	Project
1. Initial	Ad hoc/chaotic	Preliminary project (baseline)

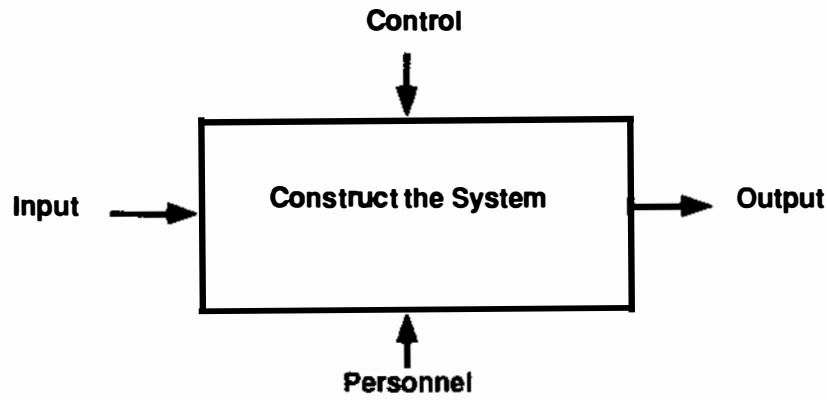
MITRE

Level 1: Initial or Ad Hoc Process

. . . Baseline Metrics

MITRE

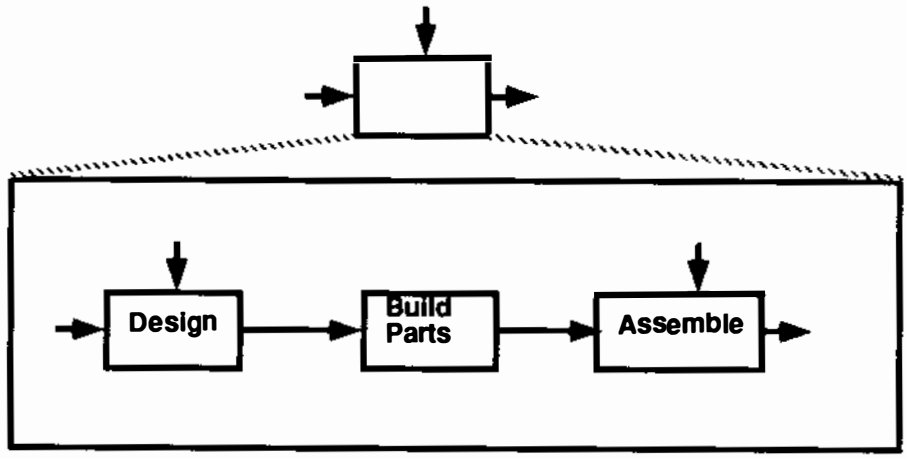
Level 2: Repeatable Process



. . . Project Management Metrics

MITRE

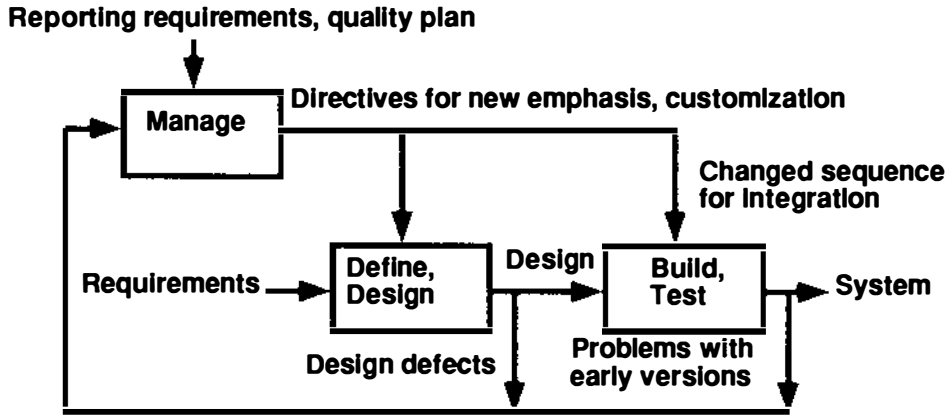
Level 3: Defined Process



. . . Product Metrics

MITRE

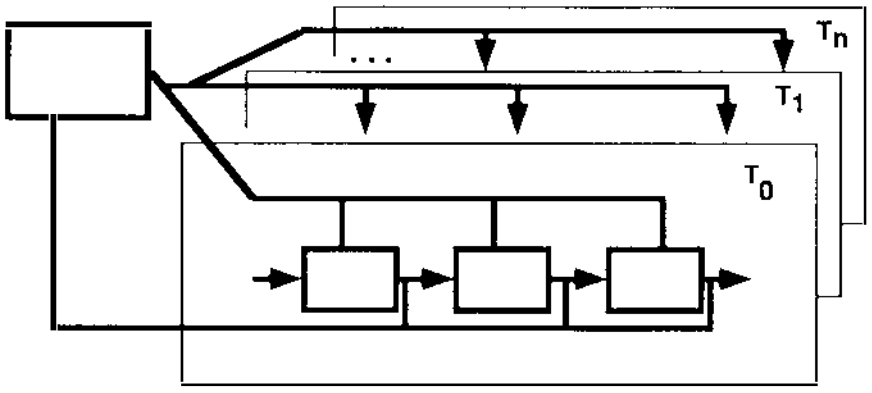
Level 4: Managed Process



. . . Process Metrics for Control

MITRE

Level 5: Optimizing



. . . Process Metrics with Feedback for Change

MITRE

What Measures Can Provide

- Early warning:
  - Are the requirements changing alot?
  - Are problems being found more quickly than they are solved?
  - Are we reusing code as much as we expected?
  - Is the design too complex?
- Evaluation of activities:
  - How many errors are being caught in requirements review? design review? code review? test case review?
  - Is testing effective?
  - Are standards being complied with?
- Prediction:
  - How do qualities of early products suggest those of later ones?
- Historical perspective:
  - Is the current situation abnormal?

Example: Deriving Metrics from Maturity Model Questions

- In Process Maturity Model's questionnaire:

What do I need to measure to answer this question "yes"?

- In Capability Maturity Model key process area:

What do I need to measure to be able to do/understand this activity?

Example Derivation

From Process Maturity Questionnaire:

Question 2.4.13: Is a mechanism used for controlling changes to software design? (level 3)

What must I measure to answer this question "yes"?

- **number of design units**
- **number of changes to each design unit (with date of each change)**
- **requirement trace for each change, responsible party, requirements verification for each change**

MITRE

Example from Key Process Areas

Key Process Area: Requirements Management

Implies tracking:

- **number of requirements**
- **changes to requirements**
- **requirements traceability**

MITRE

Migration Path for Improvement

Level 1: Baseline Metrics -- size, effort, trouble reports (DoD core)

Level 2:
- cost and schedule
- units designed/tested/integrated (progress metrics)
- number of requirements, requirements changes
- errors discovered in requirements/design/code/test
- CPU and I/O channel utilization

Level 3:
- quality of requirements/design/code/test (e.g. complexity, fan-in/out)
- traceability
- review progress and quality (errors, action items)
- number of standards and extent of implementation

MITRE

Migration Path (continued)

Level 4:
- number of errors discovered, where and when
- number of units reviewed, regression-tested
- analysis of trends over time, comparison of estimates with actuals
- number of staff-months per process step
- size of product per staff-month
- analysis of source of errors

Level 5:
- root cause analysis
- comparison across projects
- analysis of requirements/design/code volatility

MITRE

Example: MITRE Metrics Advisor

- GOALS:
  - Improve productivity
  - Improve quality
  - Reduce risk
  - Satisfy customer
- MAP GOALS TO RESOURCES/PRODUCTS/PROCESS:
  Improve quality -- product

Quality requirements
Quality design
Quality code
Quality tests
Quality documentation

MITRE

Example: MITRE Metrics Advisor (cont'd)

- GENERATE QUESTIONS FROM GOALS:

  Quality requirements

Are the requirements growing?

Are the requirements changing?

Are the requirements reusable?

Are the requirements testable? ... etc.

MITRE

Example: MITRE Metrics Advisor (cont'd)

- GENERATE METRICS FROM QUESTIONS:

  Are my requirements growing?
  Basic measure: number of requirements
  Repeatable process: number of requirements by requirement type
  Defined process: Mapping of requirements to design/code/test

  Are my requirements changing?
  Basic measure: number of changes to requirements
  Repeatable process: number of requirements changes by requirement type
  Defined process: Mapping of requirements changes to design/code/test

MITRE

Metrics Advisor Screens

MITRE

Welcome to the Metrics Advisor (DISA/CIM's Prototype)

The Metrics Advisor prototype was developed by the MITRE Corporation to support the Department of Defense (DoD) Corporate Information Management Software Metrics Program. Please click the start button to begin your session with the Metrics Advisor.

Questions/comments on the tools should be sent to:

Hillary Davidson
DISA/CIM/XEI
701 S. Courthouse Rd.
Arlington, VA 22204
(703) 285-6584
email:
Hillary_Davidson%cimpfk%cim@CIMgate.osd.mil

or

Jeffrey Heimberger
The MITRE Corporation
7525 Colshire Dr.
McLean, VA 22102
(703) 883-5250
email:
heimberg@mitre.org

Click Here to Start Metrics Advisor

Metrics Advisor Prototype Version: 0.65β

User Information

Please enter the appropriate information
and click on the Main Menu button to continue

Organization Name
Organization 1

Project Name
Project 1

User Name
Naive User

Metrics Advisor Prototype Version: 0.65β

Features | Instructions | Main Menu | End Session

Main Menu

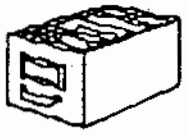
Goals and Questions

Metrics Tutorial

User Profile

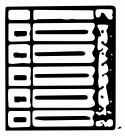
Organization and Project Characteristics

Metrics Glossary



Metrics References

Examine Recommended Metrics



Acronym List

General Information | General Help | Instructions | End Session

High-Level Goals

Select all of the appropriate Goals by clicking the mouse on each goal. A checkmark '√' next to the goal, indicates that it is selected. To deselect or modify a goal choice, click and hold the mouse on a selected goal, wait until the pop-up menu appear, select Modify, or Reset, and release the mouse.

	Improve Productivity
	Improve Quality
	Reduce Risk
	Satisfy Customer

General Help | Instructions | Reset All Goals | Main Menu

Improve Productivity

Select all of the appropriate Goals by clicking the mouse on each goal. A checkmark '√' next to the goal, indicates that it is selected. To deselect or modify a goal choice, click and hold the mouse on a selected goal, wait until the pop-up menu appear, select Modify, or Reset, and release the mouse.

	Resources
	Products
	Process

General Help | Instructions | Reset Sub-Goals | High-Level Goals

Improve Productivity: Products

Select all of the appropriate Goals by clicking the mouse on each goal. A checkmark '√' next to the goal, indicates that it is selected. To deselect or modify a goal choice, click and hold the mouse on a selected goal, wait until the pop-up menu appear, select Modify, or Clear, and release the mouse.

	Early Problem Identification
	Appropriate Technology Products
	Reuse

General Help | Instructions | Sub-Goals | High-Level Goals

Reuse - Questions

Please indicate the questions you would like to answer using metrics by clicking the box next to the desired question.

☐ How much of the system is being built from reused components?

Reset All | Select All | Go Back

Improve - Productivity: - Products

Select all of the appropriate Goals by clicking the mouse on each goal. A checkmark '√' next to the goal, indicates that it is selected. To deselect or modify a goal choice, click and hold the mouse on a selected goal, wait until the pop-up menu appear, select Modify, or Clear, and release the mouse.

	Early Problem Identification
	Appropriate Technology Products
√	Reuse

General Help | Instructions | Sub-Goals | High-Level Goals

Early Problem Identification Questions

Please indicate the questions you would like to answer using metrics by clicking the box next to the desired question.

☐ Are problems being discovered as early possible?

Reset All | Select All | Go Back

Improve Productivity: Products

Select all of the appropriate Goals by clicking the mouse on each goal. A checkmark '√' next to the goal, indicates that it is selected. To deselect or modify a goal choice, click and hold the mouse on a selected goal, wait until the pop-up menu appear, select Modify, or Clear, and release the mouse.

√	Early Problem Identification
	Appropriate Technology Products
√	Reuse

General Help | Instructions | Sub-Goals | High-Level Goals

High-Level Goals

Select all of the appropriate Goals by clicking the mouse on each goal. A checkmark '√' next to the goal, indicates that it is selected. To deselect or modify a goal choice, click and hold the mouse on a selected goal, wait until the pop-up menu appear, select Modify, or Reset, and release the mouse.

√	Improve Productivity
	Improve Quality
	Reduce Risk
	Satisfy Customer

General Help | Instructions | Reset All Goals | Main Menu

Improve Quality

Select all of the appropriate Goals by clicking the mouse on each goal. A checkmark '√' next to the goal, indicates that it is selected. To deselect or modify a goal choice, click and hold the mouse on a selected goal, wait until the pop-up menu appear, select Modify, or Reset, and release the mouse.

	Resources
	Products
	Process

General Help | Instructions | Reset Sub-Goals | High-Level Goals

Improve Quality: Products

Select all of the appropriate Goals by clicking the mouse on each goal. A checkmark '√' next to the goal, indicates that it is selected. To deselect or modify a goal choice, click and hold the mouse on a selected goal, wait until the pop-up menu appear, select Modify, or Reset, and release the mouse.

	Quality of Requirements
	Quality of Design
	Quality of Code
	Quality of Tests
	Quality of Documentation

General Help | Instructions | Sub-Goals | High-Level Goals

Quality of Requirements Questions

Please indicate the questions you would like to answer using metrics by clicking the box next to the desired question.

- ☐ Are the requirements clear and understandable?
- ☐ Are the requirements testable?
- ☒ Are the requirements reusable?
- ☐ Are the requirements maintainable?
- ☐ Are the requirements correct?

Reset All | Select All | Go Back

Quality of Design Questions

Please indicate the questions you would like to answer using metrics by clicking the box next to the desired question.

- ☐ Is the design complete?
- ☐ Is the design clear and understandable?
- ☒ Is the design reusable?
- ☐ Is the design maintainable?
- ☐ Is the design correct?

Reset All | Select All | Go Back

Improve Quality: Products

Select all of the appropriate Goals by clicking the mouse on each goal. A checkmark '√' next to the goal, indicates that it is selected. To deselect or modify a goal choice, click and hold the mouse on a selected goal, wait until the pop-up menu appear, select Modify, or Reset, and release the mouse.

√	Quality of Requirements
√	Quality of Design
	Quality of Code
	Quality of Tests
	Quality of Documentation

General Help | Instructions | Sub-Goals | High-Level Goals

Quality of Code Questions

Please indicate the questions you would like to answer using metrics by clicking the box next to the desired question.

- ☐ Is the code complete?
- ☐ Is the code correct?
- ☐ Is the code clear and understandable?
- ☒ Is the code reusable?
- ☐ Is the code reliable?
- ☐ Is the code maintainable?

Reset All | Select All | Go Back

Quality of Tests Questions

Please indicate the questions you would like to answer using metrics by clicking the box next to the desired question.

- ☐ Do the tests thoroughly exercise the code?
- ☒ Are the tests reusable?

Reset All | Select All | Go Back

Improve Quality:: Products

Select all of the appropriate Goals by clicking the mouse on each goal. A checkmark '√' next to the goal, indicates that it is selected. To deselect or modify a goal choice, click and hold the mouse on a selected goal, wait until the pop-up menu appear, select Modify, or Reset, and release the mouse.

√	Quality of Requirements
√	Quality of Design
√	Quality of Code
√	Quality of Tests
	Quality of Documentation

General Help | Instructions | Sub-Goals | High-Level Goals

High-Level Goals

Select all of the appropriate Goals by clicking the mouse on each goal. A checkmark '√' next to the goal, indicates that it is selected. To deselect or modify a goal choice, click and hold the mouse on a selected goal, wait until the pop-up menu appear, select Modify, or Reset, and release the mouse.

√	Improve Productivity
√	Improve Quality
	Reduce Risk
	Satisfy Customer

General Help | Instructions | Reset All Goals | Main Menu

Main Menu

Goals and Questions

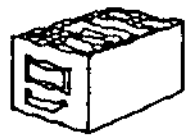
User Profile

Organization and Project Characteristics

Examine Recommended Metrics

Metrics Tutorial

Metrics Glossary



Metrics References

Acronym List

General Information | General Help | Instructions | End Session

Process Maturity Level

Click on Yes or No

Do you know your process maturity level? ○ Yes ○ No

Next Characteristic

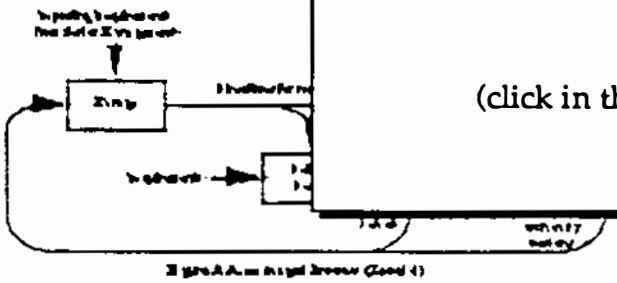
Instructions | Clear Choices | Show Processes | Main Menu

Process Maturity Assessment

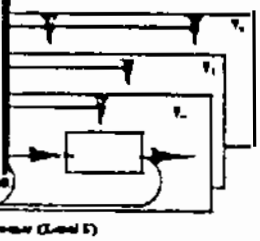
Level 1

To get more information about any of the Process Maturity Levels, move the mouse over any of the diagrams and an information window will appear. Click on the diagram or the information window that best represents your process.

(click in this box to continue)



Level 4

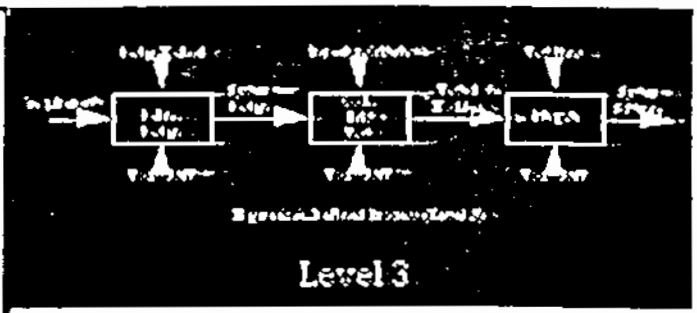


Level 5

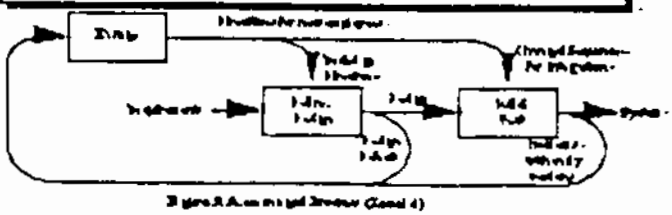
Go Back

Process Maturity Assessment

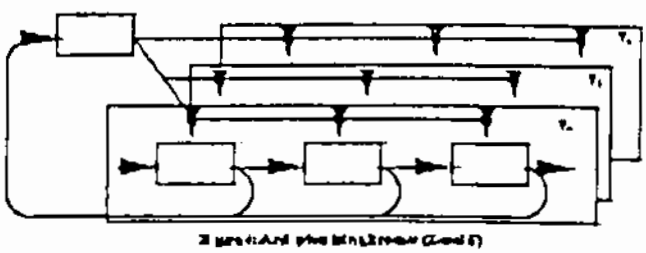
Defined:
- Intermediate activities defined, with known and understood inputs and outputs
- Intermediate activities have clear entry and exit criteria
- Requirements are well-defined and managed
- Final product size can be estimated
- Budget and schedule constraints are known



Level 3



Level 4



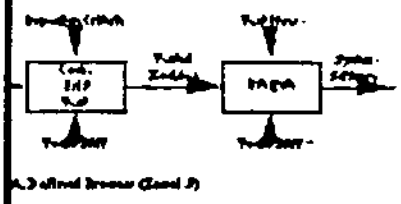
Level 5

Go Back

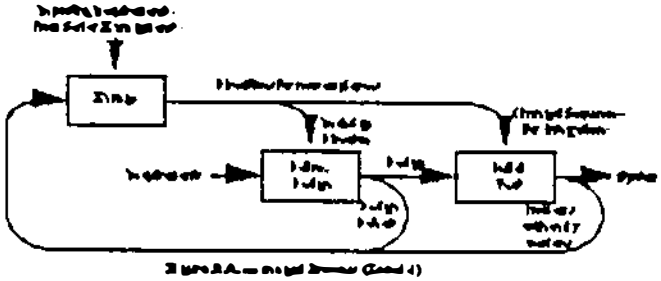
Process Maturity Assessment

Level 1

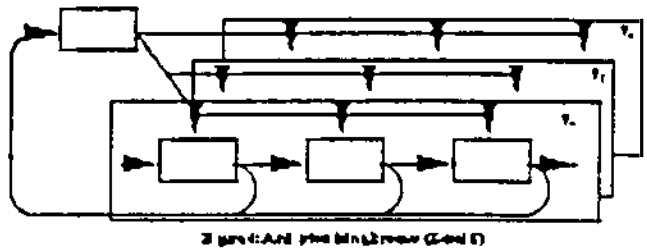
Ad hoc:
- Inputs are ill-defined
- Outputs are expected as software
- Transition from inputs to outputs is undefined, uncontrolled
- Lack of structure and control
- Cannot write down or depict process
- Current levels of quality and productivity are unknown



Level 3



Level 4



Level 5

Go Back

Process Maturity Level

Click on Yes or No

Do you know your process maturity level? ○ Yes ○ No

Select the process level for your organization by clicking the mouse on the appropriate level.

- ○ Level 1: Ad Hoc Process
- ◉ Level 2: Repeatable Process
- ○ Level 3: Defined Process
- ○ Level 4: Managed Process
- ○ Level 5: Optimized Process

Next Characteristic

Instructions | Clear Choices | Show Processes | Main Menu

Branch of Service

Please select at least one

- [x] Army
- [ ] Navy
- [ ] Air Force
- [ ] Defense Logistics Agency
- [ ] Defense Information Systems Agency
- [ ] National Security Agency
- [ ] Others

Previous Characteristic

Instructions | Clear Choices | Main Menu

Main Menu

User Profile

Goals and Questions

Organization and Project Characteristics

Examine Recommended Metrics

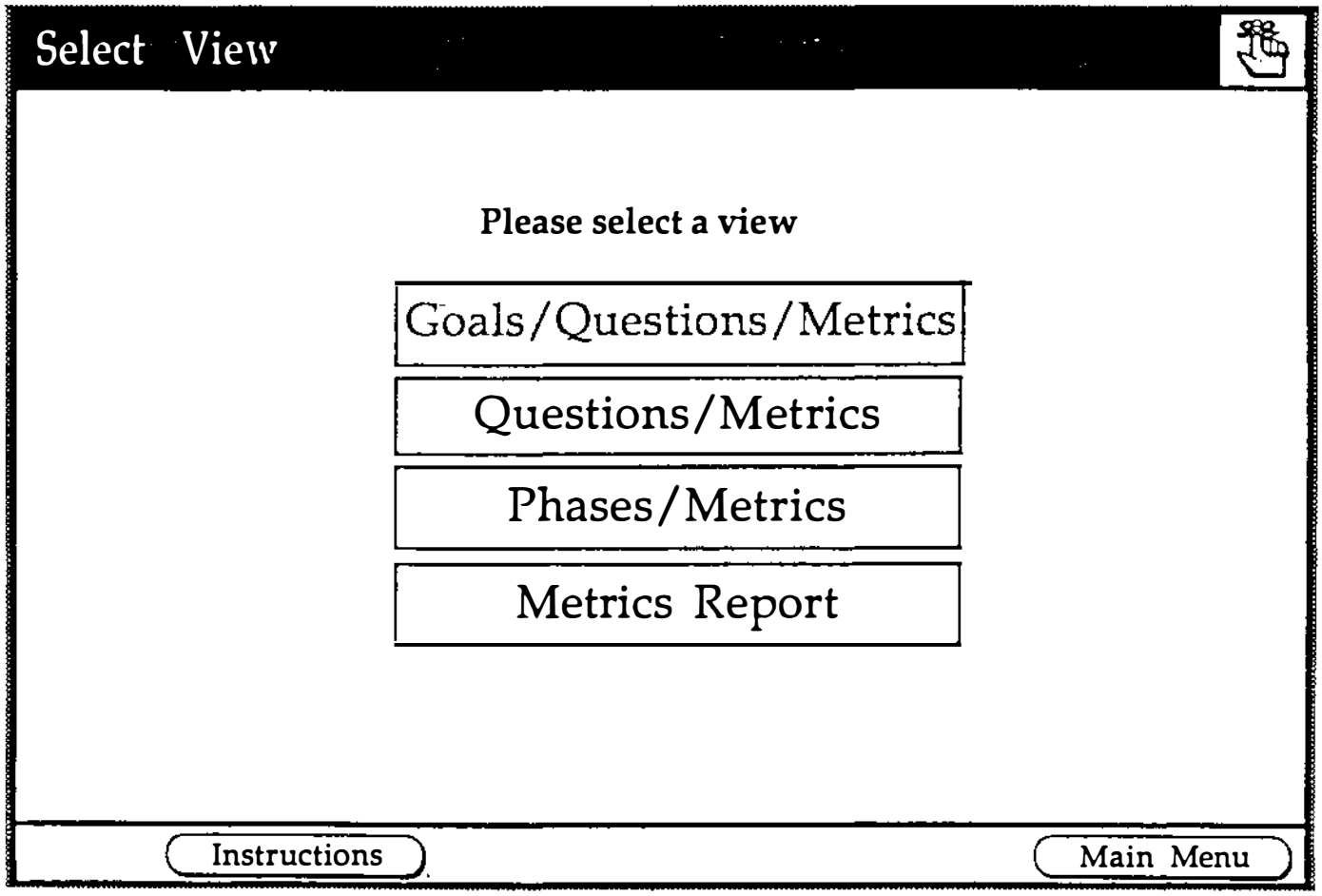
Metrics Tutorial

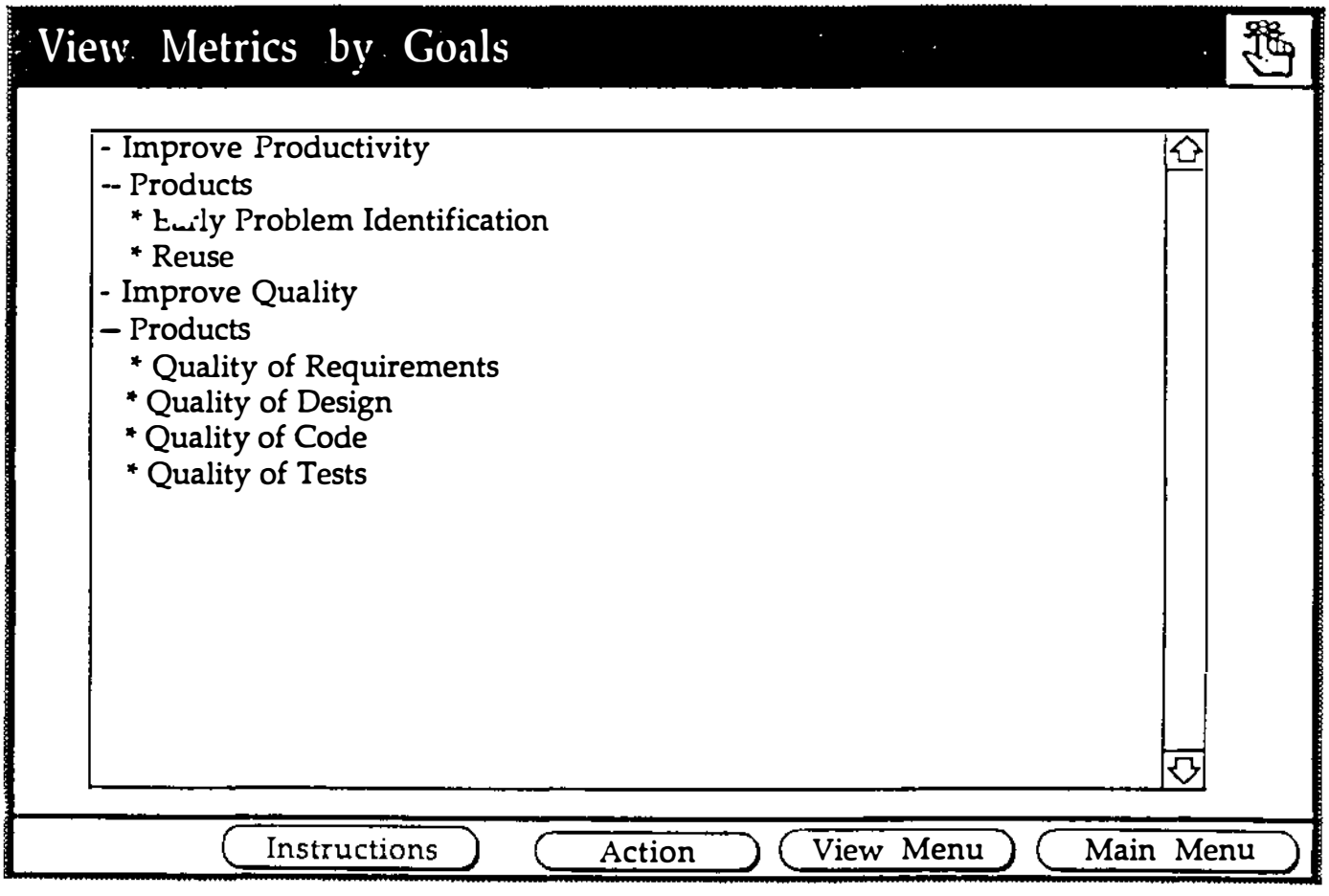
Metrics Glossary

Metrics References

Acronym List

General Information | General Help | Instructions | End Session


Select View
Please select a view
Goals/Questions/Metrics
Questions/Metrics
Phases/Metrics
Metrics Report
Instructions
Main Menu


View Metrics by Goals
- Improve Productivity
-- Products
* Early Problem Identification
* Reuse
- Improve Quality
– Products
* Quality of Requirements
* Quality of Design
* Quality of Code
* Quality of Tests
Instructions
Action
View Menu
Main Menu

View Metrics by Questions

- Are problems being discovered as early possible?
- How much of the system is being built from reused components?
- Are the requirements reusable?
- Is the design reusable?
- Is the code reusable?
- Are the tests reusable?

Instructions | Action | View Menu | Main Menu

Goal: Quality of Design

Question

Process Maturity Level:

Is the design reusable?

Metric

There are no metrics defined for this process maturity level

Next Level Metric

Number of design components; for each design component, number of dependencies on other design components; percent of components with no dependencies

Go Back

Goal: Quality of Requirements

Question

Process Maturity Level:

Are the requirements reusable?

Metric

For each requirement, number of dependencies on other requirements; percent of requirements with no dependencies

Next Level Metric

For each requirement, number of dependencies on other requirements; percent of requirements with no dependencies

Go Back

Goal: Reuse

Question

Process Maturity Level:

How much of the system is being built from reused components?

Metric

Percentage of code new/modified/reused

Next Level Metric

Percentage of design/tests/documents/new/modified/reused

Go Back

Recommended Metrics

Organization Name: Organization 1
Project Name: Project 1
User Name: Naive User
Branch of Service:
Army
Process Maturity Level: 2

- Improve Productivity
— Products
  * Early Problem Identification
    Does the process force errors to be corrected as soon they are found?
    • Mapping of problem to requirement and requirement type, mapping of source to requirement and requirement type
  * Reuse
    Does the process force errors to be corrected as soon they are found?
    • Percentage of code new/modified/reused
- Improve Quality
— Products
  * Quality of Requirements

Instructions | Print Report | Recommendation | View Menu | Main Menu

Metrics Tutorial Main Index

- Process Maturity Level Index
- Software Development Phase Index
- Measurement Area Index
- Return to Metrics Advisor

General Information | Instructions

Process Maturity Level Index

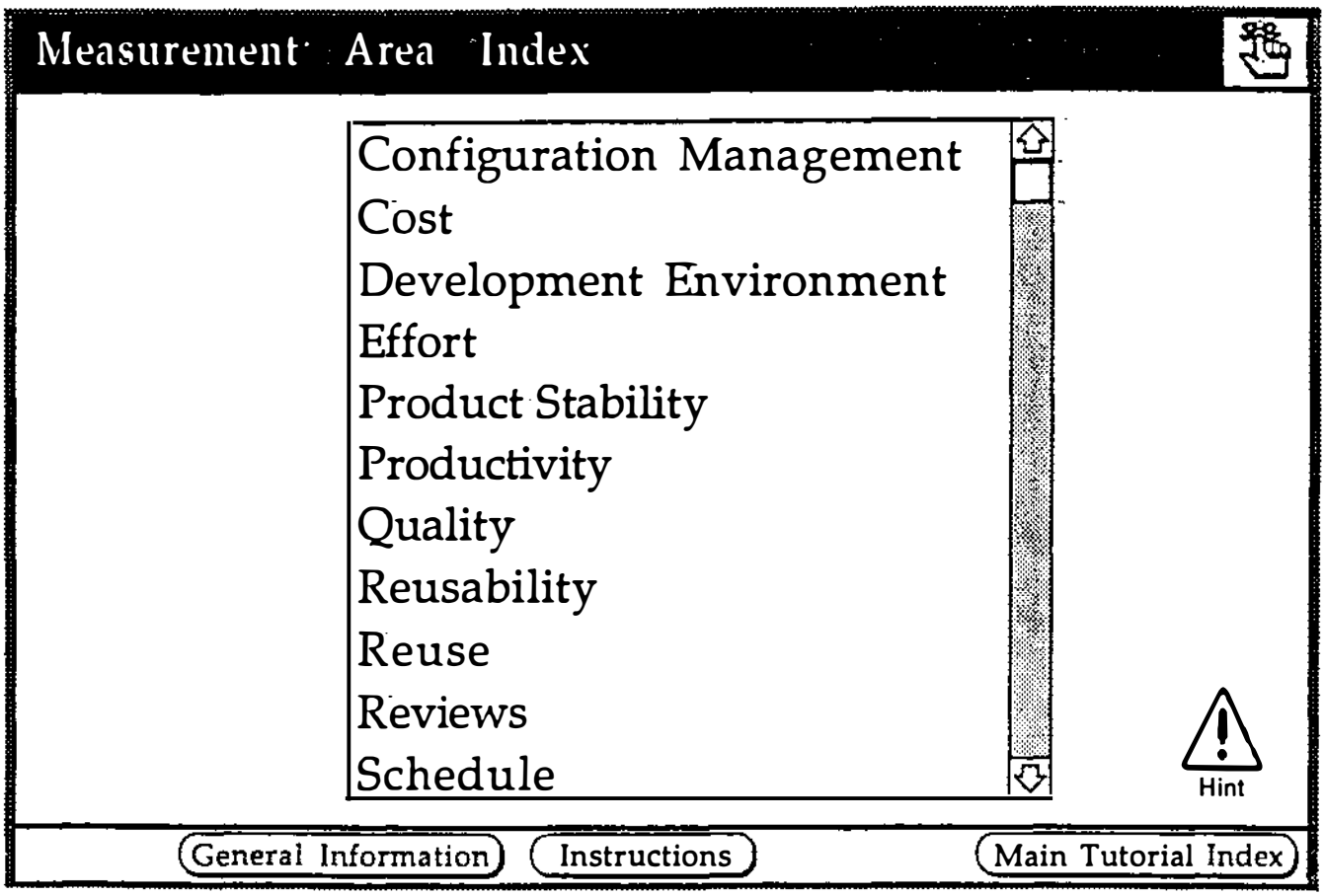
Level 1: Basic
Level 2: Repeatable
Level 3: Defined
Level 4: Managed
Level 5: Optimizing

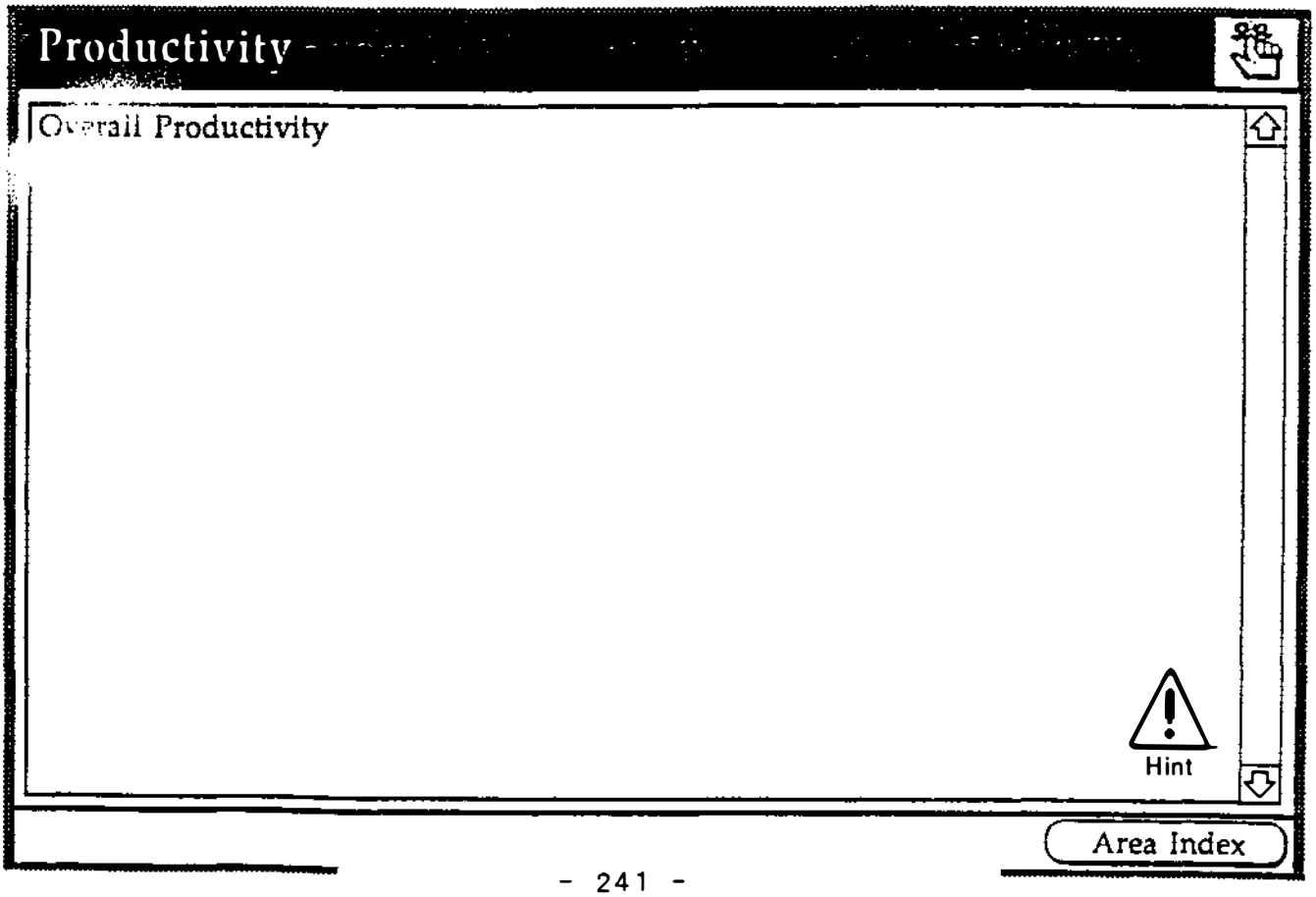
General Information | Instructions | Main Tutorial Index

Software Development Phase Index

Requirements Analysis
Design
Coding
Testing
Operational Deployment
All Phases

General Information | Instructions | Main Tutorial Index


Measurement Area Index
Configuration Management
Cost
Development Environment
Effort
Product Stability
Productivity
Quality
Reusability
Reuse
Reviews
Schedule
Hint
General Information
Instructions
Main Tutorial Index


Productivity
Overall Productivity
Hint
Area Index

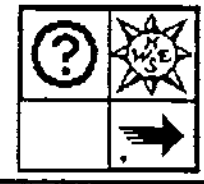
Overall Productivity Definition

The amount of product per unit of effort.

Productivity can be measured for all software development life cycle phases, and for all software development products. As a result, many things affect productivity, including the experience of the staff and management, tool availability and usage, early detection of errors, and reuse.

Click on the buttons below to see more information on size and effort.

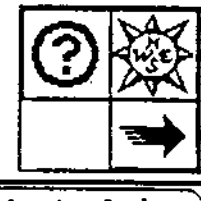
Size | Effort



General Information | Instructions | Metrics Index

Size of Products Definition

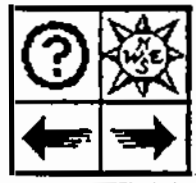
The size of intermediate or end products of software development. Size can include size of requirements, design, code, test plans, and documentation.



General Information | Instructions | Metrics Index

Size of Products Uses

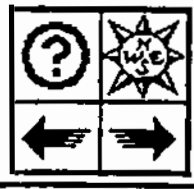
- Tracking progress (plans vs. actuals)
- Cost estimation
- Normalization of other measures (effort or defects per unit size)



General Information | Instructions | Metrics Index

Size of Products Examples

- Number of requirements
- Number of design modules
- Number of objects and methods
- Number of lines of code
- Function points
- Number of test cases



General Information | Instructions | Metrics Index

Size of Products Actions

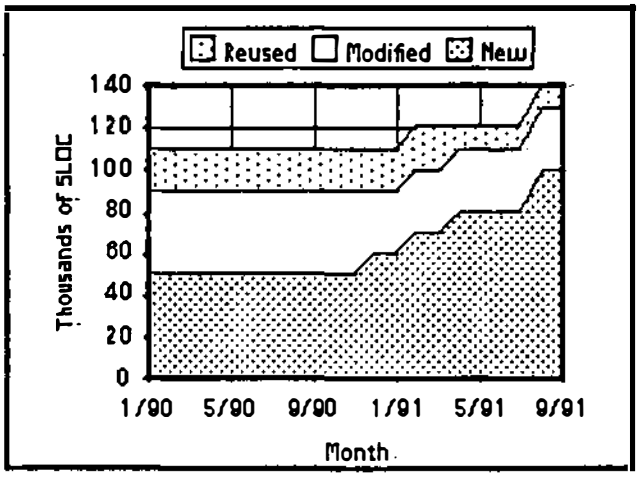
Question:

How much of the system is being built rom reused components?

Actions:

Possible Actions:

- Increase amount of reuse by creating a procedure that ensures every opportunity for reuse is fully



Interpretation:

The chart at the left [SHU88] shows the amount of reused, modified, and new software for a particular project, over time. The shaded areas are "stacked," i.e., they add up to produce a

General Information | Instructions | Metrics Index

Size of Products Actions (continued)

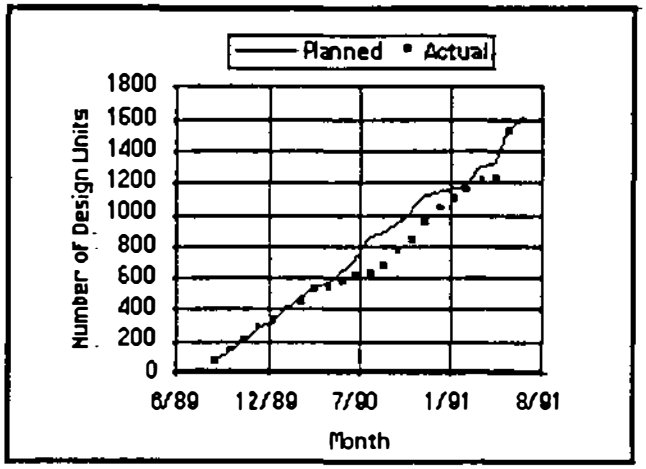
Question:

Is software development progressing as planned?

Actions:

Possible Actions:

- Increase human resources or physical resources (e.g., tools, workstations) dedicated to software



Interpretation:

The chart at left shows the planned and actual design units for a large software development project.

The chart illustrates that the design activity fell behind its planned

General Information | Instructions | Metrics Index

Size of Products Actions (concluded)

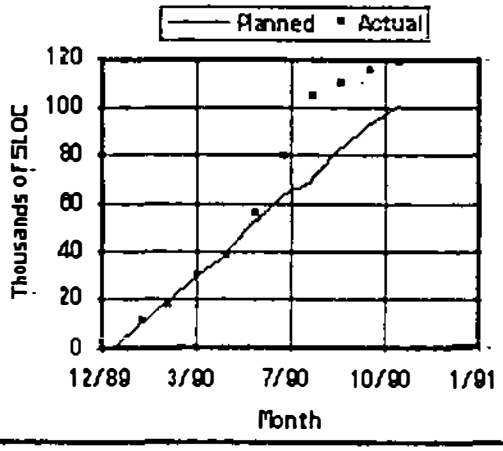
Question:

Is software development progressing as planned? (cont.)

Actions:

Possible Actions:

- Increase human resources or physical resources (e.g., tools, workstations) dedicated to software



Interpretation:

The chart at left is another way of showing progress in source code development. This chart shows planned (line) vs. actual (dots) source lines of code over time.

General Information | Instructions | Metrics Index

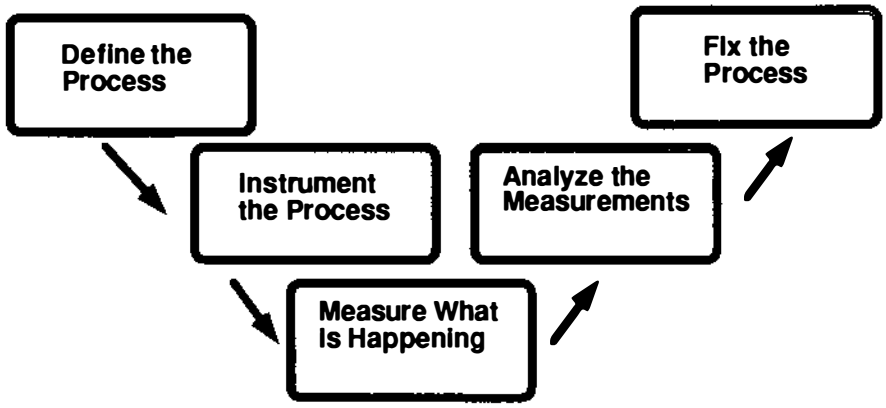
Size of Products References

Reference Annotation

[AFSC86]	Air Force Systems Command Software Management Indicators.
[AFSC87]	Air Force Systems Command Software Quality Indicators.
[BETZ91]	U.S. Army STEP Metrics.
[IFPUG92]	Int'l Function Point User's Group description of function points.
[PFL89]	Pleeger recommends an initial set of software metrics.
[PFL91]	Pfleeger's book on software engineering.
[SHULTZ88]	Shultz describes various software metrics.
[SSSMDWG91]	SEI's recommendations for counting source lines of code.
[VJ92]	Verner and Tate provide empirical support for funtion points.

General Information | Instructions | Metrics Index

Process to Metrics to Process



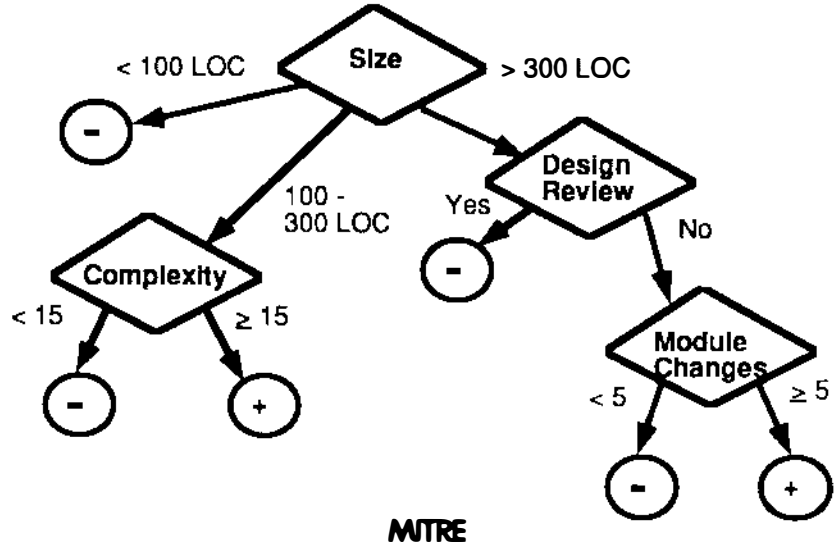
MITRE

Data Analysis from a Process Perspective

- Classification trees
- Multiple Metrics graphs

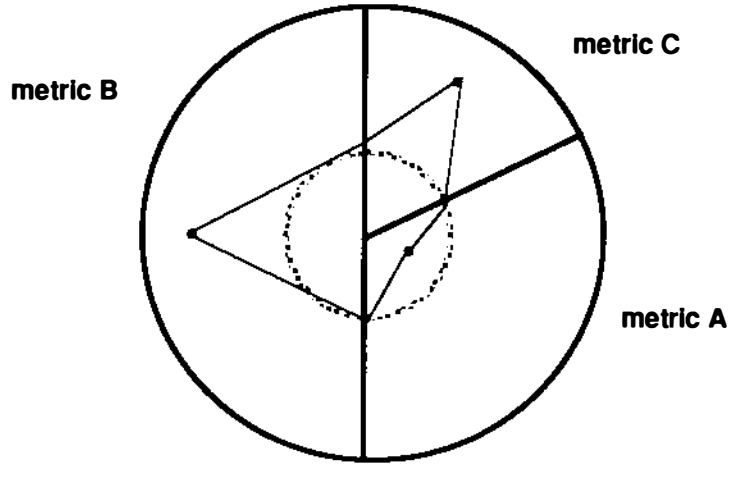
MITRE

Classification Trees



MITRE

Multiple Metrics Graphs



MITRE

Summary and Conclusions

- Process maturity and project goals suggest strategy for metrics: There is no universal metrics set!
- Begin by setting goals and defining process: You can't plan how to get there if you don't know where you're going!
- Associate metrics with each goal: If you don't know why you are measuring and what you will do with the measure, then you shouldn't be measuring it!
- Start small and implement goals according to project priority; process and product improvement will follow.
- The goal is process and product improvement, not just metrics implementation!

MITRE

References

Grady and Caswell, *Software Metrics: Establishing a Company-wide Program*, Prentice-Hall, 1988

Conte, Dunsmore and Shen, *Software Metrics and Models*, Benjamin Cummings, 1986

Rifkin and Cox, *Measurement in Practice*, CMU/SEI TR-16, July 1991

Pfleeger, *Recommendations for an Initial Metrics Set*, Contel Technology Center Report CTC-TR-89-017, December 1989

Pfleeger and McGowan, "Software Metrics in a Process Maturity Framework," *Journal of Systems and Software*, July 1990

Pfleeger, "Process Maturity as a Framework for CASE Tool Selection," *Information and Software Technology*, November 1991

Pfleeger, "Building a Corporate Metrics Program," *IEEE Software*, to appear

MITRE

An Approach to Test Management using an Entity-Relationship Model

Rukmani Tekchandani

Intel Corporation

5200 NE Elam Young Pkwy

Hillsboro, OR 97124-6497

email: ruku_tekchandani@ccm.hf.intel.com

Abstract

Often ignored during test automation is the management of the information flow. This paper describes an entity-relationship model for representing test objects, and applies the model to manage the testing process.

Biography

Rukmani Tekchandani is a software engineer at Intel Corporation. She has been involved with improving and automating the testing process of the C Compilers. She received her M.S. from Oregon State University in 1987.

1. Introduction

Testing is usually a very ad-hoc process comprising a number of complex, time-consuming activities such as test planning, test development, test execution, and error reporting. Most of the testing activities are interrelated in the sense that the output from one activity may force a tester to make modifications to the next activity. The information flow in the testing process is so overwhelming that it gets difficult for the tester to make the right decisions at the right time.

There is no doubt that the major portion of the testing life-cycle is spent in executing tests and verifying test results. As a consequence we see that there are many tools available that address the test efficiency by automating test execution. Many of these tools can be placed in the capture-replay category. Our experience with testing shows that after test execution most of the testing time is spent in managing the testing process. An automated test management system can free the tester from routine work by providing a powerful, integrated test environment to manage information flow between various tasks associated with the testing process. A properly designed system not only helps in storing, organizing, executing, and reporting test results but also helps in making test decisions based on output from the ongoing testing activities.

2. Overview of a Testing Process

Testing is a very complex process; literally hundreds and perhaps thousands of details need to be accounted for and controlled. The objective of this section is to define a typical testing process and to identify the flow of information that is interesting from the view of test management.

A typical testing process usually starts with a *test planning* phase. This phase uses information from the project documents to define a strategy for testing the product. Testing resources are identified and a schedule is published. The most important output from the test plan is the tasks that need to be executed to test the product. Managing the testing tasks is quite complex, as the number and priority of tasks keep changing with the changing requirements of the product. Some organizations use project management tools to automate this phase of the project.

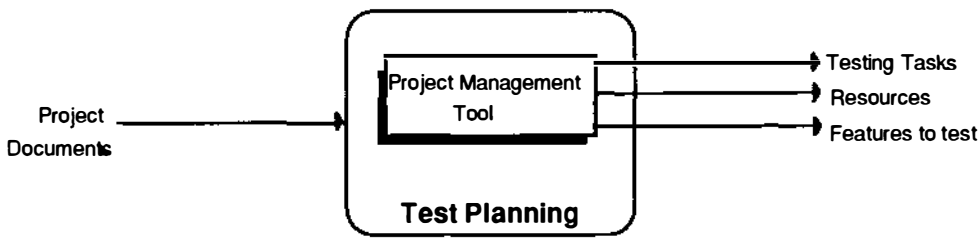


Fig 1: Information flow in the Test Planning Phase

The *test preparation* phase involves creating or modifying test suites and test procedures. Tools like Test Coverage Analyzers are often used to identify untested areas in the product, and sometimes test case generators are used to create tests. Some of the difficulties encountered in this phase are management of test cases and the association of test cases to the product features.

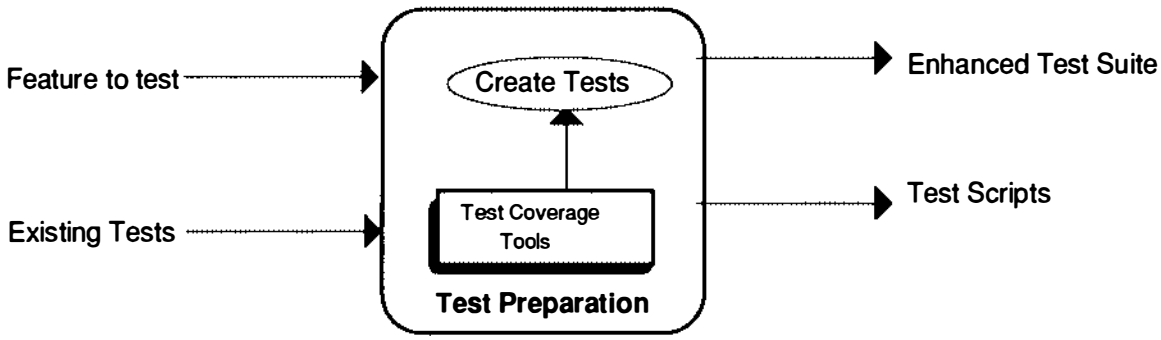


Fig 2: Information Flow in the Test Preparation Phase

The *test execution* and *analysis* phase uses test cases to test the product and produces problem reports and a summary report. This phase is usually automated first in any organization. Log files and report files are used in this phase to identify software failures and testing failures.

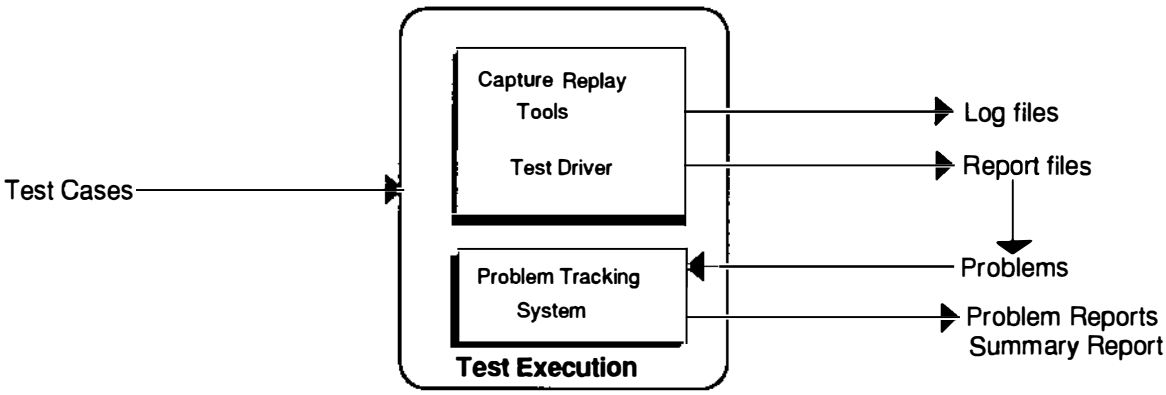


Fig 3: Information Flow in the Test Execution Phase

Every test cycle ends in the *test reporting* phase. In this phase, test results are formally recorded, and related to the test cases and the product features. This phase is often omitted because it is assumed that test results are by this time sufficiently well known to the development team.

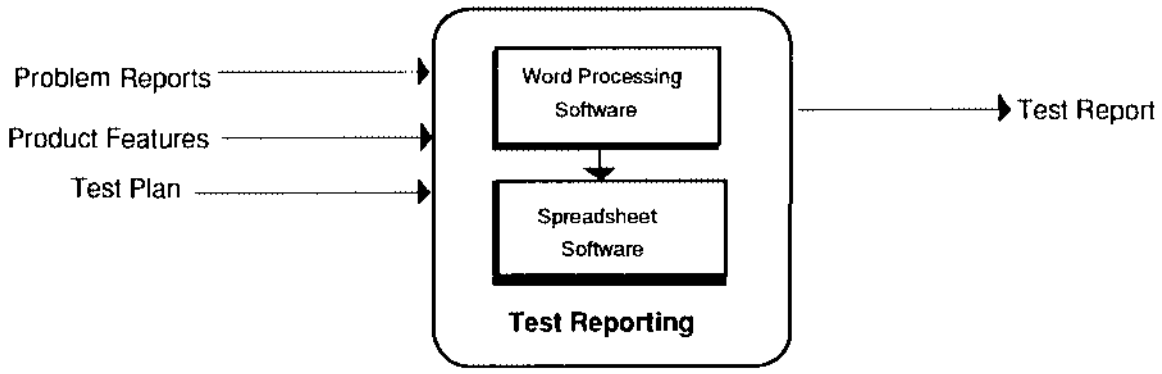


Fig 4: Information flow in the Test Reporting Phase

3. Test Management

During the course of the testing process, many test objects are created, updated, and deleted. Some of the test objects worth mentioning are test cases, test results, and testing activities. Most of the testing objects are related to each other through some common attributes. By representing the test objects and their relationships in an entity-relationship model, the testing process can be managed automatically. In Section 3.1, some of the common testing objects are identified and defined, and in Section 3.2, the relationships between objects are made explicit by creating an entity-relationship model.

3.1 Basic Entities of the Test Management Database

The first step in developing a conceptual schema is to locate and identify the entity types about which the test management system will need to store the information. From Section 2, the following entity types are evident:

- ***Task*** *(testing activity)* A task is an activity necessary to test a software product. For example, creating a test and running a test are examples of testing tasks. Each task requires two kinds of resources: a person responsible for the task and a hardware/software platform required for testing.

- ***Tester*** *(Staffing for testing)* A tester is the person responsible for carrying out the testing tasks.

- ***Test Item*** *(Software to be tested)* A test item is the software

item that is the object of testing. A test item can be a product, a subproduct, a component, a module, or a function.

• ***Platform*** *(Computer System)* A platform is the computer system on which the test item is going to be tested. This information is very useful if you are running the tests on more than one platform.

• ***Version*** A version is the version number of the test item.

• ***Feature*** *(Test Item Feature)* A feature represents a product feature, a product requirement, or a test objective. In other words a feature represents something in a test item that needs to be tested. For example, structure packing is a compiler feature, and will be counted as a feature of the compiler to be tested.

• ***Test Suite*** A test Suite is a collection of tests that test a feature of the test item. For example, "structure test suite" for the test item compiler will have all tests necessary to test the structure data type.

• ***Test*** In this document, a Test is defined as that piece of software, which can be executed in a standalone manner. Each test may consist of one or more than one test conditions.

• ***Test Cycle*** A Test Cycle is the process of running selected tests on a given version of the test item.

• ***Test Result*** A Test Result contains the result (pass/fail) of executing a test.

• ***Problem*** A Problem is the complete description of a test result when reported as a problem in a problem tracking system.

3.2 Defining an Entity-Relationship Model

The entity-relationship model is very useful for identifying relationships between entities. In this model, entities are represented as nodes and relationships as edges.

For the sake of modeling, the testing process can be thought of as a series of testing activities performed by a tester. Each testing activity is represented by the node named **task**. In this model, each task is performed by one tester and is executed on one platform. Note that the term *execution* in this context means accomplishing a task.

Each task is also related to **feature**, in other words, each testing activity is related to testing a certain product feature. By establishing a one-to-one relationship between the task and the feature, we ensure that we can predict the quality of the product in terms of the testing of product features.

To ensure feature test coverage, it is necessary to define a relationship between a feature and the test suite that needs to be executed to test that feature. Thus, all tests pertaining to testing a

single feature are grouped into a test suite and the test suite is linked to the feature. Each feature can have one test suite, and each test suite can have many tests.

All features belong to a test item. Remember that a test item is a part of the product that needs to be tested. A Feature entity can also represent test objectives or product requirements. Thus all tests necessary to test a test item can be found by tracing through all product feature entries.

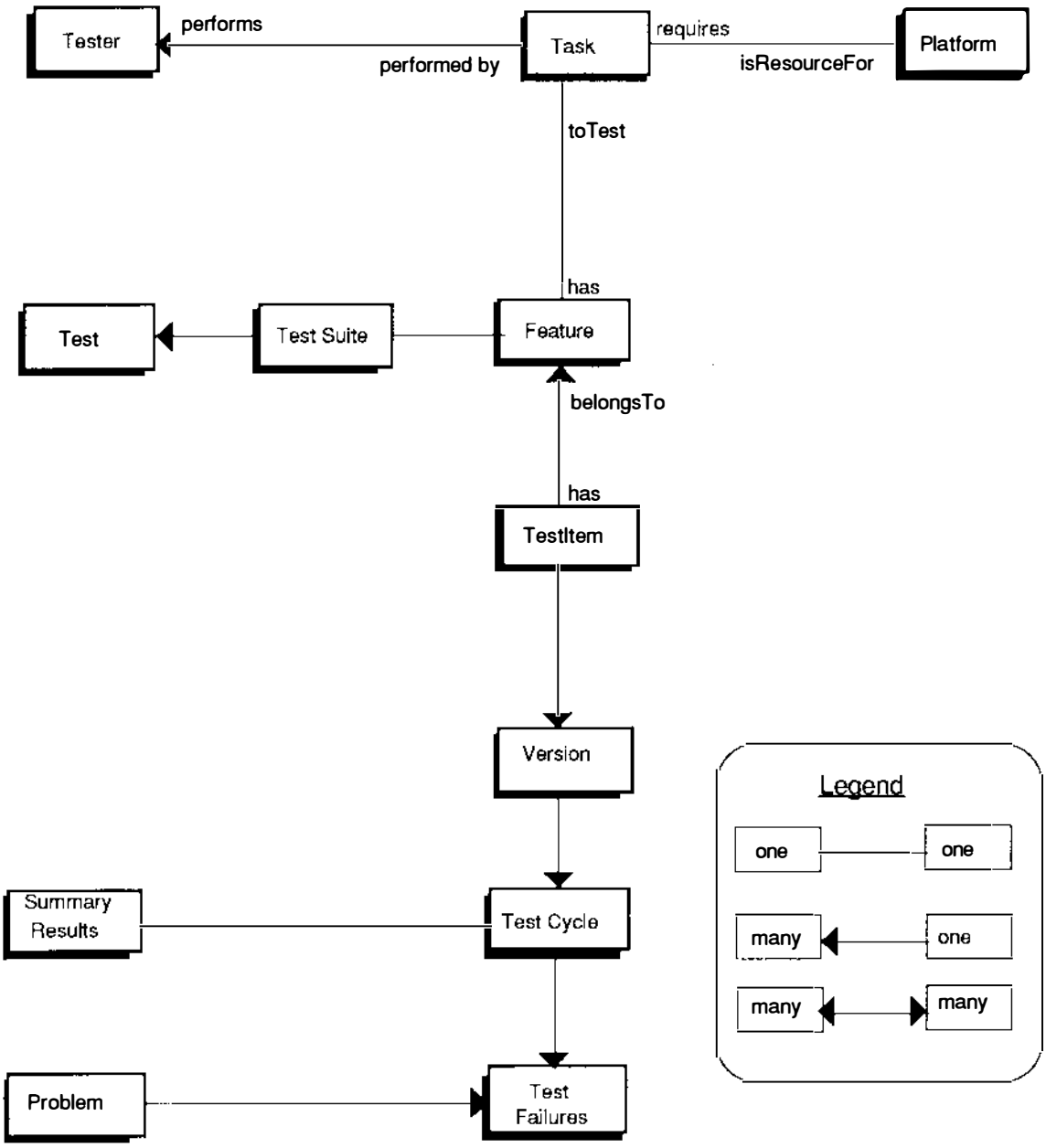


Fig 5: Entity-Relationship Model of Test Management System

One of the drawbacks of the Entity-Relationship model is that it does not provide facilities to represent hierarchical structure. The relationships between various test items belonging to a product are thus lost. This problem can be overcome by keeping an attribute called parent-id with each test item, which is simply a link from a test item to its parent in the hierarchy.

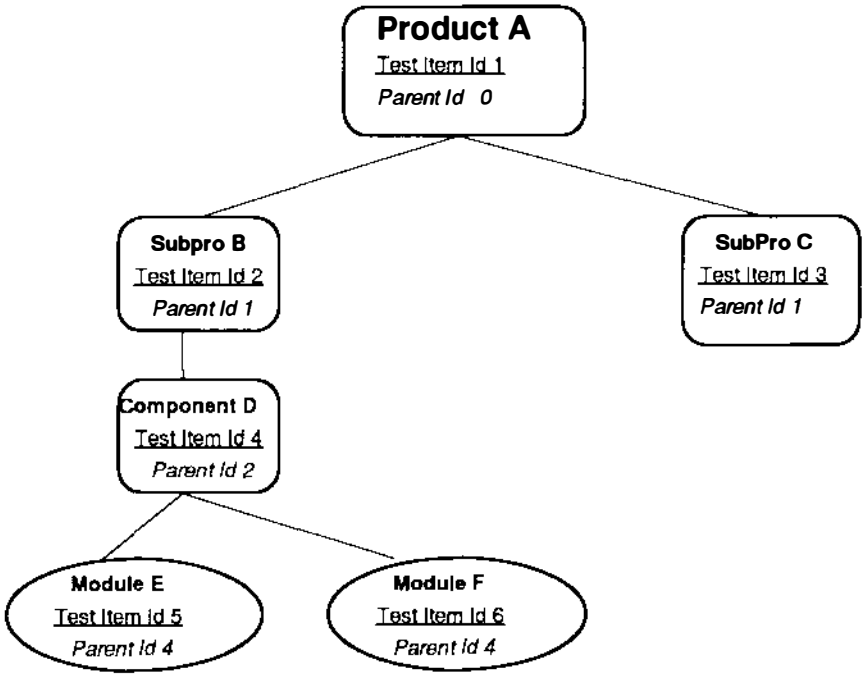


Fig 6: Inter-relationships of the Product components

Each test item goes through many versions during the development phase of the product. Each version goes through one to many test cycles. It is necessary to store test results for each test cycle for all versions. Each test cycle has a one-to-one relationship with summary results, and a one-to-many relationship with test failures. Each test failure is again related to a problem entity. The problem entity can be a part of the test management system or it can be a separate problem tracking system.

4. Application of the Entity-Relationship Model to the Testing Process

The Entity-Relationship model defined in the previous section provides a conceptual schema of a test management system. An implementation of the Test Management System (TMS) in an organization may include interface routines to access the data according to their need. This section describes the kind of management issues that can be addressed by a TMS. Please note that the model assumes an interface to a problem tracking system, and so this section does not cover the issues like bug statistics that can be addressed only by a problem tracking system.

- **Provide management of testing activities**

A Test Management System(TMS) keeps records of testing tasks in the test management database. Examples of some testing tasks are: creating tests, running tests, and analyzing results. Each task is related to a tester, a feature, and a platform entity. This means that for any given task it is possible to know who is responsible for carrying out the action, the hardware/software that is required for the task execution, and the product feature that will be tested by carrying out the task.

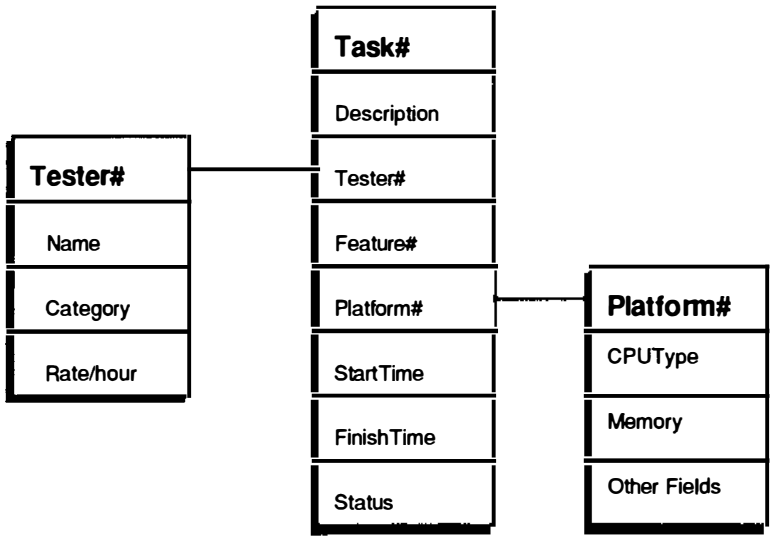


Fig 7: Relationship Between Task, Tester and Feature

The TMS may provide an interface to a Project Management Tool to manage testing activities and resources. For example, the following questions can be answered by this model:

1. *What is the next scheduled testing activity?*
2. *Who is testing Feature X?*
3. *When will the testing be complete?*
4. *How many features remain to be tested?*
5. *What has been done so far?*
6. *For any task, who did it and how long did it take? Or how much did it cost to execute an activity?*
7. *What kind of hardware/platform do I need from date XX/YY/ZZ to X1/Y1/Z1?*

- **Provide assistance in predicting testing cost**

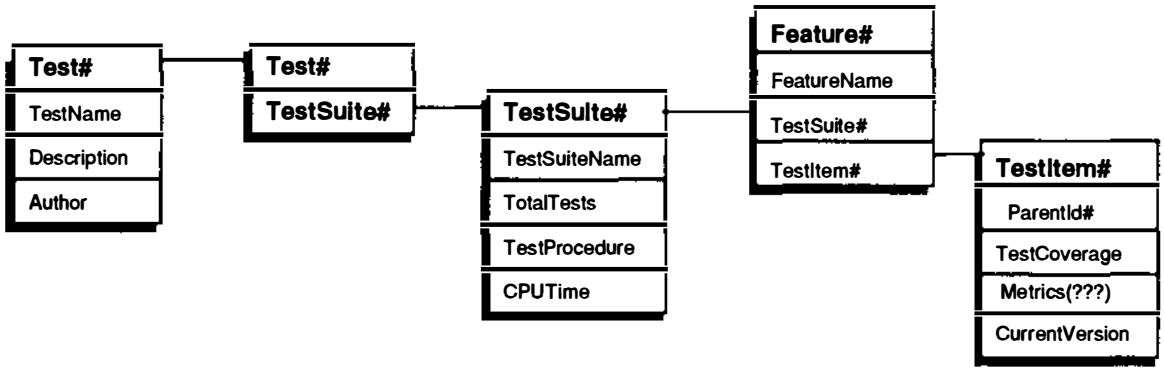
By adding the cost of all resources required to complete all tasks related to a feature, a TMS is able to estimate the cost of testing a feature. Combining testing costs for all features will give you the cost to test the complete product. The cost information combined with

the number of defects per feature will help reallocate resources. Here are some examples of cost-related questions:

1. What is the cost of testing Feature-X?
2. How much time will it take to test the complete product?
3. Will the cost increase/decrease by adding/removing resources?

- **Provide Requirement Traceability Management**

The underlying theory behind the model proposed in this paper is the concept of testing-by-features. The features in this context mean a product feature, a requirement or a test objective. Thus, all testing activities have a goal of testing a feature. All activities, test results, and tests are traceable back to a specific product feature.



On the basis of the relationships between product features and tests, the following questions can be answered:

1. How much time is required to execute tests for Feature-X?
2. Are all product features being tested?

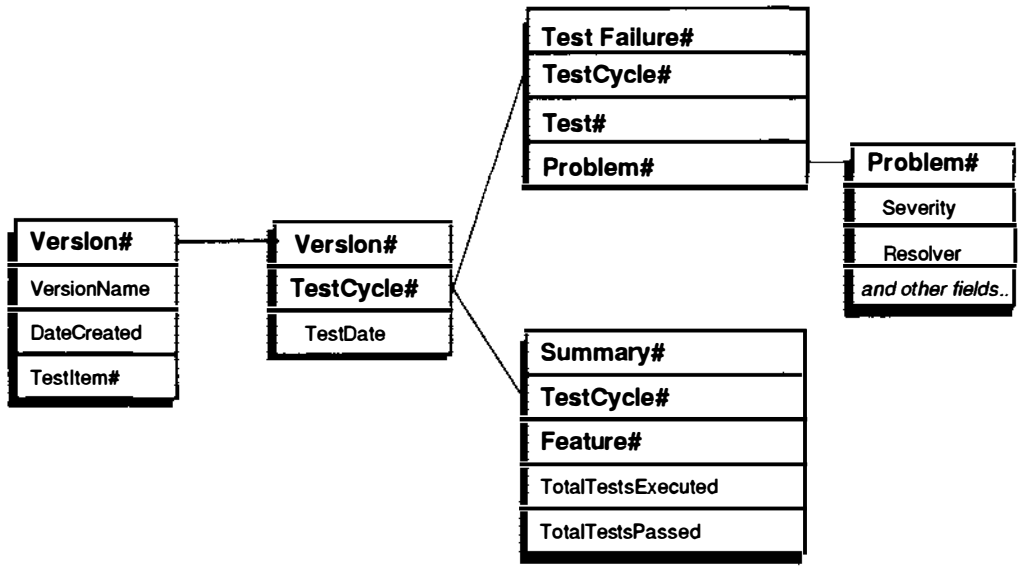
- **Provide management of testing multiple versions**

One of the difficult activities during testing is to compare the testing results of multiple versions of each **testitem**. The TMS model provides the capability by linking **testitem** entity with a version entity by a one-to-many relationship. Thus, the test results always point back to a version entity. The kind of questions that can be answered are:

1. How does the version under test compare to the previous version of the testitem?

- **Provide association between test cases and problem reports**

Each test failure is linked with the problem in the problem tracking system and with the test case in the TMS. This allows the tester to select test cases on the basis of past failures.



- **Provide reporting of testing status**

The report writing process works backwards to develop an opinion on the completeness and correctness of the system at this point. The following are typical pieces of information that are desired in status reporting:

1. *How many tests were executed per testing cycle/version/feature?*
2. *How many tests executed correctly per testing cycle/version/feature?*
3. *How many problems were found per testing cycle/version/feature?*

- **Provide assistance in creating test cases**

The test coverage percentage and other related metrics are maintained with each testitem. This stored data provides the basis for creating new tests. Also reporting the total number of problems by category and frequency helps identify defect-prone components of the system. The following questions can be answered by the system:

1. *What is the current percentage of test coverage?*
2. *How many components have complexity (or any other metric) higher than X?*

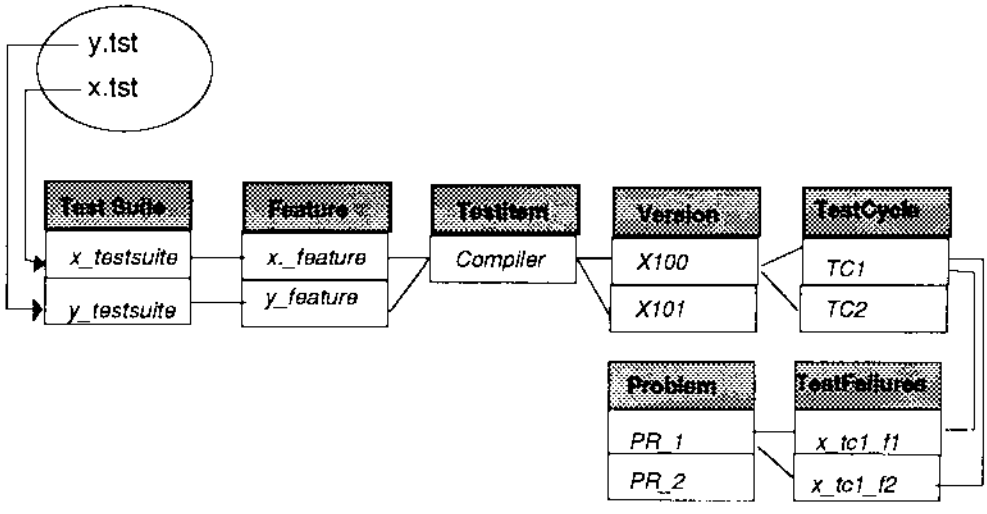
6. Experiences With Building a Prototype TMS Tool

The concept of an automated test management system developed during the testing of the Code Builder™ product at Intel Corporation. The product Code Builder™ consists of many components such as a compiler, libraries, a linker, a make utility, and a debugger. The compiler tests consisted of two very large test suites that took almost two days to run. For any minor change to the compiler, it was always necessary to run the complete test suite. This inspired us to invent the testing-by-feature technique. The test suites were analyzed and all tests were re-categorized into feature groups and assigned to a compiler feature. After this was done, it was easier to select what tests to run for any given compiler feature change.

We chose Borland's Paradox database to create the Test Management System. We already had a problem tracking system in place, so at the start of the test cycle we dumped all problem reports in the Problem database in the Paradox.

The TMS prototype implementation started by listing all the features of the compiler that we wanted to test. Next, we grouped the current test suites according to the features, and created a test driver (x.tst, y.tst, ...) for each test suite. The Test Suite database was created, that contained the name of the test driver. The version entry and testcycle entries were added to the databases by the main test driver. The test execution produced a fail-file that was imported in Paradox. If the fail-entry did not match any record in the TestFailure database, it was flagged as a new failure. The filtering of failures into known and new failures reduced Test Analysis time by almost 50%.

The known bugs were linked to a problem database, and any known bug with fixed resolution status was flagged as a regression.



The test results could be consumed by word processor programs to generate test reports without any delay. It became possible to compare any two versions of the compiler from the stored results in the database. Testing time was recorded for each test cycle and was used to estimate the time

required for the next test cycle.

Use of the Test Management System increased productivity by reducing the time required to perform Test Analysis. Test Reports were generated for each test cycle, and became basis for decision-making. Product testing was increased; time gained was used to write more tests.

7. Conclusion

An approach to automation of test management using a relational database has been described in this paper. The benefits from this model were greatly felt in the use of its prototype implementation. By automating test management, you also get a user friendly integrated test environment, and consistent management of test log and report files. A flexible report generator can be built to extract information from the database for management and test administrator. Use of an automated test management system also enforces corporate standard and guidelines.

Test Management has started to get some notice in software testing literature and industry. This area is still in its infancy and more research is required. Automation of test management can greatly reduce the frustrations faced by most testers. The increased quality testing will then eventually result in better quality products.

8. References

[1] Bill Hetzel, *The complete Guide to Software Testing.*
[2] William E. Perry. *A Standard for Testing Application Software.*
[3] DBMS Developing Corporate Applications, Oct-91-Feb 92.

CGEN
Automatic program generation to ensure reliability

Ray Lischner
Mentor Graphics Corporation
8005 SW Boeckman Rd.
Wilsonville, OR 97070–7777
e-mail: Ray_Lischner@mentorg.com

August 13, 1992

Abstract

The *cgen* program is a tool to generate reliable C++ source code from user-supplied prototypes and function interfaces. This paper examines how *cgen* can be used to improve software quality by relieving the programmer of an otherwise error-prone task.

About the Author

Ray Lischner is a software engineer who has been actively pursuing software quality at an assortment of companies. Mr. Lischner holds a B.S. from the California Institute of Technology.

1 Introduction

Writing correct software is never simple. The *cgen* program helps a little bit by automatically generating function implementations, based on interfaces determined by the user. Specifically, *cgen* reads C++ class declarations and implements certain member functions.

C++ is a complex, object-oriented programming language. Part of its nature is the separation of a class's interface from its implementation. A common approach to ensuring that an interface accurately reflects the implementation is to use a program that generates the interface automatically, based on the implementation. The *cgen* program, however, works the other way around, namely, generating an implementation based on an interface. Using *cgen* has improved programmer productivity and quality, when compared with implementing the same code by hand.

Many member functions follow a consistent pattern within a system. The *cgen* program implements member functions based on patterns and class declarations, thereby relieving an otherwise tedious and error-prone task from the programmer.

The framework in which *cgen* operates is based on the NIH classes, written by Keith Gorlen of the National Institutes for Health [2]. The NIH classes have a number of member functions in common, which can all be implemented in a similar manner. Classes that derive from the NIH classes often need to redefine these functions, albeit following the same pattern.

The user writes the interface, in the form of a header file, and runs *cgen* to produce the implementation as a C++ source file. Whenever the interface is modified, a new implementation is generated by running *cgen* again.

The member functions that are defined by NIH are built into the *cgen* program. The user can add new function generators, although it is not an easy process.

This article explains the rationale for *cgen* (Section 2), followed by examples of its use (Section 3). The design and implementation of *cgen* is described in Section 4, and Section 5 presents how *cgen* is used. For those not familiar with the NIH classes, Section 6 introduces their design and use, and also explains the modifications made at Mentor Graphics. Finally, conclusions are presented in Section 7, with a discussion of the future of *cgen* and various ideas for its successor.

The *cgen* program is used for real applications at Mentor Graphics. The examples that are shown in Section 3 are taken from internal tools that are in daily use at Mentor Graphics. The examples are compared with a similar C++ project that does not use *cgen*, with consideration for the issues of continuing development and changing interfaces.

2 Implementations and interfaces

The traditional programming method used for C++ programming is is to write a class interface in a *header* file, and write the class's implementation in a *source* file. Since they are written separately, there is always a maintenance problem of ensuring that the interface accurately represents the implementation, so the C++ compiler ensures that the interface and implementation match. It is a relatively simple matter to generate the class interface automatically from the implementation, thereby guaranteeing that the interface is accurate.

The *cgen* approach is different, however. The user writes the interface, and *cgen* produces parts of the implementation. Clearly, a program cannot produce the entire implementation, but there are usually several functions that follow simple patterns.

In particular, the kinds of functions that *cgen* is best suited for include simple data member access, comparison, construction, copying, and so on. The C++ compiler handles the barest minimum of such functionality: just member-wise copying, and trivial construction. The programmer must write member functions to perform deep copying, non-trivial construction, comparison, and so on.

The *cgen* tool depends on the NIH classes. All NIH classes are derived from a common base class, *NIH_Object*. This class defines the interface for several member functions. To derive an NIH class, the programmer must redeclare these functions and provide implementations. The functionality provided by NIH includes I/O, deep copying, comparison, and so on.

The implementation for many of these functions follow similar patterns. For example, for deep copying, a member-wise copy is performed, and then a deep copy is performed for all data members that are pointers to other NIH classes. A deep copy continues recursively, until every object has been copied. (NIH takes care of cycles in the object graph.) The implementation of the deep copy function is very regular, depending only on the types of the class's data members. Such a function is best implemented by the compiler itself, but since this functionality is not part of the C++ language, the second best solution is for a program to generate these implementations. Hence the existence of *cgen*.

C++ uses static type checking, with no run-time type support. The programmer can *cast* an object from a derived class to a base class, but not the other way around. The NIH classes implement a run-time type mechanism, including a type-safe means of casting from a base class to a derived class. The *cgen* tool also produces the information that NIH needs to implement this run-time type mechanism.

3 Examples

3.1 Before cgen

The need for *cgen* became evident when working on a processor for a hardware description language. The system currently includes almost 100 classes, all declared in a single header file of about 5000 lines. At first, the class implementations were written by hand. It quickly became obvious that there were distinct patterns to many of the functions, and that the process could be automated.

The need for automation became particularly clear when minor defects were found in the function implementations: mostly typographical errors, and omitted code. Many of these errors are difficult to detect, and when they result in failure, are difficult to trace to the original defect.

The most common error was the omission of a virtual base class from the class constructors. Since every virtual base class has a constructor with no arguments, the code was semantically correct as far as the compiler could tell.

Another common error was to introduce a discrepancy between the load and store functions. Sometimes data members were written and read in different orders; sometimes a data member was omitted from the read or write function. At the time, our *ad hoc* development process did not include inspections, so these problems were not noticed until they resulted in software failure. These failures typically occurred in a read function, typically the one called *after* the erroneous function, making defect location very difficult.

As a result of these problems, we wrote the *cgen* tool. A basic design goal was to minimize development time, so *cgen* was written in Perl [3], and it assumed a lot about the syntax of the C++ header file.

As new classes were added to the system, their implementations were generated by *cgen*. Also, when new member functions were added, their implementations were sometimes generated by *cgen*. Once we started using *cgen*, no further problems were encountered.

Afterwards, as an experiment, *cgen* was run on the header file, producing 97 files containing a total of about 8000 lines of C++. Since the developers wrote their files differently than *cgen*, it is not possible to compare the human implementations with those of *cgen*.

Another interesting result of not using *cgen* is that the designers are unwilling to make changes that affect many classes because of the bulk of code that would need to be changed. For example, adding a new member function to all the classes is easier when using *cgen*.

3.2 An incomplete example

Another example, not yet complete, is a 5000 line header file, which produces about 20,000 lines of generated C++. This is also for a language representation, the language being UDL/I, a hardware description language. Development was never completed, but a large part of a UDL/I compiler has been written merely by writing class declarations and a parser.

Since the semantic analysis routines tend not to fall into simple patterns, it is likely that they will all be written by hand. This will likely take up as much development time as the header file, and result is slightly less code.

The end result, therefore, will probably be less than 30,000 lines of code, most of which was generated by *cgen*. Instead of writing, inspecting, and testing 30,000 lines of code, we reduced our development and maintenance burden by 2/3.

3.3 The most recent example

The latest example is part of a toolkit for using the System Model Language (SML) for the DOMAIN Software Engineering Environment (DSEE). The SML represents the dependencies and build rules for a system, in the same manner as a makefile, although with a completely different syntax. (See [4] for more information about the SML and DSEE.)

We wrote a compiler for this language for internal use in maintaining our software. The language is parsed using standard tools, and a representation of the system model is built and written to a file. NIH-derived classes are defined to represent the system model, and NIH is used to read, write, and manipulate the objects.

Six classes are declared in the file, `model.h`, and one more in the file `any.h`, which total 460 non-comment lines of code. The generated C++ source files contain 556 non-comment lines of code, all generated by *cgen*. The API for the system also includes 946 non-comment lines of code written by hand. Thus, about 1/4 of the foundation was automatically generated and guaranteed to be correct.

During the course of development, the class declarations changed occasionally. Each time, *cgen* was automatically run to produce new C++ source files.

Most importantly, we never needed to worry about the correctness of the comparison, I/O, etc., routines. In previous experience with similar systems, an occasional typographical error was not uncommon. For example, a mismatch in the order between writing and reading data members happened in another system; it took many hours to trace the cause of the failure. Such defects cannot happen when using *cgen*.

Accurate effort measures were not kept for this project, so it is difficult to estimate how much time and effort was saved by using *cgen*. Since the kind of routines that *cgen* implements best are also the ones that are hardest to write correctly and the hardest to inspect and test, it is likely that we saved more than 1/4 of our total effort.

Of all the defects reported so far against the programs in this system, none were related to the functions implemented by *cgen*. One problem required a complete redesign of several classes; after the redesign, *cgen* simply rebuilt the required member functions, resulting in a rapid fix with high reliability.

3.4 A closer look

This section presents specific examples, taken from the DSEE SML compiler, described in the previous section. Excepts from the header files and the *cgen*-produced files are shown.

Figure 1 shows a simple class declaration, suitable for use by *cgen*. The **operator==** and *isEqual* functions are built into *cgen*, so their implementations are automatically built. The *print_label* function

is implemented by hand, and the user supplies a prototype implementation for all the *is_* functions, so *is_aggregate* is automatically built.

```
1 class Dsee_Aggregate : public Dsee_Block {
2    DECLARE_MEMBERS(Dsee_Aggregate);
3 public:
4    Dsee_Aggregate(NIH_String* str_p) : Dsee_Block(str_p) {}
5    Dsee_Aggregate(NIH_String& str)   : Dsee_Block(str)   {}
6    const bool operator==(const Dsee_Aggregate& obj)     const;
7    bool isEqual(const NIH_Object& obj)                  const;

8    virtual void print_label(ostream& out, int level,
9                             bool label_extern)          const;
10   virtual const bool is_aggregate()                    const;
11 };
```

Figure 1: A simple class example.

These function implementations are shown in Figure 2. Notice how the read constructor is produced by *cgen*; its declaration is implicit in the *DECLARE_MEMBERS* macro.

To produce the implementations of the *is_* functions, the author wrote a function generator for all of them. This must be written in Perl, following certain conventions. Figure 3 shows the function generator.

The function generator uses Perl to produce six almost-identical generators for each of the *is_* functions, namely, a generator for *is_any*, one for *is_block*, and so on. Each generator is quite simple: if the function name and the class name match, then a function to return *YES* is generated, otherwise the function returns *NO*. In other words, the *is_* functions implement simple *isA* predicates.

This particular form of *is_* functions is not built into *cgen* because different systems might use different methods. That is why the author must write the function generators.

4 Design and implementation

To minimize design, development, and maintenance effort, the *cgen* program is implemented in Perl [3]. It reads and parses the C++ header file, accumulating information in a set of arrays. When the end of a class declaration is reached, the information is correlated and the class implementation is generated.

Perl data types are rather simple, but they are quite useful for hiding details from the programmer. The only data types are numbers, strings, arrays, and associative arrays.

To simplify the program, its makes a number of assumptions about the C++ header file. Over time, the restrictions have been loosened slightly, but the ultimate goal is to implement a complete C++ parser, to eliminate all restrictions. The assumptions and restrictions themselves are explained in Section 5.2.

4.1 Information from the C++ source

The kinds of information kept for each class include:

- The data members: name, type and flags.
- Function members: name, type, arguments, and flags.
- Operator members: name, type, arguments, and flags.

```
1 #include <NIH/Object.h>
2 #include "model.h"

3 #define THIS Dsee_Aggregate
4 #define BASE Dsee_Block
5 #define BASE_CLASSES Dsee_Block::desc()
6 #define VIRTUAL_BASE_CLASSES
7 #define MEMBER_CLASSES

8 DEFINE_CLASS(Dsee_Aggregate, 1, "$Header$", NULL, NULL)

9 Dsee_Aggregate::Dsee_Aggregate(NIH_in& strm)
10 : Dsee_Block(strm), NIH_Object(strm)
11 {

12 }

13 const bool Dsee_Aggregate::operator==(const Dsee_Aggregate& obj)
14 const
15 {
16    if (! (*(Dsee_Block*)this == (Dsee_Block&)obj)) return NO;
17    return YES;
18 }
19 bool Dsee_Aggregate::isEqual(const NIH_Object& obj)
20 const
21 {
22    if (! obj.isKindOf(classDesc)) return NO;
23    return *this == castdown(obj);
24 }
25 const bool Dsee_Aggregate::is_aggregate()
26 const
27 {
28    return (YES);
29 }
```

Figure 2: Implementation of the *Dsee_Aggregate* class.

```
1 # Define the is_*() functions for the DSEE classes.
2 # See cgen for a description of how these Perl functions work.

3 # There is a virtual function for each class returning whether
4 # an instance is of the indicated class.  For example is_any()
5 # returns YES for instances of Dsee_Any, but returns NO for
6 # all other classes.

7 # Since every subroutine is just like the others, define a
8 # meta-definition subroutine, and use it for all the subroutines.

9 sub meta_define_is
10 {
11     local($name) = @_;

12     local($uc_name) = $name;
13     substr($uc_name, 0, 1) =~ tr/a-z/A-Z/;

14     eval <<EOF || die "$0: $@\n";
15     sub define_is_$name
16     {
17         local(\$c) = \@_;
18         '   return ('   .
19             (\$c eq 'Dsee_$uc_name' ? 'YES' : 'NO')   .
20     ');';
21     }
22     1;
23 EOF
24 }

25 &meta_define_is('aggregate');
26 &meta_define_is('any');
27 &meta_define_is('default');
28 &meta_define_is('element');
29 &meta_define_is('external');
30 &meta_define_is('model');

31 1;
```

Figure 3: Generator for the *is_* functions.

- Type conversion members: name, type, arguments, and flags.
- Constructors: name, arguments, and flags.
- Destructors: name and flags.

Also, if a one-line comment follows the member declaration, that information is kept, to be propagated to the output file. The *flags* include information about the particular declaration, i.e., a constant, a virtual function, the storage class, access (public, protected, or private), and whether a function member is pure virtual. Although functions, operators, and type conversions are all just functions in C++, the *cgen* parser recognizes them as distinct members and handles them separately.

All the base classes of each class are tracked, including the access and whether the base class is virtual. To keep track of the class hierarchy requires just one associative array: the index is the class name, and the value is a list of base class names.

4.2 Information for NIH

To support the NIH run-time typing mechanism, a class descriptor is created for each class. This descriptor contains information about a class's base classes, virtual base classes, and data members that are classes. The *cgen* script allows data members to be pointers to NIH classes, but not to any other types.

4.3 Generating an implementation

The real work of *cgen* is producing member functions. If the interface includes overloaded member functions with the same name as a data member (removing the leading *d_*), and with the same type, those functions are implemented to set or get the data member.

If the interface includes member functions that match one of the patterns known by *cgen*, those functions are implemented.

A Perl subroutine implements a member function, using the following information:

- The name of the class.
- The name and return type of the function.
- The name and type of each argument to the function.
- The storage class and other flags for the function.

The *cgen* program knows not to create implementations for pure virtual functions and functions with inline definitions.

If the interface contains a constructor whose arguments have types that match the types of the data members, then the constructor is implemented by initializing those members with the corresponding arguments. The *cgen* program handles inheritance, including multiple inheritance. When implementing a constructor, *cgen* correctly builds all the virtual base classes.

Each member function implemented by *cgen* has a corresponding Perl subroutine to implement that function. Although it is possible to extend *cgen* by adding new subroutines, in practice, this is difficult. For example, Figure 4 shows the Perl subroutine that implements *printOn*. This function is used to print a human-readable version of the object.

The *printOn* function takes two arguments: *strm* is the output stream to print on, and *level* is an indentation level, to make nested objects easier to understand.

The argument names and types are passed to the Perl subroutine, which verifies the arguments. If the user has declared types without names, *cgen* must make up a name (lines 7–18).

```
1 # void printOn(ostream& strm, int level = 0);
2 sub define_printOn
3 {
4     local($c, $strm, $level) = @_;

5     # Get the formal argument names and types.
6     local($stype, $sarg) = split("\t", $strm, 2);
7     local($ltype, $larg) = split("\t", $level, 2);

8     # See if user supplied names; if not create them and warn the user.
9     if ($sarg eq '') {
10        $sarg = 'strm';
11        $a[$[] = "$stype\t$sarg";
12        &warn($linenum[$n], "using ", $sarg,
13      " as first formal argument to printOn()");
14    }
15    if ($larg eq '') {
16        $larg = 'level';
17        $a[$[+1] = "$ltype\t$larg";
18        &warn($linenum[$n], "using ", $larg,
19      " as second formal argument to printOn()");
20    }

21    # Print the base classes, unless printOn is pure virtual.
22    local($d, $b, @lines);
23    foreach $b (split($;, $classes{$c})) {
24        $b =~ s/\b(virtual|public|private) //g;
25        next if $b eq 'NIH_Object';
26        next if defined $pure{"$b::printOn"};
27        push(@lines, "  $b::printOn($sarg, $larg);");
28    }

29    # Print the data members by calling printOn or using <<.
30    foreach $d (@data_members) {
31        push(@lines, "  $sarg << setw($larg) << ' ';");
32        # A reference is the same as a non-reference.
33        ($type = $type[$d]) =~ s/&$//;
34        if ($type =~ /\*$/) {
35            push(@lines, "  if ($name[$d]==NULL)");
36            push(@lines, "    $sarg << \"null\\n\";");
37            push(@lines, "  else");
38            push(@lines, "    $name[$d]->printOn($sarg, $larg+1);");
39        } elsif (defined $classes{$type}) {
40            push(@lines, "  $name[$d].printOn($sarg, $larg+1);");
41        } elsif (defined $scalar{$type}) {
42            push(@lines, "  $sarg << $name[$d] << '\\n';");
43        } else {
44            &error($linenum[$d], "cannot print $c::$name[$d]");
45            push(@lines, "  // $sarg << $name[$d] << '\\n';");
46        }
47    }
48    join("\n", @lines);
49 }
```

Figure 4: Perl subroutine to implement *printOn* for *cgen*.

The next part (lines 21–28) of the subroutine arranges for printing the bases classes. The list of base classes is obtained, and extraneous information is removed. The subroutine produces code to call the *printOn* function for each base class, unless the function is pure virtual in that class.

The last part (lines 29–47) arranges for printing the data members. The data member is indented, and pointers to or instances of other NIH classes are printed by recursively calling their *printOn* functions. Scalars are printed normally, and unknown types produce an error from *cgen*. In the latter case, a comment is put into the output file, showing a guess as to how the output statement might be written.

The Perl subroutine could also examine the flags for each data member to avoid printing the static ones, for example.

The result of the subroutine (line 48) is a single string that represents the body of the *printOn* function. Indentation is used in the output string to help keep the resulting C++ easy to read.

5 Using Cgen

The *cgen* program reads a C++ header file and writes C++ source files. As mentioned earlier, it is written in Perl, which limits its ability to parse all of C++, so a restricted subset is used.

A separate C++ source file is created for each class. The file contains the implementations of the member functions declared in the header file for that class.

5.1 Command line options

Below is a summary of the command line options recognized by *cgen*.

cgen [**–usage**] [**–help**] [**–version**] [**–qw**] [**–l** *file*] [**–e** *extension*] [**–c** *class,...*] [**–t** *type,...*] [*file...*]

If no files are named on the command line, then the standard input is read.

–u[sage] Type a one-line usage summary, and exit.

–h[elp] Type help information, and exit.

–v[ersion] Type the version of *cgen*, and exit.

–q quiet: do not echo implementation file names as they are created. The default is to type the name of each file as it is created.

–w do not type warning messages. The default is to type warnings.

–l*file* Load Perl definitions from the named file. These definitions can be used to implement member functions and operators not otherwise handled by *cgen*. Great care must be taken with this option because the user can inadvertently cause *cgen* to do terrible things.

–e *extension* specifies a different file name extension to use for implementation files. The default extension is '.C'.

–c *class,...* specifies class names to implement. The class names are separated by commas or white space. Without this option, every class declared in the interface files is implemented.

–t *type,...* specifies additional type names. The type names are separated by commas or white space. C++ is a context-sensitive language, and it is important to know whether an identifier is a type name or not when parsing. Since *cgen* does not do a complete job of parsing, the user can help it by listing type names not otherwise recognized by *cgen*.

5.2 Parser restrictions

For *cgen* to parse its input successfully, a number of restrictions must be obeyed.

- Nested class declarations are not recognized.
- Data members must have names starting with *d_*, and be all lower case.
- Data members must not be hidden in parentheses. This means function pointers must be declared using **typedef**s.
- At most one member may be declared on a line of text.
- Character strings that look like declarations can fool the parser.
- A one-line comment about a member can follow it on its line. This comment is duplicated in the implementation.
- Complicated type specifiers, especially those involving functions, can confuse the parser. The use of complex constructs must be restricted to **typedef**s.
- Multiple line declarations, such as enumerations with many items, can fool the parser. The parser works best when the opening curly brace is at the end of the first declaration line, and the closing curly brace is on a line by itself.
- Inline functions can fool the parser. They are best written using the **inline** specifier, with an definition after the class declaration. If the function body must be put in the class declaration, the user must make sure the opening curly brace ends the first declaration line, and the closing curly brace is on a line by itself.
- A line that ends with a comma is assumed to continue on the next line. This is useful for inline constructors with many base classes and data member initializers, or for functions with many arguments.
- The C preprocessor is not run because *cgen* needs to be able to locate and recognize the NIH macros, such as *DECLARE_MEMBERS*. The user, therefore, cannot use the C preprocessor before running *cgen*.

6 NIH Classes

The NIH classes are a set of C++ classes implementing some basic, useful collections, vectors, strings, and similar classes. The fundamental organization of these classes is based on Smalltalk [1].

6.1 Local changes

At Mentor Graphics, we use a subset of the full NIH suite, including the implementations that are portable, and omitting the multi-tasking and scheduling classes, semaphores, and so on.

Since other sets of classes are used at Mentor Graphics, the names of all the global identifiers in NIH had to be changed to begin with *NIH_*. Otherwise, there would be conflict between different classes with identical names, such as *String*. The *cgen* program was designed to work closely with the NIH classes specifically, with class names built into the program.

6.2 Objects

Every class in NIH is derived from the class, *NIH_Object*. This abstract class defines the interface to several pure virtual functions. NIH also supports run-time type identification and safely casting down from a base class to a derived class.

The NIH classes contain support for I/O of any class derived from *NIH_Object*, copying objects, comparing objects, and so on.

Any object, that is, any instance of a class derived from *NIH_Object*, can be stored in any collection. Two member functions are required: *hash* and *isEqual*. Objects that are equal must also have the same hash values.

There are two levels of copy: *shallow* and *deep*. A shallow copy is a memberwise copy. A deep copy is a memberwise copy, following all members recursively, making deep copies of all members.

The following sections describe these functions in greater detail.

6.2.1 hash

A hash value is an unsigned integer. Any two objects that are equal must also have equal hash values, but two objects with the same hash values are not necessarily equal. The *cgen* program implements the *hash* function with an exclusive OR of the scalar data members with the hash values from the NIH data members and the base classes.

6.2.2 isEqual

Any two objects that are equal must also have equal hash values, but two objects with the same hash values are not necessarily equal. Since *isEqual* is defined in the base class, *NIH_Object*, its argument is of type *NIH_Object* (in particular, a constant reference). When defining *isEqual* in a derived class, the function must ensure that the comparand is of the same type. The *cgen* implementation performs a deep comparison, recursively calling *isEqual* for base classes and data members of an NIH-derived type.

6.2.3 printOn

Printing is for presenting an object to the user. There are no restrictions on the format. The *cgen* program implements *printOn* by recursively printing the base classes and the data members of NIH-derived type and the non-NIH data members. There is a similar function, *dumpOn*, which differs from *printOn* by having the names of the data members printed, too. The *dumpOn* function is often more useful for debugging.

6.2.4 deepenShallowCopy

To make a deep copy, NIH first performs a shallow copy, and then calls *deepenShallowCopy*, to replace pointers with deep copies, and to deepen copies of reference members and class instance members.

6.2.5 storer

NIH defines I/O mechanisms. The user must implement the functions to read and write the class's members. The *cgen* program takes care of writing base classes and virtual base classes in the correct order, in addition to storing non-static data members.

6.2.6 Read constructor

NIH defines I/O mechanisms. The user must implement the functions to read and write the class's members. The *cgen* program sets up the initializers for the base classes and all the virtual base classes, and reads the data members in the same order in which they are stored.

6.3 Derived classes

To derive a class from an NIH class, certain formalities must be followed. The base class or classes must be identified, as well as the virtual base classes. Data members that are class instances must also be noted. The pure virtual functions must be clearly declared.

NIH provides macros to facilitate the declaration of NIH-derived classes. Since NIH uses a run-time type mechanism, it needs help from the user in defining the attributes of a class.

6.3.1 DECLARE_MEMBERS

In a class declaration, some standard member function must be declared for any NIH-derived class. The *DECLARE_MEMBERS* macro declares these functions. The *cgen* program looks for this macro to determine whether a class is derived in the NIH scheme.

If the macro is not used, then *cgen* ignores the class, unless it is derived from a class that *cgen* recognizes as being an NIH class, in which case the user is warned. Additionally, if the class is not derived from NIH classes, yet *DECLARE_MEMBERS* is used, the user is warned.

6.3.2 DEFINE_CLASS

To implement the members declared by the macro, *DECLARE_MEMBERS*, the *cgen* program sets up some macros to define the base classes, virtual base classes, and data members that are classes, and uses the *DEFINE_CLASS* macro to implement the standard member functions.

6.4 Why NIH?

The Smalltalk model of class hierarchy is not always considered desirable for C++ programming. In particular, the use of a single base class, *NIH_Object*, from which all other classes derive results in a deep, complicated hierarchy of classes. Using templates to implement container classes is often easier to understand, use, and maintain.

One aspect of using a common base class is that all classes inherit all the functions, even if they do not make sense. For example, there is a *compare* function that returns -1, 0, or $+1$ if an argument is less than, equal to, or greater than an object. Some classes do not have any meaningful ordering, so this function is currently implemented as a run-time error for those classes. If there were no common base class, there would be no need to declare such a function, making the class definition simpler and easier to understand.

On the other hand, using a common base class, with common member functions, permits the use of tools like *cgen*. If we were to use templates for container classes, it would be harder to implement a *cgen*-like tool that would automatically implement functions for common actions, such as Input/Output, comparison, and so on. By using a common base class, *cgen* has built-in knowledge about that class and about the member functions that are inherited by all other classes.

Another reason to use the NIH classes is that C++ did not have templates when our work began. Without templates, the *NIH_Object*-based hierarchy was the easiest way to design the kinds of systems that we needed, as described in Section 3. Now that we have close to one hundred thousand lines of code written using the NIH classes, it is more difficult to change than to continue using the deep hierarchy.

7 Conclusions

The *cgen* tool operates in a restrictive environment. Within that environment, however, it serves admirably. The *cgen* program has been used successfully on different projects, and when compared to projects that did not use *cgen*, there is a noticeable improvement in productivity and the reliability of the end-product.

Since *cgen* is implemented in Perl, and is extended in Perl, it is not always easy to write new function generators. It would be preferable to have a cleaner way to write function generators, and the future of *cgen* certainly lies in that direction. Another aspect of its rapid implementation is that *cgen* does not recognize the full C++ language, but parses its input based on simple patterns. A full parser is easy to write, but changing the rest of *cgen* from Perl to another language would probably involve a lot of work.

Another limitation of *cgen* is that it is tied to the Mentor Graphics' modified implementation of the NIH classes, and assumes Mentor Graphics' coding standards. When a complete C++ parser has been implemented, the latter restrictions can be lifted. The problem of removing the NIH dependencies is harder, without an easier way to specify the function implementation prototypes.

References

[1] Adele Goldberg and David Robson. *Smalltalk-80: The language.* Reading, Mass.: Addison-Wesley, 1989. ISBN 0-201-13688-0.

[2] Keith Gorlen, Sandy Orlow, and Perry Plexico. *Data Abstraction and Object-Oriented Programming in C++.* Wiley, 1990. ISBN 0-0471-92346-X.

[3] Larry Wall and Randal L. Schwartz. *Programming Perl.* Sebastopol, CA: O'Reilly, 1990. ISBN 0-937175-64-1.

[4] *Getting started with the Domain Software Engineering Environment (DSEE).* Chelmsford, MA: Apollo Computer, 1988.

Designing-In Quality for Large C++ Projects

John S. Lakos

Mentor Graphics Corporation

Wednesday, October 21, 1992

Quality is an engineering responsibility; it must be actively sought from the start. Quality is something which simply cannot be added in after a project is largely complete. This paper will address engineering aspects of designing-in quality for large C++ projects, including *levelization*, *insulation*, interface design, effective programming, performance, and hierarchical testing.

1.0 Introduction

FIGURE 1. Designing-In Quality

GDT 6.0 is the next major release of the Generator Development Tools, developed in the IC Group at Mentor Graphics Corporation. GDT is a suite of Computer-Aided Design (CAD) tools to aid in the design and development of state-of-the-art integrated circuits. Considering the extensive functionality to be added to the product in this release, the decision was made to rewrite much of the existing (C Language) code and to write all new code in C++. My current responsibility is to oversee and actively ensure the engineering quality of this product.

FIGURE 2. Advantages Of C++ Over C

The advantages of the object-oriented paradigm in managing the complexity of large systems is becoming more well known every day. The number of C++ programmers is doubling every 7-9 months.

Once one has made the mental leap from thinking exclusively in a procedural paradigm, large software systems can be more readily comprehended. Because they are easier to understand, these systems are easier to use and maintain. Packaging the functionality with the data provides for more modular code; hence, it is easier to transplant and reuse existing software.

The strong typing of C++ affords more error checking at compile time. The layers of interfaces encouraged by classes facilitate the evolution and extensibility of a system. These same interface facilitate both incremental and hierarchical testing of the system. By creating well defined interfaces, it is possible to objectively analyze a running system, determine bottlenecks and reimplement the 20% of the software which consumes 80% of the running time.

FIGURE 3. Disadvantages Of C++ Over C

While C++ can, at first, be somewhat overwhelming for C programmers to comprehend, it does not take too long for a competent C programmer to get a non-trivial C++ pro-

gram up and running. Unfortunately, the undisciplined techniques to create small programs in C++ are totally inadequate for tackling larger projects. That is, naive application of C++ technology does not scale up well to larger projects. The consequences for the uninitiated are many.

C++ is harder than C. There are innumerable ways to shoot yourself in the foot. Often you won't realize a fatal error until it is much too late.

By passing user-defined types by value, it is easy to write a perfectly correct C++ program that runs ten times slower than it would have had you written it in C. By ignoring compiler dependencies, it is possible to cause every compilation unit in the system to include every header file in the system, reducing compilation speed to a crawl.

By ignoring linker dependencies, over zealous class developers have created sophisticated classes which indirectly depend on enormous amounts of code. The executable for a "Hello World" program employing one particularly elaborate `String` class produced an executable image size of over a megabyte!

Cyclic dependencies among most all modules will occur if not carefully prevented, as will compile-time dependencies. It is by eliminating these dependencies that the more obvious symptoms will disappear.

A poorly written C++ program can be very hard to understand and maintain. If interfaces are not encapsulating, it will be very difficult to tune or enhance implementations. Poor encapsulation and modularity will hinder reuse. The advantage in testability will be eliminated.

Because the user community of C++ is doubling about every 8 months, it follows that half of the C++ programmers out there have been programming in C++ for less than 8 months.

FIGURE 4. FOCUS: QUALITY *ENSURANCE*

Quality Assurance (QA) is typically responsible for "assuring" quality. The problem is that QA does not typically get involved until late in the development process (after the damage is already done). Typically QA does not influence the design of the product. QA is rarely involved in low-level engineering design decisions. Typically, any testing QA does is at the end-user level, relying on engineering for any low-level regression testing.

Engineering has the responsibility for "ensuring" quality. That is, the quality must already be there for QA and test engineers to find it. It cannot be QA'ed in after the design is nearly complete. Engineering must take the lead in designing-in the quality. Designing-in quality is expensive, failing to do so can result in huge, unmaintainable, unusable programs (and failed projects).

FIGURE 5. "No Silver Bullet"

There is no quick and easy way to design-in quality. It takes experience, intelligence, and often "good taste". Tools alone cannot do the job for you. Testing is an expensive part of development and must be given appropriate weight in schedules (about 25%). Testing provides an objective measure of how well your program does what you say it does. As importantly, testing also drags you through the process of saying everything your program actually does. Creating an acyclic architecture, fleshing out consistent and minimal interfaces, and minimizing dependencies all take time. Designing for test must also be given appropriate weight in schedules.

FIGURE 6. Designing-in Quality... When do we start?

Designing-in quality must be considered right from the start, before writing code, before considering detailed interfaces, even before the high-level architecture is established. Only in this way can fundamental quality problems be avoided.

2.0 Abstractions

FIGURE 7. Example: A Stack Abstraction

The figure shows a simple `Stack` class and a `StackIterator` class in a single abstraction "`stack`". The *logical* view of this abstraction consists of these two classes with the `StackIterator` using the `Stack` in its interface (open bubble with two lines). The *physical* view shows the abstraction decomposed into two files: the interface or header file (`stack.h`) and the implementation file (`stack.c`).

Throughout this presentation, boxes with rounded corners represent logical entities such as classes and structs, while boxes with angular corners represent physical entities such as files. An abstraction itself is considered a physical entity and is also represented as a box with angular corners.

In this example the `stack` abstraction has a fairly complete interface. The first four functions of the `Stack` class are the basic functions required of all types; namely: a constructor, a copy constructor, an assignment operator, and a destructor. The next four functions are specific to stacks (e.g., `push`, `pop`, `top`, and `isEmpty`).

The `StackIterator` is able to construct itself from a `Stack`, advance to the next stack member, test for end, and extract the current entry from the `Stack`. The copy constructor, assignment operator, and destructor generated by default are adequate. This `StackIterator` is designed to be used in a "for-loop" such as:

```
for (StackIterator it(stack); it; ++it) {
   cerr << it() << endl;
}
```

Of course `StackIterator` must be a friend of the stack. (The friend declaration is not shown.)

FIGURE 8. The Abstraction Level

An abstraction is the smallest physical entity in a system. Typically, it has a single ".h" file and a single ".c" file. It is quite helpful (to tools and people) if the root names match. The name of the abstraction is the root name (e.g., `set`). Notice that (by convention) type names (classes, structs, unions, typedefs, and enumerations) begin with an uppercase first letter. Abstraction names (like file names) are all lower case.

An abstraction represents a single concept which is typically implemented as one or more closely related classes. By requiring all external names be declared in the ".h" file, we remove back doors from the system; the interface is then accurately represented in the header file. The use of external declarations to gain access to unpublished global names is strongly discouraged. To ensure correctness, the ".c" file must include the corresponding ".h" file.

While cyclic dependencies among physical entities is discouraged, there are often good reasons for having mutual dependencies among classes within a single abstraction. The logical view of the `set` abstraction shows that `SetIterator` uses `Set` in its interface (open bubble with two lines) and that `Set` uses `SetIterator` in its implementation (solid bubble with two lines).

FIGURE 9. Logical Relationships (IsA and Uses)

Here we can see all three logical relationships in a single design. The string class uses an array-of-character (template) class in its implementation, but the user of the `String` class has no programmatic access to that information. The `Array<char>` class is an encapsulated implementation detail.

A `Word` is a kind of `String`. A `Word` can be used whereever a `String` is required. In addition, `Word` has other properties that `String` does not have (perhaps only additional functionality). The `wordlist` abstraction contains two classes. The `Link<Word>` class is an encapsulated implementation detail of the `List<Word>` class (but since `Link<Word>` is not a private nested class, others could use `Link<Word>` if they so desired). Both `Link<Word>` and `List<Word>` use `Word` in their interfaces. `Alias` is an interesting class; it both is a `Word` and has a `Word` (the `Word` for which this `Word` is an alias). Since you can get at this `Word` through the interface of `Alias`, `Alias` uses `Word` in its interface.

Each of the above dependencies are *logical* dependencies among classes. These *logical* dependencies will imply *physical* dependencies among abstractions.

FIGURE 10. Physical Relationship: DependsOn

In this example, the most basic abstraction is the character array (`chararray`) because it has no dependencies on other abstractions. Next we notice that the class `String` uses the class `Array<char>` so the `string` abstraction depends on the `chararray` abstraction. In other words, we could create an instance of `String` as long as we linked in the object code for `chararray` (`chararray.o`). We need not link in any of the other object files in order to test `string`.

Now, `Word` (which is a `String`) clearly depends on `String`. We will have to `#include "string.h"` in order even to create an instance of `Word`. We could test the

`word` abstraction provided we also had access to `string` and (of course) `chararray`.

The abstractions `alias` and `wordlist` both depend on `word`, but not on each other. We could test either of these abstractions without the other provided we had our hands on the other three.

The numbers in the lower left corners indicate the depth (beginning with 1) of this abstraction in the abstraction dependency graph. It is referred to as the "level" of the abstraction.

3.0 Hierarchical Testing

FIGURE 11. Design For Test

A major component of designed-in quality is design for testability (DFT). The importance of DFT is well recognized in the Integrated Circuit (IC) Industry. In many cases, it is impractical to test chips (with over a million transistors) from the outside pins alone. Hence, extra internal test circuitry is often provided to partition the chip into its subsystems. By granting the tester direct access to a subsystem, the cost of propagating a signal from the chip's inputs to those of the subsystem and sensitizing a path back to an observable output is eliminated. In this way, the full functionality of the subsystem can be explored efficiently.

Like IC design, object-oriented software design involves the creation of a relatively small number of types, which are then instantiated repeatedly to form a working system. For example, a primitive object in many software systems is a `String` class. Testing the functionality of the `String` class is easiest and most effective if done directly, rather than attempting to test it as part of a larger system. And, unlike IC testing, we have direct access to the software subsystems (e.g., the `String` class).

FIGURE 12. Avoid Cycles In The Dependency Graph

A collection of abstractions with an acyclic dependency graph is said to be "*levelizable*". The notion of "`level`" is borrowed from the field of digital, gate-level circuit simulation. The level number indicates the longest path from the gate to a primary input. For example, level-1 gates are fed only by primary inputs; these gates are evaluated first (in arbitrary order). Next to be evaluated are all level-2 gates, fed only by one or more level-1 gates and zero or more primary inputs. The subset of gates with the highest level are evaluated last. The property that makes a circuit levelizable is that it has no feedback. This is exactly the property we would like our software designs to possess.

The group of abstractions on the left of the figure illustrates what happens when A1 knows about A2 and visa versa. It is not possible to reuse or test either A1 or A2 without linking in the other. The system on the right is levelizable. A1 can be tested and used in isolation. A2 is still tested in the presence of a (tested) A1. A3 requires both A1 and A2 (which are now both tested). In the latter case, a user would be free to grab A1 without having to pull in the code for A2. In general, the less you depend on, the more reusable you are.

FIGURE 13. Hierarchical Testing Of Abstractions

If designs formed a binary tree of abstractions, half of the abstractions in the system would be level-1 and could be tested in isolation. Another quarter would depend on only (two of) these leaf abstractions. While real designs are not nearly so regular, the advantage of testing a hierarchy of interfaces remains.

The ability to test software incrementally and hierarchically is a direct result of DFT. Low-level abstractions (such as strings) are easily tested in isolation without much effort; but at higher levels, care must be taken to ensure that each of the subcomponents of the system can be tested with a minimal dependence on the rest of the system. This progressive minimal interdependence will be facilitated if the dependencies of the components of the system generate a Directed Acyclic Graph (DAG).

Our goal is to be able to build a test driver for each abstraction that involves the abstraction to be tested and only the (few) abstractions on which it depends. This ability not only implies that we can test the subsystem in isolation but that we can reuse it in other applications as well.

FIGURE 14. Integration Testing: Testing The Interaction

Unlike IC testing where every object is unique, if we can test an instance of a class (e.g., `Stack`) in isolation and it works, they'll all work.

As we move to higher levels of the design we need not retest the `stack` at every level. Rather, we must test the *interaction* of the low-level abstractions which form the higher-level abstraction.

The arrows in the figure depict the flow of a computation as it passes out of the interface of one sub-object and into another sub-object during the testing of the encompassing object of type "E". We can assume that objects of types "A", "B", "C", and "D" have been tested and are internally as sound as `int` and `double`.

FIGURE 15. Testable Designs Are Levelizable

Both IC designs and object-oriented software designs are hierarchical in nature. Subsystems are composed to form larger systems. Once thoroughly tested in isolation, the subsystem itself can be assumed correct. Additional testing at the next *level* will focus on verifying the correct *use* of the subsystem, and not the *internal* correctness of the subsystem itself.

Thorough regression testing is a costly development process that deserves much consideration. On the order of 25% of development time might be devoted to creating high-quality regression tests. Creating such tests prematurely runs the risk of wasted effort since interfaces often change and evolve during the early stages of development.

Typically a developer will create a small main program to try out the functionality of a class as it is being implemented. The cost of recompiling and relinking with each new test is relatively small since the class is in the development phase and in flux.

Most often, the driver executes a fixed test. A more sophisticated driver might allow the user to alter the parameters of the test without having to recompile. A still more general driver that permits interpretive access to all the functionality of and abstraction is called an *"exerciser"* for that abstraction.

In simple terms, an exerciser is an interpreter that successively reads input strings from the terminal (or a file) that specify what function to call and with what arguments. The exerciser looks up the function to be called and calls it, passing the indicated arguments. The value returned (if any) is stored in a symbol table for future reference.

Once class interfaces have sufficiently stabilized, the focus is more one of Quality Assurance. It is no longer expected that the abstractions will have to recompile and relink with each test. As a result, the ability to write interactive tests has proven extremely powerful.

4.0 Levelization

FIGURE 16. Levelization Example

This example illustrates that the exact behavior need not be understood in order to analyze a design. Each abstraction in this design contains a single class. The relations depicted are "IsA" and "Uses (in the interface)".

The question is, "does the dependency graph implied by this design have cycles?"

FIGURE 17. Levelized Diagram

The answer to the previous question is, "No."

The `name` abstraction depends on nothing else, so `name` is at level 1. `Word` uses `Name` and nothing else so `word` is at level 2. The `node` abstraction is also at level 2 (for the same reasons). `WordList` uses `Word`, hence `word` is level 3. Both `File` and `Directory` are kinds of `Node`. In addition `Directory` uses `Node` in its interface. Both `file` and `directory` depend on `node` (but not on each other); both are level 3. `WordexBuilder` uses both `Directory` and `File`, but since both are level 3, `wordexbuilder` is at level 4. `Wordex` uses both `WordList` and `Wordex-Builder`. The abstraction holding `WordList` is at level 3 but that of `WordexBuilder` is at level 4. The highest level dominates, and `wordex` is at level 5.

FIGURE 18. Abstraction Dependency Diagram

Reformatting the diagram often makes the structure easier to understand, maintain and test. *DependsOn* is a transitive property of a design. Redundant transitive arcs may be removed to reduce the clutter of the diagram (e.g., the direct dependency of `wordexbuilder` on `node`). Tools exist which automate the process of detecting design cycles and producing diagrams such as this one. Such tools are invaluable for large projects.

FIGURE 19. Levelizability is *Not* Automatic

Not all designs are levelizable. In this design we have two similar types, `Box` and `Rectangle`, in separate abstractions; each knows how to construct itself from the other. There is obviously a cyclic dependency which would not even compile without the use of proper include guards.

FIGURE 20. Forward Declarations Do Not Solve Problem

The problem is not just in the fact that the `#include` statement is in the header file. Each implementation file must still include the other header file (as well as its own). It will not be possible to use or test one of these abstractions without linking in the other.

FIGURE 21. Solution #1

This solution requires that we change our point of view somewhat. Instead of having both `Box` and `Rectangle` know about each other, we will make `Box` the primitive class and move both conversion functions into `rectangle`. `Rectangle` now uses `Box` but not visa versa. In this case `box` is at level 1 and `rectangle` is at level 2.

FIGURE 22. Solution #2

This solution is almost identical to the previous one except that we have made `Rectangle` the primitive class and pushed both conversion functions into `Box`. Now, `rectangle` is at level 1 and `box` is at level 2.

Note that there was no apparent need to have the `#include "rectangle"` statement in `box.h`. Clients of box will have to link in `rectangle.o` but should not have to reparse `rectangle.h` if they are not using rectangles. This technique also *insulates* clients of `Box` from having to recompile due to changes made to the `rectangle.h` file. We will talk more about *insulation* later in this presentation.

FIGURE 23. Solution #3

This solution is again symmetric. It removes the conversion routines from both abstractions leaving them both at level 1. Now both `box` and `rectangle` can be (re)used and tested independently. The new `convert` abstraction is at level 2. Only those clients using both rectangles and boxes who wish to convert from one to the other need include it. This solution is preferred over the previous two because it allows the most code to be tested/(re)used in isolation.

FIGURE 24. Example: A Shape Editor

This example illustrates a common situation. The system consists of a number of subcomponents used in the implementation of an editor. Each subcomponent uses `Shape` in its interface. One of the modules (say E_1) needs to build any particular (derived class of) `Shape` from a character string. Anticipating this need, the mechanism to do so is made a static member of the base class `Shape`.

This is a poor decision because it forces a "uses (in the implementation)" relationship of the base class (`Shape`) on all of its derived classes. No one sub-object of the editor can be tested without including all shapes. Worse, the design is not extensible. Adding a derived class forces the base class to be modified, which in turn impacts every other abstraction in the system.

FIGURE 25. Shape Editor: A Better Design

The solution is simple, create a separate abstraction for creating specific shapes. All but E_1 can be tested with one or two *concrete* kinds of `Shape`. The impact of adding another *concrete* kind of `Shape` (e.g., `Trapezoid`) is greatly reduced.

FIGURE 26. Graphs: Nodes and Edges

This somewhat contrived example is intended to illustrate even closely related abstractions are separable. In this example, `Edge` uses on `Node` in its interface, while `Node` uses `Edge` only in its implementation. The intent is that it should be possible to determine if two instances of `Node` are connected using the `isConnected()` member of the `Node` class. As it stands, it is impossible to create either a `Node` or and `Edge` without linking in both abstractions.

FIGURE 27. Graphs: Nodes and Edges (Break The Cycle)

A general solution to this type of problem is illustrated at the lower right of this figure. A *protocol* `Node` class replaces the original *concrete* `node` class which is renamed `RealNode`. The `Edge` class remains unmodified.

The dependency of `Node` on `Edge` is now only in name – neither `node.h` nor `node.c` includes the `edge.h` file.

It is possible to include node.h in a client module without including `edge.h` or linking in `edge.o`. Clients which did not need to create particular (concrete) nodes could still make use of the protocol supplied by `Node` to determine if two instances of type `Node` were connected.

Instances of `Edge` could be constructed and (partially) tested in isolation. Features of `edge` that did not involve `node` could be exercised. Testing those features that did depend on `node` would require an instance of a concrete node (such as `RealNode`).

How do you test an abstract class (i.e., one that cannot be instantiated such as `Node`)?. The answer is that you cannot, at least not without the help of one or more sample derived classes. Authors of protocol classes should supply at least one representative for the protocol class to be used in verifying other abstractions which depend on this protocol. In many cases, several such representatives will be necessary to obtain full test coverage. Anticipating such testing needs is an important part of DFT.

Levelization is something to strive for, but is not law. There are some cases where mutually depended abstractions make sense. Had both `node` and `edge` depended on each other in their interface in an inseparable way, the only alternative would have been to place them in the same abstraction. Sometimes the "best" design may have a few small pockets of unlevelizable abstractions.

5.0 Packages

FIGURE 28. The Big Picture

In large projects, "societies" of cohesive abstractions are further grouped into larger units we call "packages". The physical layout of a package is a directory of ".c" and ".h" file pairs. There is no physical dependency expressed in the directory structure; all packages live at the same physical level of the file system. The package dependencies are dictated by the dependencies among abstractions comprise the packages.

When a package is to be released (for global consumption) the subset of the header files which form the packages exported interface are copied to a global release directory. Note many of package's abstractions may be needed only locally and are not released. All ".o" files for the package are grouped into a single archive file (".a") and placed in the public "lib" area.

FIGURE 29. High-Level Architecture Design

It is even more important at the package level to avoid mutual dependencies. The ability to unbundle a large software system and sell these parts stand-alone is just one good reason to avoid mutual dependencies. Linking can become difficult if there is no predefined order for the libraries. Libraries may need to be searched more than once as symbols in one are resolved in the next. Repeating libraries on a link line two or three times is slow and undesirable. As code changes, so may the order of the required library sequence. Trying to come up with the right link command may become an experimental, iterative process.

Another is that interdependencies among packages often involve interdependencies among people working on different floors or even distinct geographical locations.

FIGURE 30. The Package Level

The package is the fundamental course building block of the system architecture. It is made available via a directory of ".h" files and a library of ".o" files. Each package has a collection of responsibilities and dependencies on other packages. These dependencies should be minimal, acyclic, and explicitly specified by the architects. In particular, the architects should specify the kinds of dependencies which are to occur (e.g., *interface* vs. *implementation* and *size/layout* vs. *in-name-only*). The larger the project, the more critical these dependencies become.

6.0 INSULATION

FIGURE 31. Example: Encapsulation vs. Insulation.

In this example there is only a subtle difference between the classes `Foo` and `John`. `Foo` "*hasA*" `Bar` as a data member while `John` "*holdsA*" reference to a `Bar`. This distinction allows the header file for the abstraction `john` to forward declare the class `Bar` rather than including `bar`'s header file. In doing so, `john` *insulates* its clients from

changes to `bar`'s header file including `Bar`'s private data members. Reimplementing `bar` will force all of Foo's clients to recompile, but not `John`'s.

FIGURE 32. Encapsulation vs. Insulation

A common synonym for encapsulation is "data hiding" or more generally "information hiding". Encapsulation means hiding the implementation details of your abstraction, but in a programmatic sense. Even though clients of an abstraction cannot get at the internal implementation of class, the compiler will often need to know all about the internals of the class in order to compile the client. Encapsulation is important because it defines the problem you are solving without clouding the interface with details arising from the implementation. A good technique for ensuring encapsulation is to come up with at least two different ways of implementing your interface.

Encapsulation is a *logical* property of a design. The physical analog is *"insulation"*. *"Insulation"* means hiding the implementation details even from the compiler. All references to the implementation classes in your header file are *in-name-only.* All of these classes are *forward declared* instead of including their header files in your header file. Changes to these classes will not force clients of your class to recompile.

FIGURE 33. Problem: Shapes (Uninsulated)

This example is similar in nature to the "Shape Editor" example. A number of clients are using the `Shape` base class. The `Shape` class defines the protocol by which all derived shapes must abide.

Inheritance can serve too distinct purposes: it can be used to factor the interface (i.e, via polymorphism and virtual functions) or it can be used to factor the implementation (by sharing common structure and behavior). In this example the `Shape` class is trying to do both at once, and therein lies the problem.

Since `Shape` is used by many clients, it should insulate its implementation. Because `Shape` also serves to factor common implementation details (e.g., origin) it fails to completely insulate its interface from its implementation.

Consider who would have to recompile if the implementation of coordinates were changed from `int` to `short`.

There are (at least) two ways to solve this problem; both require the addition of a new class.

FIGURE 34. Solution #1: Hold A Pointer To Your Data.

This solution is the standard, straightforward bruit force insulation technique. Create a class that holds all of your data (it could even be a struct). Remove all *inline* functions from the header file of the shape abstraction and add a forward declaration of the ShapeData class.

In `shape.c` you will `#include "shape_data.h"`. Now forward all calls to your `ShapeData` class using the single pointer data member, `d_data_p`, as shown at the lower left of the figure. Changes to `ShapeData` may force `shape.c` to have to recompile, but clients of shape will be spared.

The cost associated with this solution is that the `ShapeData` object will have to be allocated and decollated dynamically. Each data access will suffer one pointer indirection. And, as with any kind of insulating interface, functions cannot be *inline*.

FIGURE 35. Solution #2: Create A Protocol Class.

The more elegant solution is to factor the `Shape` class into two classes: one will serve to factor the interface, the other will serve to factor the implementation. The Shape class will be made a *pure protocol* class (i.e., it will have no data and declare only pure virtual functions). All functions in the existing class will become virtual. This class will be renamed `ShapeImpl` and serve to factor common implementation details.

One advantage of this approach over the first solution is that additional shapes that use the same implantation technique can inherit from ShapeImpl, while radically new implementations, such as `Box`, are at liberty to inherit directly from the protocol itself, allowing `Box` to share the interface without inheriting the unwanted implementation.

This approach costs nothing for functions that were already virtual, but does force all previously statically bound functions to become dynamically bound. This approach does avoid the non-negligible expense of having to allocate and decollate a second structure dynamically.

7.0 Interfaces

FIGURE 36. Abstraction Interfaces

The ability to right meaningful tests rests on having a crisp, well-defined interface. Small is beautiful when it comes to interfaces. Further, it is important to avoid letting the details of your particular implementation slip into the interface. The example in the upper right corner of the figure is a particularly ugly one in which the internal representation is completely exposed in the interface.

Allowing performance needs to warp the interface can have not only have an unnecessarily impact on the maintainability of the code, but can ultimately lead to lower performance. Altering the interface in a way that achieves higher performance by exposing the nature of the implementation puts additional, artificial constraints on the interface. The flexibility to radically alter the implementation is lost.

A good interface is very hard for a single person to create, particularly the implementor. There needs to be an independent reviewer of the interface whose concern is from the perspective of the users and testers of the abstraction. Issues such as encapsulation, insulation, levelization, etc. are too subtle for many implementors who are more interested in just getting something working. Ensuring good interfaces requires specialists and tools. We have found that a small group of Interface Engineers can support a large number of developers, and have been effective at ensuring consistent and minimal interfaces.

FIGURE 37. Class Interface Design (hard problem)

Knowing how to write classes takes the programmer from the realm of application design to library design and sometimes boarders on compiler design. There is a great deal of knowledge and good practices to be learned.

This figure could actually have been an entire book... and it is. "Effective C++" by Scott Meyers is (IMHO) a landmark work giving an excellent treatment of this topic.

FIGURE 38. A Development Flow for a Large System

The development flow for a large project is of critical importance. High-level requirements flow in from marketing and are passed along to the architects and are negotiated. The architects must specify the packages and their reproducibilities. The architect and/or the interface engineer determine the individual abstractions within the package. Typically the interface engineer is responsible for defining the low-level interfaces of the individual abstractions. When the developer and the interface engineer agree, the developer implements the abstraction.

Typically the interface will change during development but it is the responsibility of the interface engineer to protect the eventual interests of both the user and tester. When the developer is through, the interface engineer passes the abstraction along for testing. Additional feedback will be collected in the form of bug reports and enhanced documentation. When testing is complete, the accumulated comments form the bases of a reference manual. The architect will review the results to verify that the marketing requirements are being met.

Where in this process do we consider tweaking the code to obtaining high performance? An essential aspect of code tuning is that the temptation to do so be suppressed until the last possible moment.

1. Premature tuning of code can be costly in development time. During development, it is inevitable that interfaces will evolve. Abstraction and even entire packages my change radically or even disappear. A highly tuned implantation is always much more costly to discard than a straightforward one.
2. Even if the interface does not change, the 80/20 rule states that roughly 20% of the code accounts for 80 percent of the execution time. This ratio can actually be more pronounced in many cases. It is hard for even the most experienced programmers to predict exactly what a bottleneck in a complex system will be. Optimizing code that is not part of the bottleneck is worse than useless since it complicates the code unnecessarily, wasting development time in the process.
3. If good encapsulating interfaces are in place, regression tests can be created based on the straightforward implementations, and then used to ensure correctness of the more elaborate, higher performance versions.

Inlining is a common technique for increasing the speed of function calls in small executibles. In large executibles it can actually have the opposite effect.

4. Consider a machine with a fixed amount of real memory. If the size of the code for the inlined call is larger then a normal function call and that call is made frequently, the overall executable size may become significantly larger. If many such functions are inlined, the effect can be to bloat the size of the executable beyond what can be reasonably supported by the physical memory of the machine.
5. Furthermore, the frequently-called function will no longer exist in a single place; instead, its multiple inlined copies will be distributed across many pages in memory, thwarting the caching mechanism of the host computer.
6. Finally, premature inlining forces compile-time dependencies on other abstractions that slow the compilation process and make debugging and profiling more difficult. On large projects, changing the implementation of a primitive class (e.g., `String`) would cause the entire world to have to recompile.

Only when the system is stable, and a bottleneck is established through objective measurement, is performance tuning appropriate

8.0 Conclusion

FIGURE 39. Conclusions

Using C++ effectively in large projects requires actively avoiding the pilaffs that lead to degraded behavior. The first step is education: become aware of the problems and their causes. Once enlightened, take what may at first seem like extreme measures to ensure the quality of the design throughout the entire development process.

9.0 Appendix

FIGURE 40. Appendix: Some Useful Tools:

Tools are an essential ingredient of Object Oriented Software Engineering. Surprisingly, commercial tools are sorely lacking. As a means of survival we have created our own tools in order to facilitate our C++ development effort.

DESIGNING-IN QUALITY

Wednesday, October 21, 1992

Presented By: John S. Lakos

Mentor Graphics Corporation

IC Group Engineering

Figure 1.

Advantages Of C++ Over C

If used properly, C++ can

- make programs easier to comprehend.
- provide more modular, maintainable, and reusable software.
- reduce type-based programming errors.
- facilitate evolution and extensibility.
- allow for more incremental/hierarchical testing.
- improve run-time performance.

Figure 2.

Disadvantages of C++ Compared to C

If used improperly, C++ can result in

- order of magnitude loss in run-time performance.
- order(s) of magnitude increase in compile time.
- huge executable size.
- cyclic dependencies among most/all modules.
- compile-time dependencies among most/all modules.
- software that is difficult to understand/maintain.
- code that can be neither tuned nor enhanced.
- greatly reduced reusability.
- programs that are (practically) untestable.

Figure 3.

FOCUS OF THIS TALK: QUALITY *ENSURANCE*

QA is responsible for *ASSURING* quality.

- QA is often involved only late in the development process.
- QA typically does not influence the design of the product.
- QA often relies on engineering to produce regression tests.

ENGINEERING is responsible for *ENSURING* quality.

- Quality must be designed-in from the very start.
- It cannot be QA'ed in after the design is nearly complete.

Designing-in quality is expensive.

Failing to design-in quality is *more expensive*.

Figure 4.

"No Silver Bullet"

Designing-In Quality Is Hard Work.

- There is no substitute for
  - Experience.
  - Intelligence.
  - "Good Taste".
- No tool will
  - Design-in the quality for you.
  - Ensure your design does what the specification says.
  - Generate exhaustive test suites for you automatically.
- It takes time to
  - Create an *acyclic* high-level architecture.
  - Flesh out *consistent*, *sufficient*, and *minimal* interfaces.
  - Minimize coupling (especially cyclic coupling) of abstractions.
  - Package code in such a way as to minimize compiler dependencies.

Mentor Graphics = IC Group

Figure 5.

Designing-In Quality...

When do we start?

- Before the first line of code is ever written.
- Before detailed interfaces are considered.
- Before the high-level architecture is established.

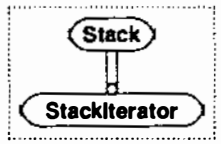
START NOW!

Mentor Graphics = IC Group

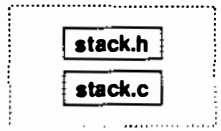
Figure 6.

Example: A Stack Abstraction

```
class Stack {
  public:
    Stack();
    Stack(const Stack& stack);
    Stack& operator=(const Stack& stack);
    ~Stack();
    void push (int i);
    int pop();
    int top() const;
    int isEmpty() const;
};
class StackIterator() {
  public:
    StackIterator(const Stack& stack);
    void operator++();
    operator const void *() const;
    int operator()() const;
};
```



Logical View



Physical View

Mentor Graphics = IC Group

Figure 7.

The Abstraction Level

What is an abstraction?

- It is the smallest physical entity in the system.
- It has a single header file (.h).
- It has a single implementation file (.c).
- It represents a concept.
- It is implemented as closely-related classes.
- Its header *MUST* declare all external names.
- Its .c file *MUST* includes the corresponding .h file.

■ Note: classes within a single abstraction may (and often do) have cyclic dependencies.

stack abstraction

Physical View

Logical View

Mentor Graphics = IC Group

Figure 8.

Logical Relationships (IsA and Uses)

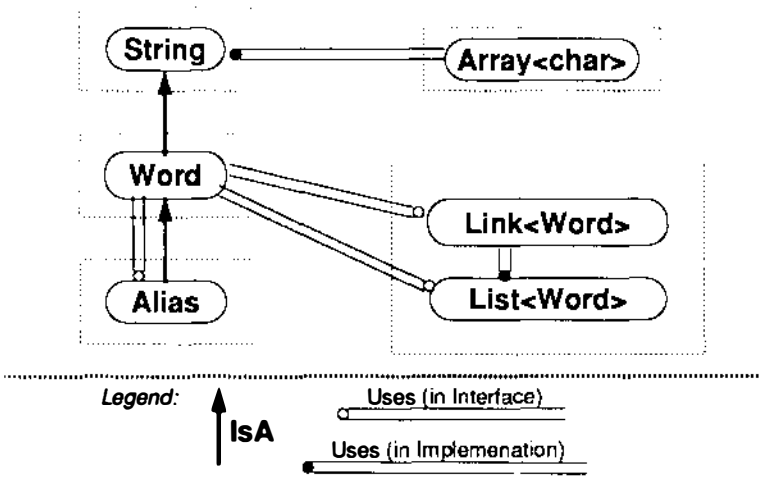


Figure 9.

Physical Relationship: DependsOn

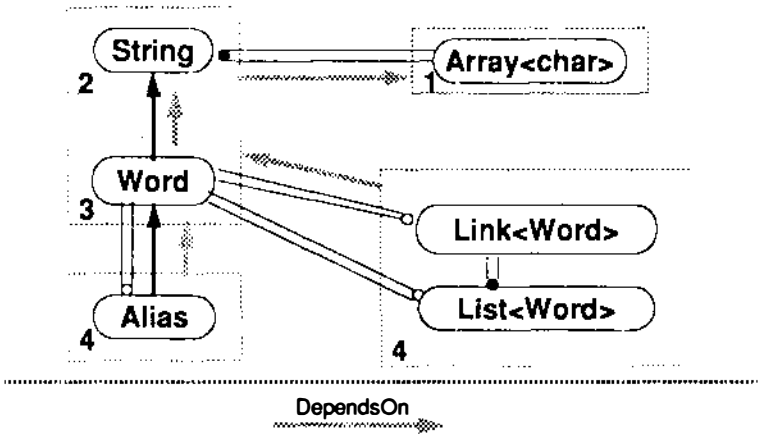


Figure 10.

Design For Test

High-Quality Designs Can Be Tested Hierarchically:

- The advantage of testing subsystems is proven by IC Designers:
  - Built-In Self Test (BIST) is common in large circuits.

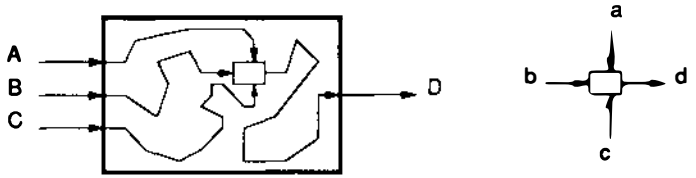
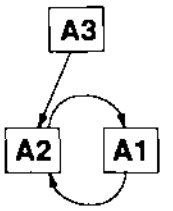


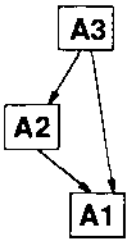
Figure 11.

Avoid Cycles In The Dependency Graph

Designs With Acyclic Dependency Graphs are *Levelizable*



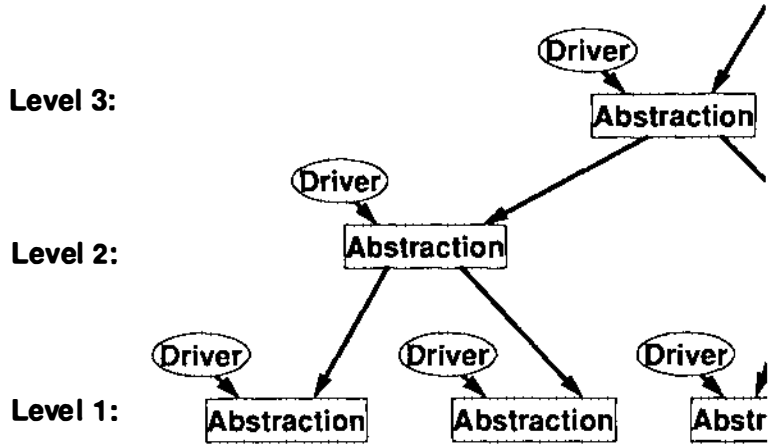
Cyclic Dependencies (Not Levelizable)



Acyclic Dependencies (Levelizable)

Figure 12.

Hierarchical Testing Of Levelizable Abstractions



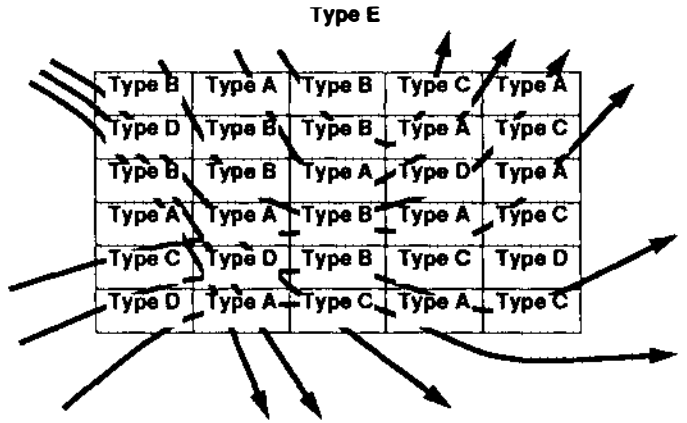
8/24/92 13

Mentor Graphics IC Group

Figure 13.

Integration Testing: Testing The Interaction

Incremental Testing: No need to retest the sub-objects



8/24/92 14

Mentor Graphics IC Group

Figure 14.

Testable Designs Are Levelizable

Levelizable designs can be tested hierarchically

- Each leaf abstraction can be tested in isolation.
- Higher-level abstractions can be tested in context.
- Only the value added at each level need be tested.

Effective testing requires interpretation

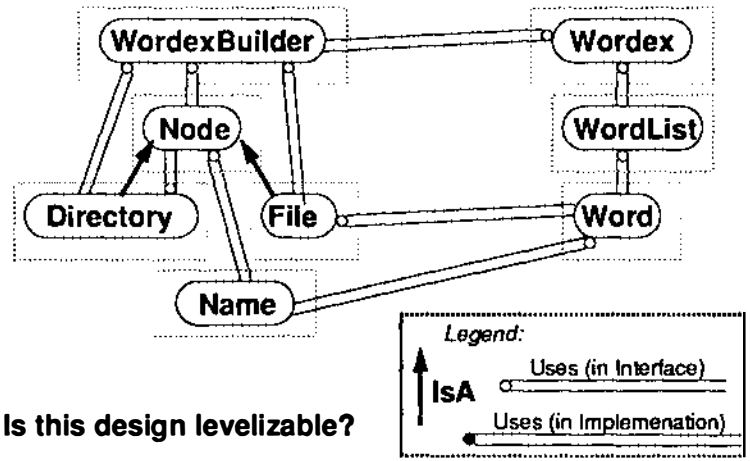
- Recompile/relink whenever a small test is to be added? (too painful)
- Relink each test when doing comprehensive regression tests? (too costly)
- We want a single interpreter to grow to be able to test large subsystems.
- We must be able to reuse tests at any level in the design hierarchy.

8/24/92 15

Mentor Graphics IC Group

Figure 15.

Levelization Example



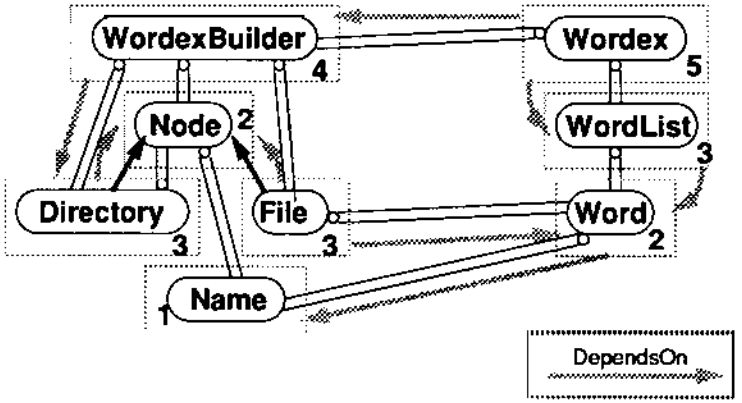
Is this design levelizable?

8/24/92 16

Mentor Graphics IC Group

Figure 16.

Levelized Diagram



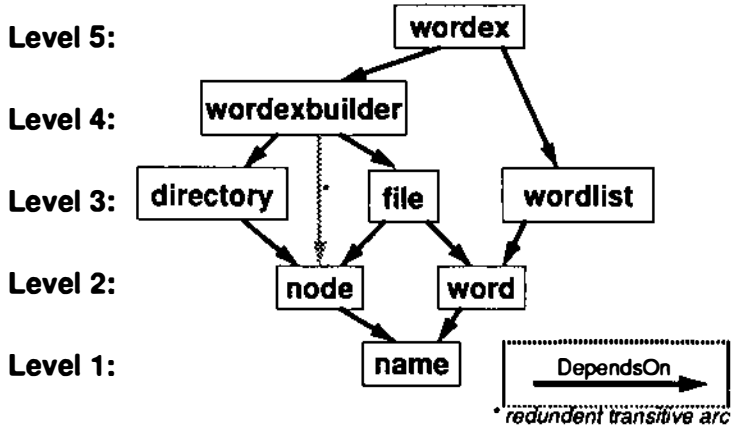
8/24/92 17

Mentor Graphics = IC Group

Figure 17.

Abstraction Dependency Diagram

Dependency diagram generation is easily automated.



8/24/92 18

Mentor Graphics = IC Group

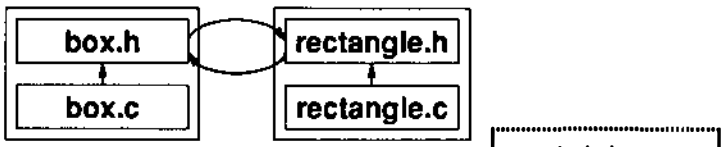
Figure 18.

Levelizability is *Not* Automatic

Problem: Each header file includes the other.

```
#include "rectangle.h"
class Box {
   // implementation
 public:
   Box(int x1, int y1, int x2, int y2);
   Box(const Rectangle& rect);
   // ...
};                box.h
```

```
#include "box.h"
class Rectangle {
   // implementation
 public:
   Rectangle(int x, int y, int wid, int len);
   Rectangle(const Box& box);
   // ...
};                rectangle.h
```



Note: Above example needs include guards to compile.

8/24/92 19

Mentor Graphics = IC Group

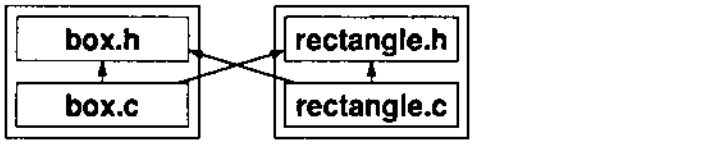
Figure 19.

Forward Declarations Do Not Solve This Problem

Each implementation file must include *both* header files.

```
class Rectangle;
class Box {
   // implementation
 public:
   Box(int x1, int y1, int x2, int y2);
   Box(const Rectangle& rect);
   // ...
};                box.h
```

```
class Box;
class Rectangle {
   // implementation
 public:
   Rectangle(int x, int y, int wid, int len);
   Rectangle(const Box& box);
   // ...
};                rectangle.h
```



8/24/92 20

Mentor Graphics = IC Group

Figure 20.

Solution #1

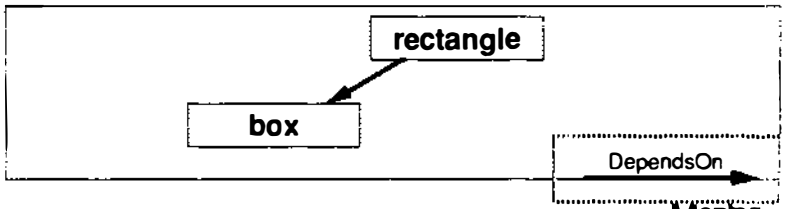
Let *rectangle* know about and depend on *box* but not visa versa.

```
class Box {
   // implementation
  public:
   Box(int x1, int y1, int x2, int y2);
   // . .
};
```

Box.h

```
# include "box.h"
class Rectangle {
   // implementation
  public:
   Rectangle(int x, int y, int wid, int len);
   Rectangle(const Box& box);
   // ...
   operator Box () const
};
```

Rectangle.h



8/24/92 21

Mentor Graphics IC Group

Figure 21.

Solution #2

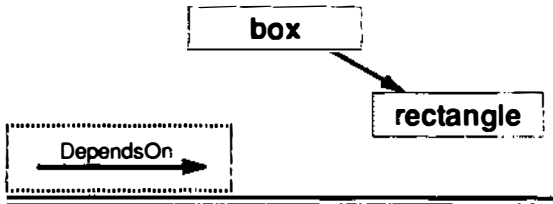
Let *box* know about and depend on *rectangle* but not visa versa.

```
class Rectangle;
class Box {
   // implementation
  public:
   Box(int x1, int y1, int x2, int y2);
   Box(const Rectangle& rect);
   // ..
   Rectangle toRectangle() const;
};
```

box.h

```
class Rectangle {
   // implementation
  public:
   Rectangle(int x, int y, int wid, int len);
   // ...
};
```

rectangle.h



8/24/92 22

Mentor Graphics IC Group

Figure 22.

Solution #3

Make a separate *convert* facility (neither is aware of the other).

```
class Box {
   // implementation
  public:
   Box(int x1, int y1, int x2, int y2);
   // ...
};
```

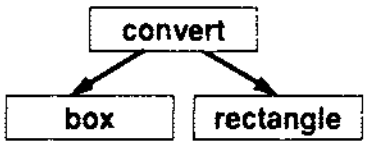
box.h

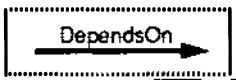
```
class Rectangle {
   // implementation
  public:
   Rectangle(int x, int y, int wid, int len);
   // ...
};
```

rectangle.h

```
class Box;
class Rectangle;
struct Convert {
   Box toBox(const Rectangle& rect);
   Rectangle toRectangle(const Box& box);
};
```

convert.h





8/24/92 23

Mentor Graphics IC Group

Figure 23.

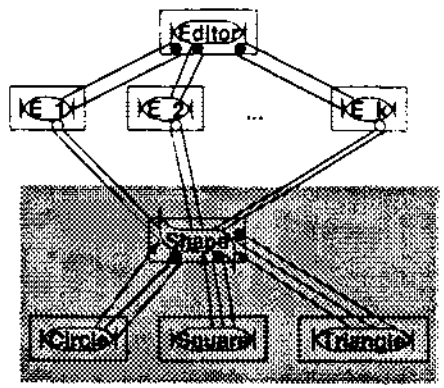
Example: A Shape Editor

Cyclic Dependency: the *Shape* class knows about its derived classes.

```
class Shape {
   int d_xOrigin,
   int d_yOrigin;
  public:
   static Shape *create(const char *);

   Shape(int x, int y);
   Shape(const Shape& shape);
   Shape& operator=(const Shape&);
   virtual ~Shape();

   void moveTo(int x, int y);
   virtual void draw() const = 0;
};
```



8/24/92 24

Mentor Graphics IC Group

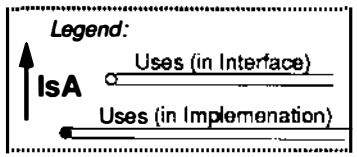
Figure 24.

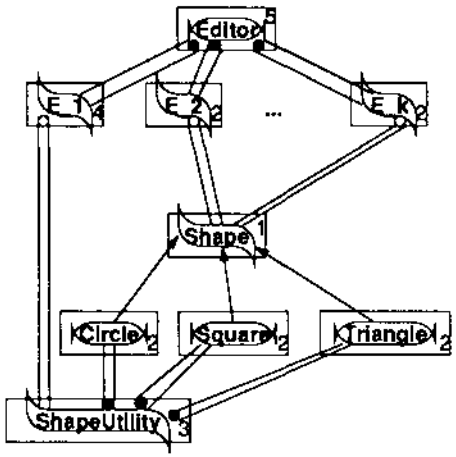
Shape Editor: A Better Design

No Cyclic Dependencies

- More Extensible
- More Reusable
- More Easily Tested

```
struct ShapeUtility {
    static Shape *create(const char *);
};
```





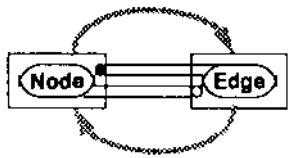
8/24/92 25

Figure 25.

Graphs: Nodes and Edges (Two Mutually Dependent Classes)

```
class Edge;
class Node {
  Edge *d_edges_p;
  int d_edgeCount;
  void add(Edge *edge);
  friend class Edge;
public:
  Node (int x, int y);
  ~Node();
  int isConnected (const Node& node) const;
  //...
};                    node.h
```

```
class Node;
class Edge {
  Node *d_node1_p;
  Node *d_node2_p;
public:
  Edge();      // floating edge
  Edge(Node *n1, Node *n2);
  Node *node1() const;
  Node *node2() const;
  // ...
};                    edge.h
```



8/24/92 26

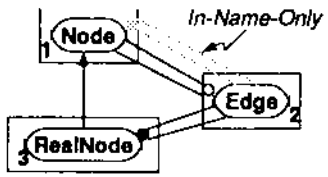
Figure 26.

Graphs: Nodes and Edges (Trying To Break The Cycle)

```
class Node {
  friend class Edge;
  virtual void add(Edge *edge) = 0;
  public:
    virtual ~Node();
    virtual int *isConnected(const Node&) const = 0;
};                    node.h
```

```
# include "node.h"
class RealNode : public Node {
    Edge *d_edges_p;
    int d_edgeCount;
    void add(Edge *edge);
  public:
    Node (int x, int y);
    ~Node()
    int isConnected(const Node& node) const;
};                    real_node.h
```

```
class Node;
class Edge {
    Node *d_node1_p;
    Node *d_node2_p;
  public:
    Edge(); // use to test without node
    Edge(Node *n1, Node *n2);
    Node *node1() const;
    Node *node2() const;
    // ...
};                    edge.h
```

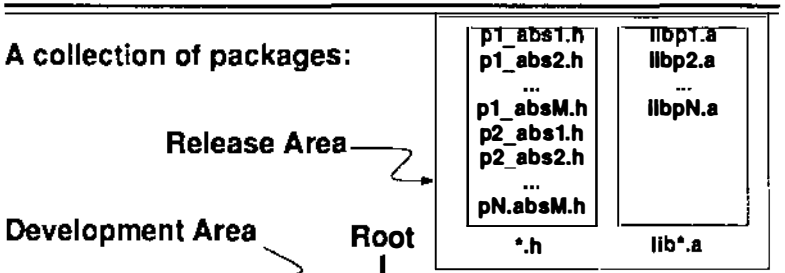


8/24/92 27

Figure 27.

The Big Picture

A collection of packages:



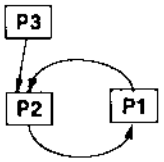
8/24/92 28

Figure 28.

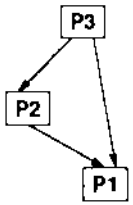
High-Level Architecture Design

Ensure Package Dependencies Are Acyclic:

- Enhance Use
- Enhance Reuse
- Enhance Testability



Cyclic Design



Acyclic Design

8/24/92 29

Mentor Graphics IC Group

Figure 29.

The Package Level

What is a package?

- A package is a fundamental architectural component.
  - Forms a coarse system building block.
  - Consists of a collection of cohesive abstractions.
  - Exports a library of .o files (.a file) and a directory of .h files.
  - Often, is implemented by a single developer and/or at a single location.
  - Has an associated unique registered prefix.
- Each package has a collection of responsibilities in the system.
- Each package has a collection of dependencies on other packages.
- Architects should specify the form of the dependencies among packages.
  - Logical Dependencies: *Interface* vs. *implementation.*
  - Physical Dependencies: *Size/Layout* vs. *In-name-only.*

8/24/92 30

Mentor Graphics IC Group

Figure 30.

Example: Encapsulation vs. *Insulation.*

Class "Foo" *encapsulates* class "Bar".

Abstraction "john" *insulates* users of "John" from changes in "Bar".

```
#include "bar.h"

class Foo {

    Bar d_bar;

  public:

  //...
```

foo.h

```
class Bar;

class John {

    Bar& d_bar;

  public:

  //...
```

john.h

8/24/92 31

Mentor Graphics IC Group

Figure 31.

Encapsulation vs. *Insulation*

Encapsulation: A *LOGICAL* property of a design.

An implementation detail (type, data, or function) that is not accessible programmatically from the interface of an abstraction is "encapasulated".

- Consider what the user needs in the interface.
  - Avoid exposing other details of implementation.
  - Try to think of at least two ways to implement your interface.

Insulation: A *PHYSICAL* property of a design.

An implementation detail (type, data, or function) that can be altered without forcing clients of the abstraction to have to recompile is "insulated".

- Where possible use objects *in name only* in header files.
  - Avoid forcing client code to know the size/layout of your objects.

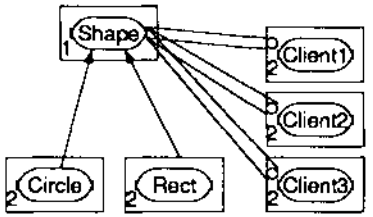
8/24/92 32

Mentor Graphics IC Group

Figure 32.

Problem: Shapes (with Uninsulated Implementation)

```
class Shape {
    int d_x;  // could change to short
    int d_y;  //  "      "      "   "
  public:
    Shape(int x, int y);
    virtual void draw() const;
    int xOrig() const { return d_x; }
    //...
};          shape.h
```




```
class Circle : public Shape {
  int d_radius;
 public:
  Circle(int x, int y, int r);
  void draw() const;
  //...
};          circle.h
```

```
class Rect : public Shape {
  int d_width;
  int d_height;
 public:
  Rect(int x, int y, int w, int l);
  void draw() const;
  //...
};          rect.h
```

```
# include "shape.h"
void Client3::draw() {
  // draw each shape...
  Shape *p;
  while (p = nextShape())
    p->draw();
  // ...
}          client3.c
```

8/24/92 33

Figure 33.

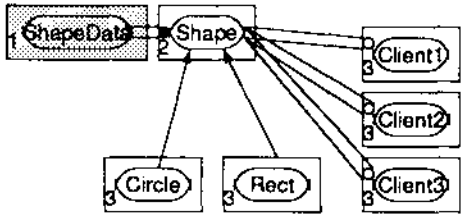
Solution #1: Hold A Pointer To Your Data.

Forward function calls to data class.

```
class ShapeData;
Shape {
  ShapeData *d_data_p;
 public:
  Shape(int x, int y);
  virtual void draw() const;
  int xOrig() const; // forward call
  // ...
};          shape.h
```

```
#include "shape_data.h"
int Shape::xOrig() const {
  return d_data_p->xOrig();
}
          shape.c
```

```
class ShapeData {
  int d_x;  // could change to short
  int d_y;  //  "      "      "   "
 public:
  ShapeData(int x, int y);
  int xOrig() const { return d_x; }
  //...
};          shape_data.h
```



8/24/92 34

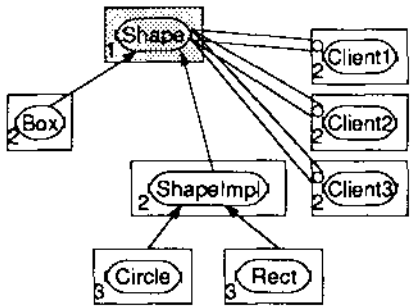
Figure 34.

Solution #2: Create A Protocol Class.

```
class Shape {
 public:
   virtual void draw() const = 0;
   virtual int xOrig() const = 0;
   //...
};          shape.h

class ShapeImpl : public Shape {
   int d_x;  // could change to short
   int d_y;  //  "      "      "   "
 public:
   Shape(int x, int y);
   void draw() const;
   int xOrig() const;
   //...
};          shapeImpl.h
```

Clients work off of a protocol only.



8/24/92 35

Figure 35.

Abstraction Interfaces

What's In an Interface?

- As little as possible (but no less).
  - Reduces complexity.
  - Facilitates usability.
  - Facilitates testing and maintenance.
  - Allows for reimplementation.

Avoid Premature Optimizations!

- Consistency
  - C++ interfaces raise issues that are far too subtle for casual attention.
  - Important issues can appear as merely style to the uninitiated.
  - Ensuring consistency requires a specialist (as well as tools).
  - A small group of "*Interface Engineers*" can support many developers.
  - Interface engineers have been effective in ensuring consistently correct (and minimal) interfaces.

When In Doubt... Do without!

```
class Array {
  int *d_array_p;
 public:
  Array(int *array);
  // ...
  int *getIntArray() const;
  // ...
};          array.h
```

(A Bad Interface)

8/24/92 36

Figure 36.

Class Interface Design (probably harder than you think)

Mastering the Issues... **Warning: "IsA", "HasA", and "Inline Functions" can introduce compiler dependencies, *beware!***

- Private Inheritance or Layering (in the context of virtual functions)?
- Passing parameters by value, reference, or pointer?
- Returning parameters by value, reference, or pointer?
- Member versus free function?
- Const or non-const member?

What is the appropriate declaration for:

- operator +=
- operator +
- (prefix) operator++ (postfix)?

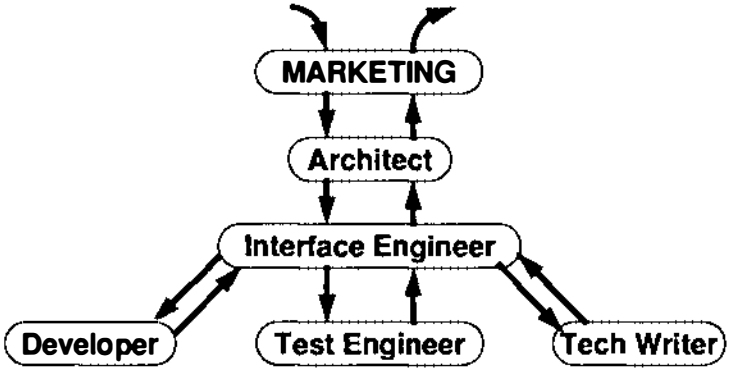
8/24/92 37

Mentor Graphics = IC Group

Figure 37.

A Development Flow for a Large System

The Interface Engineer Ensures Quality In The Interfaces...



8/24/92 38

Mentor Graphics = IC Group

Figure 38.

Conclusions

Designed-In Quality *DOES NOT JUST HAPPEN.*

- must be aggressively sought fron the start..
- requires *commitment* from management.
- requires design, development, and testing *tools*.
- requires *knowledge*, *insight*, and *experience*.
- requires *discipline* in following established practice.

The most Important Ingredient Is the mInd-set.

- Methodology, tools, etc. can help.
- The discipline and commitment are up to you.

8/24/92 39

Mentor Graphics = IC Group

Figure 39.

Appendix: Some Useful Tools:

- Idep - determine Implementation DEPendencies (levelization).
- Uses - determine interface dependencies (encapsulation).
- Lint++ - detect errors/inconsistences in the interface.
- StubGen - used to generate the bindings of a .c file from a .h file.
- Cx - used to develop and test objects interactively.
- CxBind - used to generate Cx bindings directly from a header file.
- CxGenie - used to generate tests for Cx-based exercisers.
- MetaClass DataBase - used to hold representation of program.
- MetaClass ToolKit - used to write arbitrary tools acting on interfaces.
- Diff++ - used to compare interfaces (including structured comments).
- Build - used to create hierarchical snapshots of a subsystem for testing.

8/24/92 40

Mentor Graphics = IC Group

Figure 40.

Industrial Applications of the Category-Partition Method

Thomas J. Ostrand
tostrand@siemens.siemens.com
James M. Wood
jwood@siemens.siemens.com

Siemens Corporate Research, Inc.
755 College Road East
Princeton, NJ 08540
(609) 734-6500

ABSTRACT: The category-partition method of generating test cases from a functional specification has been successfully implemented as the Test Specification Language (TSL). This tool has been used by the software development groups of several Siemens operating companies since 1990. This paper discusses the benefits of the category-partition method, experience with TSL in several applications, its effect on testing practices and productivity, and how its use has affected its revisions.

KEYWORDS: Software testing, functional testing, specification-based testing, automatic test generation, formal specification languages, category-partition method.

Thomas J. Ostrand is a Senior Research Scientist in the Software Engineering Department at Siemens Corporate Research. He holds an S.B. in mathematics from M.I.T. and an M.S.E. and Ph.D. in computer science from the University of Pennsylvania. His interests include automatic test case generation, test adequacy criteria, software reliability, and software safety.

James M. Wood is a Research Scientist in the Software Engineering Department at Siemens Corporate Research. He received his B.S. in applied mathematics from Carnegie-Mellon University and his M.S.E. in computer science from the University of Pennsylvania. His interests include expert systems, software testing, test adequacy criteria, automatic test case generation, and formal specification languages.

1. INTRODUCTION

The quality of delivered software has a major impact on product competitiveness in the marketplace. Ensuring quality in an increasingly complex and sophisticated software development environment gives testing a critical role in the process. Consequently, software testing and quality are directly related to one another. Because testing consumes a significant portion of the software development process, industry has great need for tools that automate the testing process. The successful deployment of these tools in industrial applications can both reduce the cost of software and improve its resulting quality.

The automatic generation of test data and test cases is an important area of software testing research and development. Research has focused on techniques of deriving test data either directly from code or from specifications. Code-based techniques derive test data from the implementation. While having the advantage of achieving significant coverage of code in tests, these techniques cannot derive *expected* results from the code to verify correctness. Furthermore, code-based tests can easily miss errors that result from omitting required code. Specification-based techniques derive test data from representations of the function under test, possibly from design, requirements, or functional specifications, where expected results may be specified for verifying correctness.

Research in this area at Siemens Corporate Research (SCR) has produced the category-partition method, a specification-based technique that systematically generates test cases from a high-level test specification. The category-partition method has been implemented as the Test Specification Language (TSL) tool.

There are other tools in the commercial marketplace that generate test data from specification-based information. The TDGEN product [6] is a component of the Test Regression Toolset of Software Research, Inc. This tool generates test cases for execution from two files. The first defines the input parameters and possible values for testing. The second defines a template textual format for test cases, with references to input parameters as variables for substitution. The T product [4] of Programming Environments, Inc. has four processes. It translates a software description internally, performing semantic analysis, then partitions the input for testing and generates test cases, analyzes the results and comprehensiveness of test cases, and provides a way to alter the evaluation of tests.

The TSL tool has been internally available to Siemens operating companies since 1990. Among its customers, it has been given both experimental and production use in a wide variety of applications. Major applications of TSL include system testing of software embedded in magnetic resonance imaging (MRI) systems, integration and system testing of operating system utility programs, regression and new feature testing of telephone switching software, and unit and module testing of embedded C functions in patient monitoring systems. Its use in these and other domains demonstrates its domain independence and versatility.

This paper is organized into four principal sections. We first define the category-partition method on which TSL is based. Second, we discuss the testing practices and requirements of TSL users prior to their use of it in their testing environments. Third, we describe the application of the category-partition method to their testing problems. Fourth, we evaluate the method and TSL in these environments and identify some areas for future improvement. We then conclude the paper with a summary of the TSL applications and the main benefits that have been realized from its use. Appendix A gives an example of a TSL specification.

2. THE CATEGORY-PARTITION METHOD

Partition analysis is a fundamental technique for generating tests at the functional level. At this level, the input domain may be partitioned into sets of data points for which the elements of the sets are expected to produce the same behavior for the function. The sets constitute *equivalence classes* whose elements may be considered interchangeable from the standpoint of functional testing. Several different variations of partition analysis have been specified [5,7], although there is controversy over its ability to expose program errors adequately [2].

The category-partition method [3] is a systematic approach for deriving functional tests from a product specification using partition analysis. It enables the user to develop a structured representation of an otherwise loosely structured functional specification. The method requires a user to specify the function being tested in terms of *categories* and to *partition* them into *equivalence classes*. Building a structured representation, the *test specification*, using the category-partition method involves these steps:

- Decompose the product specification into independently testable functional units, and apply the category-partition method separately to each unit.
- Identify explicit input *parameters*, external conditions or *environments* that affect function behavior, and expected output *results* of the function.
- Characterize properties of parameters, environments, and results as categories.
- Partition each category into *choices* or equivalence classes.
- Define constraints between choices of categories to restrict their combination and to specify when they occur.

The resulting test specification allows the tester to describe the function's input and output domains, and to control the content of generated test cases. Partitioning categories into choice classes implies that test cases produced by exhaustively combining elements of the choice classes achieve significant coverage of the function. The degree of coverage depends on how thoroughly the specification writer defines and constrains the test specification.

The category-partition method provides a means to specify expected results for general classes of input combinations. It also permits a user to decompose the various aspects of

a result into separately described equivalence classes. This process simplifies the specification of expected results, and makes it possible to build individual test cases where the expected results can be compared to actual results at execution time.

The category-partition method has been implemented at Siemens Corporate Research in the TSL tool, described more fully in [1]. A concise example of applying TSL to testing a C function is given in Appendix A. The tool has undergone one major and several minor revisions since its initial version, which supported the basic concepts of the category-partition method. These revisions were driven by requirements of the Siemens operating companies.

3. TRADITIONAL TESTING ENVIRONMENTS

Medical Imaging Software

A Siemens operating company in the medical products industry develops and tests software that drives magnetic resonance imaging (MRI) diagnostic equipment. The technician operates an MRI keyboard which controls many parameters of system operation (such as patient position, image characteristics, timing, and data recorded). The keyboard acts as an interface between the operator and embedded software running on a VAX processor. MRI software has more than 1.6 million lines of code across four thousand modules (forty megabytes of compiled code), implemented mostly in Fortran, with some Assembler and C. Third-party contractors do some (though not all) module development and testing. System tests are written to validate the software in its embedded environment.

Since no simulator is presently available for the equipment or its keyboard interface, system testing takes place on the actual hardware, and a human operator executes test scenarios which check complete system behavior. System *test scripts*, or *test books*, consist of sequences of test cases for a test operator to execute, and a typical test case is a sequence of instructions following this structure:

- a series of operations to establish or *setup* the scenario of the test case,
- operations to create images for the test case, and
- instructions for the operator to record displayed values or to compare computed images to reference images.

Such test cases may have anywhere from twenty to fifty instructions, and there are many possible variations. Thorough equipment testing requires building many instruction sequences, using many combinations of variables. The chief goal is to test all significant instruction sequences representing actions likely to be taken during actual equipment operation. Because of the large number of variations possible for each instruction, complete testing of all sequences is clearly prohibitive, both for test generation and execution, yet it is critical to execute a set of test cases that ensures confidence in the system.

In their manual testing process, software developers design these system tests by hand, attempting to cover behavioral aspects of the entire system. As a manual process, this lacks a systematic approach and does not guarantee full coverage of system behavior. The primary method to evaluate software correctness and reliability is for an experienced operator to exercise the system in as many different ways as possible. Although these informal testing practices can be valuable in detecting system flaws, engineers no longer wish to rely solely on them.

Operating System Utilities

At one European operating company, software is developed for utilities and compilers of the BS2000 operating system. The software development process consists of four parts, modeled after the IBM programming process architecture: requirements, development, quality assurance, and customer service. Testing has the highest concentration in the development and quality assurance phases. Development includes the study of requirements, design, implementation, and unit testing; quality assurance includes functional, product, and system testing.

Software utilities and compilers are developed in this organization as tools on the BS2000 operating system. Before distributed architectures became popular, these tools were written to interface directly with the operating system, and system users wrote programs to interface with either the tools or the operating system. Today, the operating system is divided into subsystems that interface with a BS2000 control program. User programs now interface with the tools or subsystems.

Programming languages used in this environment include C, C++, Assembler, and SPL, a PL-1 language used with a data base for regression testing. Tools developed with these languages include distributed command interfaces, user programs, data base management tools, and language interfaces. The software varies widely in size, from five thousand to nearly three hundred thousand lines of code, hence the software development process must support both small scale and large scale systems. Real-time, distributed data base, network, and hardware interface requirements also complicate the scenario of software development.

Unit testing of software, which includes function and module testing, takes place during the development phase. The remainder of testing (functional testing, product testing, and system testing) is the responsibility of quality assurance. In the large development organization, engineers have responsibility for unit testing of their code as well as design and implementation of it. Unit testing consists of test suite development, execution, and result analysis. Quality assurance, where testing is the major activity, encompasses integration and system testing. Each quality assurance engineer is a dedicated software tester responsible for test suite development and execution, and test result analysis. Most testing is based on functional specifications, although some code-based testing is undertaken.

A manual testing environment is not adequately efficient for their requirements. Manual development of complete test suites is difficult, tedious, prone to errors, and not

systematic. Developing and executing test cases and analyzing test results is very time consuming. Automation must also be extended to regression testing and analysis of regression test results, particularly if automatic generation produces much larger sets of test cases.

Telephone Switching

The software testing environment for a public network switching system developed by another Siemens operating company is highly complex. The telephone switch which they develop and manufacture must support as many as one million subscriber lines, and system design is very hierarchical. The hardware structure of this system has three tiers of subsystems. At the first level is the set of *coordination processors*, each of which consists of a number of *line-trunk groups* at the second level, each of which also consists of a number of *digital line units*. The digital line unit is the third and lowest level to which telephone lines are connected. Software is developed for all levels, but primarily for the higher two.

Software testing for the switch is very extensive. New features and functionality are tested at the unit and module levels, and then at the integration level. System level testing focuses primarily on stress testing, by heavily loading the system with many simulated callers placing calls to one another over specified periods of time. Because features are added to the switch and switch releases are modified over time, regression testing also figures importantly in the testing environment for this product.

Testing at the unit and module level requires simulators to simulate the behavior of absent software or hardware components. Additionally, human actions and manual telephone operations are simulated in testing. During feature testing, one simulator performs the manual processes required to exercise the feature. During stress testing, another simulator places many calls between many callers over time.

The attributes of this development environment present many challenges to testing and many opportunities for automation. The amount of time to create, execute, and evaluate test cases (at all levels) must be minimized while achieving an adequate level of test coverage. A tight production schedule mandates a highly regulated testing schedule, often challenged by changes in software specifications, bug fixes, and unexpected or delayed test results.

Patient Monitoring Systems

Another Siemens company develops embedded software for complex patient monitoring systems in the medical domain. In the system for which TSL was eventually used, two to four thousand C functions spanning across two to four hundred modules (one to two hundred thousand lines of code) must be maintained. This code is developed on SUN workstations. Testing of the software follows a rigorously defined process, with unit, module, integration, and system testing.

Unit testing of C functions and modules is managed by module developers. Prior to subsystem and system integration testing, functions and modules are tested for correctness. Testing a C function in this environment requires writing a test driver for the function, supplying inputs to the driver and calling the function from it, and evaluating results. Depending on the function, unit testing may be performed in a stand-alone environment, a simulated environment, or the actual hardware environment.

There are a few important difficulties with testing C functions in this system. First, because the developer is entirely responsible for unit testing, there is less control over ensuring that design requirements, with respect to the function under test, are validated during unit testing. Second, the magnitude of unit testing two to four thousand C functions is so enormous as to require some form of automation.

4. APPLICATIONS OF THE CATEGORY-PARTITION METHOD TO THE TESTING PROCESS

Medical Imaging Software

MRI system testers have used TSL since 1990 to generate test books for integration and system testing. Because a human operator actually executes tests from the console, the output of the test generation process is English text of detailed instructions for the operator, produced by a standard document formatter, *Runoff*. System test designers write TSL specifications to produce the source files for *Runoff*.

The production of an efficient and thorough set of test cases requires consideration of the following issues:

- All generated combinations of instructions must be systematically recorded.
- Each test sequence must be correct, containing the proper instructions and the proper result verification information.
- The number of test cases must be limited to a manageable number for execution.
- No critical combinations may be omitted.

TSL has helped test designers with each of these issues. In a TSL specification, the test designer writes a general form for the test cases, with parameterized input and environment values. When TSL creates the executable test scripts, it replaces the parameterized values with specific values built from the choices specified by the test designer. This provides the tester with documented evidence that certain tests were created. The parameterized test scripts written in TSL greatly simplify the creation of test verification code, since *verify* statements themselves are parameterized.

Limiting the number of produced test cases is an important capability of TSL. Initial use of TSL by the MRI development group resulted in more than fifty thousand test cases for only one aspect of system use. While meeting the criteria for test thoroughness, executing

all of these test cases was obviously impractical. Judicious application of TSL *if* and *limit* statements have reduced this output to approximately one thousand test cases for the latest release.

It is more difficult to guarantee that no critical test combinations are omitted. However, because test scripts are produced from TSL specifications representing the unrestricted exhaustive test set and then filtered with *if* and *limit* statements, the category-partition method provides higher confidence that critical test combinations are covered than an ad hoc definition of tests. Since its initial use of TSL, the MRI development group has generated nearly all of its system tests with the tool, writing by hand only a few special cases for testing random misuse of the equipment.

Operating System Utilities

Within the BS2000 utilities development and quality assurance groups, TSL has been used in unit, integration, and system testing. Among the fifteen engineers using it, the category-partition method has been successful in identifying function categories and choices, thus producing better test specifications and, consequently, better test scripts with better test cases. TSL is used in conjunction with an in-house data base tool for maintaining test cases for regression testing.

TSL use in this environment has increased steadily since its introduction in 1990; the tool is producing many more test cases than previously produced by hand. Table 1 shows numbers of tests recently developed with and without TSL during different testing phases by the quality assurance group.

Table 1: Number of test cases and TSL utilization

Phase of testing	*Test cases without TSL*	*Test cases with TSL*	*Percent of TSL use*
functional testing	*318*	*1420*	*82%*
subsystem testing	*51*	*131*	*72%*
system testing (application 1)	*20*	*3000*	*99%*
system testing (application 2)	*160*	*250*	*71%*
system testing (application 3)	*395*	*105*	*21%*

The first column in Table 1 identifies the number of test cases manually written for the test phase, and the second column indicates the number of test cases generated by TSL. The last column is the percentage of total test cases generated using TSL.

This summary shows that TSL produces many more test cases than were produced manually, and that testers are relying on it heavily. All of the TSL users feel that systematic use of the category-partition method enables them to design better test

specifications and better tests. Several users state that TSL simplifies the maintenance of tests and regression test suites.

TSL has greatly reduced the amount of time these engineers spend in developing test cases. As a systematic approach to testing functional components, it generates much more complete test suites. Additionally, it generates some useful test cases not normally produced by an experienced tester.

Telephone Switching

The software development tools group of the public networks organization has designed and implemented the Darts Testing Language (DTL) to automate the execution and verification of regression and new feature tests. Automation of test creation, execution, and evaluation allows for repeated test execution throughout the software development cycle. Considerably more testing can be accomplished in a given amount of time, resulting in improved software quality.

DTL and its underlying testing system allow the tester to describe an application environment, the specific commands to test, and the expected outcome of a test. To test a new telephone feature with DTL, the human tester writes the DTL commands that

- *create* the environment in which the feature is tested,
- *perform* specific actions associated with testing the feature (for example, to dial a telephone number), and
- *verify* that the feature is exercised properly.

Although DTL greatly simplified the process of describing test cases, the human tester still had to determine the command sequences that comprised the test cases, choose values for test parameters, design the test cases, and write DTL commands by hand. The tools group evaluated TSL as a high-level language for describing test cases. In the study, TSL specifications were written that defined the various components of a feature test scenario as parameter and environment categories. Expected results of testing the feature were also defined. To cover the possible situations of each component, each category of the scenario was further partitioned into choice classes. From the test specification, TSL then generated all possible combinations of choices for the categories of the scenario to obtain many test cases for the feature. Script files were produced by TSL containing DTL commands to establish the scenarios of test cases, perform the appropriate actions, and verify that expected results occurred. These script files were then automatically executed with simulators, requiring human analysis only for debugging failed tests.

The success of this feasibility study and recent upgrades to the TSL software have prompted this organization towards using TSL in a variety of testing environments. More recent use of TSL has resulted in the generation of three thousand DTL test cases. Use of TSL has begun for module testing of software in conjunction with simulator use at both the coordination processor and line-trunk group levels. TSL generates test scripts in formats required by the simulators.

Patient Monitoring Systems

TSL has also been used to generate code-based tests for unit and module testing of the patient monitoring system. In this environment, TSL is used to generate test scripts consisting of a test driver and input/output pairs for testing C functions. The tester writes a TSL specification in which the function's input parameters correspond to TSL parameter and environment categories, and its output parameters correspond to result categories. Rather than write a test driver by hand, the tester also includes appropriate TSL statements to structure the resulting test script as a legal C module. From the test specification, TSL produces the test driver together with a series of test values. The test driver module is compiled and linked with the function under test, and automatically executed for observation of test case results.

Going one step farther, this organization, in collaboration with SCR researchers, developed a system that analyzes the code of a C function and generates a partial TSL specification for a test driver. From the code, it extracts the function name, formal parameters, and global variables used in the function. Based on the data types of function variables, the system generates choices for the corresponding TSL categories. For numeric variables, valid choices within and on a numeric boundary and invalid choices outside of the boundary are included. The *null* and *not-null* choices are used for pointer variables. For enumerated data types, choices are created for elements of the enumerations. This information is used to build a partial *template* of the TSL specification, to be completed by hand. Because *expected* results cannot be determined directly from code, it is impossible for the TSL specification generator to derive the *if* conditions that characterize a result, yet the template provides the backbone of a TSL specification.

This organization developed another tool to automate regression testing. This tool maintains a data base file of valid test cases for comparison with subsequent test cases to facilitate regression testing. Given the result of executing a test case, the system searches the data base for a previously executed test case with the same input values, and then compares their output values to verify correctness. If no matching case is found, the tester is asked to verify the results manually. If the results are correct, the new case is added to the data base. If the results are incorrect, the user can either discard the test, or add the test to the data base with its results corrected manually. Developers envision using this tool for unit testing of functions in combination with the TSL specification generator and TSL.

The use of TSL in this testing environment demonstrates its applicability to C function testing, one of the most common software development environments. Its use in combination with a TSL specification generator and an automated regression tool is an important step towards complete automation of the unit testing process.

5. AREAS FOR IMPROVEMENT

Through the use of TSL and the category-partition method in the field, we have identified a number of areas of the system that need improvement. These items are of two basic types: specific problems with writing and executing TSL specifications, and general issues concerning the category-partition method. The former type are generally amenable to correction through more careful language design and improved algorithms. The latter must be addressed with thorough training of TSL users, and the use of good design methods and tools throughout the development process.

The specific problems include performance degradation under certain circumstances and the inability to predict the sequence of test cases that the system will produce from a specification.

The performance of TSL suffers in the presence of complex constraints placed on a test specification. With the *if* statement, the TSL language provides a mechanism for placing constraints in the specification that define (as boolean expressions) when combinations of choice classes of categories may occur. As TSL builds a test case, the constraint expressions must evaluate to *true*. Through repetitive selection of categories and assignment of choices to them, it is possible to violate constraints. Satisfying these constraints then requires *backtracking*, which may be exponential in time.

The current version of TSL produces test scripts in which the individual test cases occur in an unpredictable order. In the initial release of TSL, test cases were ordered based on the order of the categories within the test specification, and the order of the choices within each category. However, the latest version of the tool has more advanced system-defined and user-defined adequacy criteria that, although resulting in better coverage with fewer test cases, gives the appearance of random organization to the test cases. Consequently, the user has no explicit control over *ordering* test cases. Test cases cannot be grouped, there can be no relationship between consecutive test cases, and one test case cannot rely on the results of the prior test case during execution.

The more general issues include the fact that although it offers a systematic *approach* to the creation of a test specification, the category-partition method depends on user expertise and experience. The proper definitions of categories and choices in a test specification are frequently not obvious and require knowledge of the tested system's functionality and potential errors in its implementation.

6. CONCLUSIONS

We have described how the category-partition method, through the implementation of the TSL tool, has been applied to industrial applications. TSL has been used in these domains to produce formal test documents, generate tests in customized languages for automatic execution and verification, and generate test drivers for C function testing. TSL's partition analysis approach to generating test cases covers a broad range of testing applications. Industrial experience verifies the domain independence of the tool.

The category-partition method and TSL have provided two major types of benefits to users: gains in their testing productivity, and improvements to the administration and management of their test information. Use of TSL has produced better test coverage of the application system's functionality, and greatly increased the number of tests that can be produced and executed in a given time, for all users. TSL also helps substantially with test documentation and maintenance. The very act of using TSL forces the tester to document the tests, and the standard format simplifies test corrections, additions, and deletions.

Siemens operating companies plan to continue using TSL in their production environments. In testing software embedded in MRI systems, TSL will be used to generate formal test documents, with the added goal of producing these documents in *Framemaker*, a more modern document processor language. Use of TSL to test BS2000 operating system utility programs will continue, along with additional testing of system interfaces with TSL. In addition to its current use for regression and new feature testing of public telephone switch software, TSL will be applied to the earlier stages of module and subsystem testing. Finally, TSL may eventually be integrated into an automated testing environment, together with tools for generating a test specification and automatically performing regression test analysis.

7. ACKNOWLEDGMENTS

We are indebted to the Siemens users of TSL for their contributions. Susan Rossnick, Alexander Chu, Rainer Schulz, and David Stevens described their use of the tool in MRI applications. Michael Defamie was instrumental in its deployment in the BS2000 utilities development environment. David Towe and Shawn Malaney in public switching encouraged investigation of the tool in their organization. Peter Morgan in patient monitoring systems and Michael Platoff at SCR were responsible for evaluating TSL and developing a test specification generator to interface between a C module and TSL.

At Siemens Corporate Research, Marc Balcer, William Hasling, and Marcus Yoo have assisted in various phases of TSL development. Thomas Murphy, William Hasling, and Patricia Vroom managed the project from inception to its current state. Finally, William Hasling, Michael Platoff, Robert Schwanke, and William Sherman evaluated this paper for clarity and correctness.

REFERENCES

[1] Marc J. Balcer, William M. Hasling, Thomas J. Ostrand, "Automatic Generation of Test Scripts from Formal Test Specifications." *Proceedings of the ACM SIGSOFT 1989 Third Symposium on Software Testing, Analysis, and Verification.* Key West, FL, December 1989.

[2] Dick Hamlet, Ross Taylor, "Partition Testing Does Not Inspire Confidence." *Proceedings of the Second Workshop on Testing, Analysis, and Verification.* Banff, Alberta, July 1988.

[3] Thomas J. Ostrand, Marc J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests." *Communications of the ACM*, volume 31, number 6, June 1988.

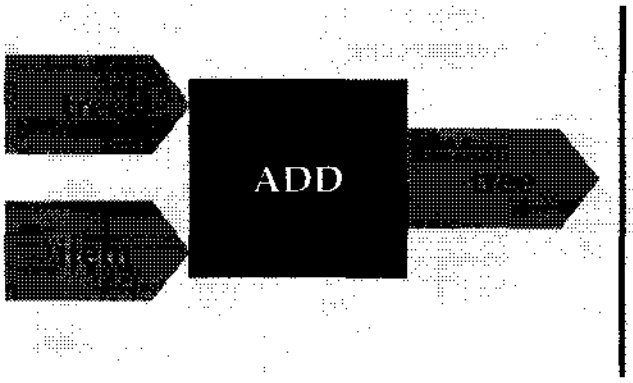
[4] Programming Environments, Inc., *T Technical Perspective.* Programming Environments, Inc., Tinton Falls, NJ, January 1990.

[5] Debra J. Richardson, Lori A. Clarke, "A Partition Analysis Method to Increase Program Reliability." *Proceedings of the Fifth International Conference on Software Engineering*. San Diego, CA, March 1981.

[6] Software Research, Inc., *TDGEN User Manual, Release 3.* Software Research, Inc., San Francisco, CA, November 1991.

[7] Elaine J. Weyuker, Thomas J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains." *IEEE Transactions on Software Engineering*, May 1980.

APPENDIX A. TSL EXAMPLE




```
#include <stdio.h>
#include <strings.h>
#include "bintree.h"

TREE_PTR add(tree, item)
    TREE_PTR    tree;
    char        *item;
{
    . . . .
    return tree;
}
```

Figure 1: Specification and code segment for binary tree ADD function

The tester is given the specification for an *add* function which adds a character string to a binary tree data structure. The item is added to the tree if and only if it is a valid character string and it is not already a member of the tree. Figure 1 shows both the high level specification and an outline of the implementation. For this example, the code implementation serves only to provide the syntax of the function call.

The TSL user starts building a test specification from the function specification by identifying the function's inputs and outputs. The inputs to the function include the *tree* and the *item* added to the tree; these are mapped into *parameter* categories, which are further partitioned into choices. For example, the tree may be *empty* or *nonempty*.

The function's output is the resulting tree, with the item either *added* or *not added*. These two possible results are mapped into the *result* categories *item_added* and *item_not_added*. To check whether the implementation behaves correctly when attempting to add a duplicate item, the tester defines an *environment* category called *duplicate_item*, whose effect is to create several test cases with the item parameter already in the tree. The *if* condition for the *true* choice of *duplicate_item* says that such a test case should be created for each input pair with a non-null item and a non-empty tree.

The *if* conditions under the *result* categories characterize when those results are expected to occur. An item is not added to the tree (result *item_not_added*) if either the item is null or the item is a duplicate, and an item is added (result *item_added*) if the item is not null and the item is not a duplicate.

The TSL user completes the test specification by adding *setup*, *cleanup*, *function*, and *verify* statements to define C statements linked to components of the specification. For example, when the tree is *nonempty*, three C statements are executed in the *setup* section to ensure that the variable *input_tree* has two elements in it. The *verify* statements of the result categories check the actual test output against the expected result. For the purpose of this example, the *verify* code is very simplistic, merely comparing the sizes of the tree before and after the call to the *add* function.

The complete test specification is shown in Figure 2. TSL builds the executable test script by forming combinations of the category choices and combining appropriate pieces of the test specification. Figure 3 shows the structure of the generated test script, displaying the complete code for just one of the test cases. This test case tries to add a duplicate *asparagus* item to the tree, and checks to ensure that it is not added. The script is a valid C test driver consisting of the *main* function. The highlighted test case consists of statements to *setup* the test case, the call to the tested function *add*, and code to *verify* that the expected result has occurred.

```
TEST add_function
   FUNCTION [  output_tree = add($tree, $item);]
   SCRIPT SETUP [#include <stdio.h>
                 main() {
                   TREE_PTR input_tree, output_tree;
                   . . . .]
   SCRIPT CLEANUP [}]
   SETUP [  test_failed = FALSE;]
   CLEANUP [   if (test_failed) . . . .]
PARAMETER tree
   *  empty
      VALUE [input_tree]
      SETUP [  input_tree = NULL;]
   *  nonempty
      VALUE [input_tree]
      SETUP [  input_tree = NULL;
               input_tree = add(input_tree, "kumquat");
               input_tree = add(input_tree, "rutabaga");]
PARAMETER item
   *  null
      VALUE [NULL]
   *  "asparagus"
   *  "nectarine"
ENVIRONMENT duplicate_item
   *  false
   *  true
      IF (item != null) and (tree != empty)
      SETUP [  input_tree = add(input_tree, $item);]
RESULT item_not_added
   IF (item = null) or (duplicate_item = true)
   SETUP [  input_size = size(input_tree);]
   VERIFY [    output_size = size(output_tree);
               if (input_size != output_size)
                  test_failed = TRUE;]
RESULT item_added
   IF (item != null) and (duplicate_item = false)
   SETUP [  input_size = size(input_tree);]
   VERIFY [    output_size = size(output_tree);
               if (output_size != input_size + 1)
                  test_failed = TRUE;]
```

Figure 2: TSL test specification for ADD function

```
#include <stdio.h>
#include "bintree.h"
main() {
   TREE_PTR input_tree, output_tree;
   int input_size, output_size;
   BOOLEAN test_failed;
   . . . .
   printf("Test Case 8:\n");
   test_failed = FALSE;
   input_tree = NULL;
   input_tree = add(input_tree, "kumquat");
   input_tree = add(input_tree, "rutabaga");
   input_tree = add(input_tree, "asparagus");
   input_size = size(input_tree);

   output_tree = add(input_tree, "asparagus");

   output_size = size(output_tree);
   if (input_size != output_size)
      test_failed = TRUE;
   if (test_failed)
      printf("Test case 8 failed.\n");
   else printf("Test case 8 passed.\n");
   . . . .
}
```

Figure 3: Structure of test script for ADD function, including sample test case

Combat: A Compiler Based Data Flow Testing System [1]

Mary Jean Harrold and Priyadarshan Kolte
Department of Computer Science
Clemson University
Clemson, SC 29634-1906
harrold@cs.clemson.edu

Abstract

Combat, our **COM**piler **BA**sed **T**ester, is based on the the GNU C compiler, and performs unit testing of C procedures using data flow testing methods. Our tester is quicker to implement and executes faster than previously developed testers. It handles arbitrary C programs, and can easily be modified to test programs written in other source languages such as FORTRAN and C++. The **Combat** system includes tools to graphically view the procedure under test and to automatically detect and eliminate redundant test cases.

Biographies

Mary Jean Harrold is an assistant professor in the Department of Computer Science at Clemson University. She received an MS and a PhD in computer science from the University of Pittsburgh. Her research interests include programming language implementation, program analysis and testing and program development, testing and maintenance environments. She is a member of IEEE Computer Society and the ACM.

Priyadarshan Kolte is a graduate student in the Department of Computer Science at Oregon Graduate Institute. He received an MS in Electrical Engineering and an MS in Computer Science, both from Clemson University. His research interests include computer architecture, compiler optimizations and software testing. He is a student member of ACM.

[1] This work was partially supported by NSF under Grant CCR-9109531 to Clemson University.

1 Introduction

Since testing is a critical part of program development that requires considerable effort, tools that automate the testing process are important and useful. *Data flow testing*[5][10][14] is a program-based testing technique that uses the flow of data in a program to determine whether the program is adequately tested. Many existing data flow testers are *source level* testers[4][10][3] that translate the program under test to some intermediate form and perform data flow analysis to compute the required testing information. Such a tester unnecessarily duplicates a large part of the computation that is usually performed by a compiler. Moreover, the usefulness of the tester is limited to testing programs written in the source language for which it is developed. On the other hand, a *compiler based tester*[2][8][6][12][13] uses an existing compiler to translate the program to some intermediate form. Since the tester is dependent on the intermediate form of the program and not on the source language, the tester can take advantage of other compilers that translate different programming languages to the same intermediate form. Further, since the information required for data flow testing is quite similar to that required by compiler optimizations, a compiler based tester can reuse the data flow information gathered by the compiler. Thus, a compiler based tester is faster to implement and can easily be adapted to test programs written in different languages.

A data flow tester generates a set of testing requirements for the program under test. As the program is executed with different test cases, the tester uses the program's execution paths to identify the testing requirements that are satisfied by each test case. The program under test is determined to be adequately tested if the test cases satisfy all testing requirements. Previous testers used an *acceptor* to identify the unsatisfied testing requirements. Since an acceptor determines whether a program's execution path satisfies only one testing requirement at a time, repeated applications of the acceptor are needed to obtain all unsatisfied requirements. On the other hand, *dynamic data flow analysis*[11] identifies *all* testing requirements satisfied by a test case in one application of the algorithm while making only one pass over the program's execution path. Thus, dynamic data flow analysis can identify the satisfied testing requirements more efficiently.

This paper describes our data flow testing system, **Combat** (**COM**piler **BA**sed **T**ester), that is based on the GNU C compiler, **gcc**[2][15]. The main benefit of **Combat** is that it does not have to translate C programs for analysis and testing since it operates on **gcc**'s intermediate code. **Combat** reuses the data flow information gathered by the compiler and only computes some additional data flow information required for testing. Thus, additional languages may be accommodated with moderate changes to **Combat**. The second benefit of **Combat** is the use of dynamic data flow analysis to improve the efficiency of the testing process. Finally, **Combat** contains several additional tools including one to graphically view the program to assist the user in developing test cases, and another to provide automatic detection and elimination of redundant test cases[9].

The remainder of this paper is organized as follows. Section 2 provides background on data flow testing. Section 3 provides an overview of the **Combat** system and describes its components. Section 4 describes the operation of **Combat**. Finally, section 5 provides conclusions.

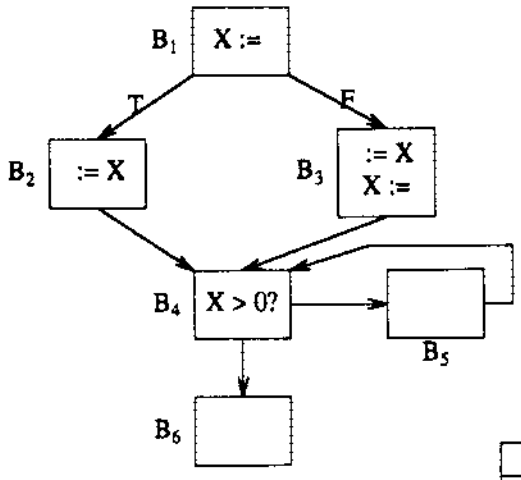
2 Data Flow Testing

The overall goal of testing is to provide confidence in the correctness of a program. With testing, the only way to guarantee a program's correctness is to verify it on all possible inputs, which is clearly impossible.

[2] Copyright (C) 1987, 1989 Free Software Foundation, Inc, 675 Mass Avenue, Cambridge, MA 02139.

Thus, systematic testing techniques generate a representative set of test cases to provide coverage of the program according to some selected criteria. A given set of test cases is *adequate* for a program with respect to a particular criterion if it covers the program according to that criterion. Data flow testing offers a family of adequacy criteria for testing programs, including 'all-definitions' and 'all-uses'.[3]

In data flow testing[5][10][14], a variable assignment in a program is tested by generating test cases that execute subpaths from the assignment (*definition*) to points where the variable's value is used (*use*). Uses in the program are either *computation* uses (c-uses) or *predicate* uses (p-uses)[5]. A c-use occurs whenever a variable is used in a computation or output statement; a p-use occurs whenever a variable is used in a conditional statement. Data flow testing techniques represent a program by its *control flow graph* where each node in the graph represents a basic block consisting of statements that are executed sequentially and each edge represents the flow of control between the basic blocks. Traditional data flow analysis techniques[1] are used to identify definition-use pairs. Figure 1 shows the control flow graph for a program segment.



definition	*definition-use pairs*
X in B_1	$(B_1, B_2), (B_1, B_3), (B_1, (B_4, B_5))$ $(B_1, (B_4, B_6))$
X in B_3	$(B_3, (B_4, B_5)), (B_3, (B_4, B_6))$

Testing Information		
test	*execution path*	*definition-use pair(s)*
1	B_1, B_2, B_4, B_6	$(B_1, B_2), (B_1, (B_4, B_6))$
2	$B_1, B_2, B_4, B_5, B_4, B_6$	$(B_1, B_2), (B_1(B_4, B_5)), (B_1, (B_4, B_6))$
3	B_1, B_3, B_4, B_6	$(B_1, B_3), (B_3, (B_4, B_6))$
4	$B_1, B_3, B_4, B_5, B_4, B_6$	$(B_1, B_3), (B_3, (B_4, B_5)), (B_3, (B_4, B_6))$

Figure 1: **Control flow graph and testing information for a sample program segment**

Definitions of variable **X** are in B_1 and B_3. C-uses of **X** are in B_2 and B_3 while B_4 contains a p-use of **X**. For the definition of **X** in B_1, definition-c-use pairs are (B_1 ,B_2) and (B_1, B_3) while definition-p-use pairs are (B_1, (B_4, B_5)) and (B_1, (B_4, B_6)). The definition-p-use pairs for the definition of **X** in B_3 are (B_3, (B_4, B_5)) and (B_3, (B_4, B_6)); there are no definition-c-use pairs for **X** in B_3.

[3]The 'all-nodes' and 'all-edges' structural criteria are also included for completeness, although they are not actually based on data flow in the program.

Test data adequacy criteria are used to select particular definition-use pairs or subpaths that are identified as the *test case requirements* for a program and test cases are generated that satisfy these requirements when used in a program's execution. A test case satisfies a definition-use pair if executing the program with the test case causes traversal of a subpath from the definition to the use without any intervening redefinition of the variable. Such a subpath is called a *def-clear subpath*. For c-uses, traversal must be from the definition block to the use block while for p-uses, test cases must traverse subpaths from the definition block to both successors of the use block. The 'all-uses' criterion[5] requires that each definition of a variable be tested on some subpath to each of its uses. The 'all-uses' criterion has been shown to be feasible since relatively few test cases typically are required for its satisfaction[16]. As the program is executed with each of the test cases as input, the execution path is recorded and associated with those definition-use pairs that it traverses. A possible 'all-uses'-adequate test set for the program segment in Figure 1 is shown in the table. The testing information gives the execution path for each test case along with the definition-use pairs that it satisfies.

3 Overview of Combat

Combat is based on the gcc compiler. Although our testing system is portable, we are currently experimenting on Sun-4 workstations[4]. This version of **Combat** supports the 'all-defs' and 'all-uses' data flow testing criteria, and 'all-nodes' and 'all-edges' structural testing criteria. **Combat** is used for unit testing of C procedures, and it handles programs composed of multiple procedures. Although **Combat** currently handles only simple variables, we are extending it to handle structured variables and pointers, and we are adding interprocedural testing capabilities.

The **Combat** system consists of six tools that perform the following four functions:

- **Program translation and analysis** is performed by the tool called **startexp**, which is a modified version of the gcc compiler.
- **Requirements satisfaction** determines the testing requirements that are satisfied by each test case. The tool called **addtest** performs this function using dynamic data flow analysis.
- **Test suite management** detects and eliminates redundant test cases in a test suite. The tool called **redtest** detects redundant test cases and another tool, **deltest** allows the user to eliminate redundant test cases.
- **User interface** reports the results of the **Combat** system in a convenient format to the user. The tool, **showexp**, displays the control flow graph and the definition–use pairs of the program under test that were computed by **startexp**. Another tool, **summexp**, displays the unsatisfied testing requirements that were computed by **addtest**.

3.1 Program Translation and Analysis

The four main objectives of this phase are (1) to translate the program under test to some intermediate form, (2) to instrument the program so that its execution can be observed, (3) to perform data flow analysis on the intermediate form of the program and (4) to maintain mappings between variables and statements in the source program and those in the intermediate program. The mappings between the source program and

[4] Sun-4 is a trademark of Sun Microsystems, Inc.

the intermediate program are important because, although the testing system operates on the intermediate program, the results are reported to the user in terms of the source program.

The **startexp** tool is the modified **gcc** compiler that performs program translation and analysis. First it translates the C program under test into the **gcc**'s intermediate code format, Register Transfer Language (RTL). Then, it instruments the program so that an execution trace will be produced at runtime, and performs some data flow analysis to supplement that performed by **gcc**. Finally, **startexp** produces mappings between the source program and the RTL program. Each invocation of **startexp** requires an experiment name, the name of the procedure to be tested and the name of the file containing that procedure.

We first provide an overview of the original **gcc** compiler[15], and concentrate on the operations and source files that we used to construct the **startexp** tool. We then describe the modifications that we made to the **gcc** compiler.

3.1.1 The original gcc compiler

The overall control structure of the compiler is in the file **toplev.c**. This file is responsible for initialization, decoding arguments, opening and closing files, and sequencing the passes. The different passes of the **gcc** compiler are :

- **Parsing** reads the entire text of a function definition and constructs partial syntax trees. The syntax tree representation supports other languages in addition to C.
- **RTL generation** converts the syntax tree into RTL code. RTL code is actually generated statement-by-statement during parsing, but for most purposes it can be thought of as a separate pass. **emit-rtl.c** is the main source file for RTL generation.
- **Various optimizations** such as jump optimization, constant propagation, common subexpression elimination, movement of invariant code out of loops, and strength reduction are performed.
- **Data flow analysis** divides the program into basic blocks and then performs live–range analysis on the pseudo-registers. The source file is **flow.c**.
- **Register allocation** first allocates hard registers to pseudo registers that are used only within one basic block (local allocation), and then to those whose live–ranges are not contained in one basic block (global allocation).
- **Reloading** renumbers the pseudo registers with the hardware registers numbers that they were allocated. Pseudo registers that did not get hard registers during register allocation are replaced with stack slots.
- **The final pass** performs machine-specific peephole optimizations and produces the assembly code for the function being compiled.
- **Debugging information output** is performed last because it has to output the stack slot offsets for pseudo registers that did not get hard registers. The source file for output in the DBX symbol table format is **dbxout.c**.

3.1.2 Startexp: Our modified gcc compiler

We first modified **toplev.c** to provide additional command line arguments to enable or disable the **Combat** procedures in **gcc**. Since the optimizing phase of **gcc** destroys much of the information needed for testing such as the mappings between line numbers of the intermediate code and the source code, we modified **toplev.c** to disable certain optimizations, including common subexpression elimination and movement of invariant code out of loops.

We modified several procedures in the file **emit-rtl.c** to provide a mapping from the line numbers in the intermediate code program to those in the source code. We also added a file called **ourdbx.c** that is based on **dbxout.c** to provide mappings between the names of variables in the intermediate code and those in the source program.

We made several modifications to procedures in the file **flow.c**. First, we added a procedure to insert probes in each basic block in the procedure under test so that as the program under test is executed, it provides an execution trace to a UNIX pipe. Second, since **Combat** requires reaching definition information to compute the definition–use pairs for testing, and **flow.c** does not compute reaching definitions, we modified it to perform this analysis.

Since some of the variables in the program under test may be allocated stack slots instead of pseudo registers, and **gcc** performs little analysis of variables in stack slots, our procedures in the file **ourdbx.c** provide the definition–use information for such variables.

3.2 Requirements satisfaction

This function identifies the testing requirements that were satisfied by the program while executing a test case. It uses the program's execution trace to determine the blocks, edges, and definition–use pairs that were executed. There are two approaches to determining whether a definition–use pair was executed. The first approach, the acceptor approach, scans the program trace for a def-clear subpath from the definition to the use. The second approach, dynamic data flow analysis, performs reaching definitions analysis on the program trace. We used the second approach in our tool **addtest**.

3.2.1 Dynamic Data Flow Analysis

Dynamic data flow analysis is similar to iterative algorithms for data flow analysis[1] except that instead of iteratively applying the data flow equations to all basic blocks and edges in the control flow graph, the data flow equations are applied to the sequence of basic blocks and edges that appear in the program execution trace for a particular execution. Thus, dynamic data flow analysis solves a data flow problem for a particular execution of the program as opposed to static data flow analysis, which solves the same problem for all possible executions of the program.

We used the data flow equations for reaching definitions in our dynamic data flow analysis, which yields the executed definition–use pairs for all definitions in just one forward scan over the program trace. Since we need only one forward scan over the program trace, we use a UNIX pipe to communicate the program trace to the data flow analyzer.

In practice, dynamic data flow analysis executes faster than the acceptor approach. Consider a program to be tested that has S statements and D definition–use pairs. Suppose k test cases are required to completely test the program (according to the 'all–uses' criterion), and the average length of the program execution trace is T blocks. Then the acceptor approach performs kDT comparisons to determine the satisfied testing requirements. Dynamic data flow performs kT set operations to obtain the same results. Since the size of the sets used in dynamic data flow is S, the total number of operations performed by dynamic data flow is kST. In our experiments, we found that the number of statements S is usually less than the number of definition–use pairs, D. Thus, dynamic data flow analysis performs fewer operations and hence is faster than the acceptor approach. An even greater speedup in execution of the dynamic data flow technique is achieved through our implementation of sets as bit–vectors. Since 32 set elements can be packed into a single integer in our implementation, one operation on this integer operates on 32 elements at once, and the total integer operations performed by the dynamic data flow method is $kST/32$.

3.2.2 `Addtest`

The **addtest** tool obtains the test case input from the user, executes the program under test, performs dynamic data flow analysis on the program, and identifies the unsatisfied testing requirements.

The program under test is instrumented to produce an execution trace to a named pipe by **startexp**. **addtest** executes the program under test in the foreground and the dynamic data flow analyzer in the background. The analyzer reads the named pipe to obtain the program execution trace, and performs the reaching definitions analysis to yield all the definition–use pairs in the trace.

3.3 Test Suite Management

Managing a test suite includes both detection and elimination of existing, redundant test cases. A test case is *redundant* if its removal does not affect the testing coverage provided by the test suite. A recent technique[9] uses an algorithm that identifies redundant test cases, while guaranteeing that the remaining test cases still satisfy the testing requirements. The algorithm finds a representative set of test cases that provides the desired testing coverage of the program. The problem of selecting a representative set of test cases that provides the desired testing coverage of a program or part of a program is stated as follows:

Given: A test suite TS, a list of definition-use pairs p_1, p_2, ... p_n that must be satisfied to provide the desired testing coverage, and a list of subsets of TS, T_1, T_2 ... T_n, one associated with each of the p_i's such that any one of the test cases t_j belonging to T_i can be used to test p_i.

Problem: Find a representative set of test cases t_j that will satisfy all of the p_i's.

A representative set of test cases that satisfies all of the definition-use pairs must contain at least one test case from each T_i. Such a set is called a *hitting set* of the group of associated testing sets T_1 , T_2 ... T_n. The maximum reduction in the test suite is achieved when the smallest representative set is found. However, this subset of the test suite is the minimum size hitting set of the T_i' and the problem of finding the minimum size hitting set has been shown to be NP-complete[7]. Thus, an approximation of this minimum set is found by using a heuristic[9].

The approximation heuristic considers associated testing sets T_i of ascending sizes, beginning with singleton sets, to find a test case to add to the representative set. For subsets of size n, it selects the test case

that appears in the greatest number of subsets. If, when examining the subsets of size n, there are several test cases that occur an equal number of times, the test case that occurs in the maximum number of subsets of size (n+1) is chosen. After each test case is added to the representative set, any other associated testing sets containing that test case are marked as being represented and no longer considered.

The **redtest** tool detects *redundant* test cases using the above heuristic. Redundant test cases depend on the data flow testing criterion selected by the user. Another tool, **deltest** allows the user to delete redundant test cases.

3.4 User Interface

The **showexp** tool is based on the X Window System[5], and provides a graphical view of the program under test. This tool displays the control flow graph of the program, and annotates the graph with the source code of the program. Optionally, **showexp** displays all the definition–use pairs in the procedure under test. The tool, **summexp**, uses the results produced by **addtest** and reports the unsatisfied testing requirements (in terms of the source program) to the user. The unsatisfied testing requirements depend on the data flow testing criterion selected by the user.

4 Operation of Combat

The user first compiles the procedure under test using our modified gcc compiler, **startexp**. If the program to be tested is composed of multiple procedures, the user can specify the procedure to be tested. Only the procedure to be tested needs to be compiled using **startexp**; all other procedures may be separately compiled using gcc. The user can then invoke **showexp** to graphically view the annotated control flow graph and the definition–use pairs in the procedure under test.

The user is responsible for developing test cases for the procedure under test and can repeatedly enter test cases using **addtest** until all testing requirements are satisfied or until he/she wants to stop the testing session (perhaps because a bug is detected). Each time the user enters a test case, the system executes the instrumented program, performs dynamic data flow analysis to determine unsatisfied testing requirements, and produces the output of the program under test. The user is expected to verify that the produced output of the program is correct. The user can then use **summexp** to display the unsatisfied testing requirements for all test cases for any of the supported testing criteria: 'all nodes', 'all-edges', 'all-defs' and 'all-uses'.

Finally, the user can use the **redexp** tool to detect all redundant test cases with respect to a specified testing criterion, and can eliminate such test cases using **deltest**.

4.1 Example of Operation of Combat

To illustrate the operation of **Combat**, consider the program 3sort.c that sorts 3 integers in ascending order as shown in Figure 2. We name the testing experiment **exp** by executing

startexp exp main 3sort.c.

[5]The X Window System is a trademark of the Massachusetts Institute of Technology.

```
/* 3-sort : sorts 3 input integers in ascending order */

main()
{
    int i, j, k, min, mid, max;

    scanf ("%d%d%d", &i, &j, &k);

    if (i < j) {
        min = i;
        mid = j;
    }
    else {
        min = j;
        mid = i;
    }
    if (k < min) {
        max = mid;
        mid = min;
        min = k;
    }
    else if (k < mid) {
        max = mid;
        mid = k;
    }
    else
        max = k;
    printf ("%d %d %d \n", min, mid, max);
}
```

Figure 2: **Program 3sort**

To illustrate the operation of **Combat**, consider the program 3sort.c that sorts 3 integers in ascending order as shown in Figure 2. We name the testing experiment **exp** by executing

startexp exp main 3sort.c .

A snapshot of the screen after executing **showexp** exp is shown in Figure 3. The window to the upper left displays the source program with line numbers. The window in the lower left corner partially displays the definition–use pairs produced by **showexp**. Finally the window to the right displays the annotated control flow graph of procedure main of program 3sort.c.

Assume that we provide the inputs 1 2 3 using **addtest**, then, the following output shows that the execution of the program on this input is correct.

```
TEST CASE 0 OUTPUT :
1 2 3
```

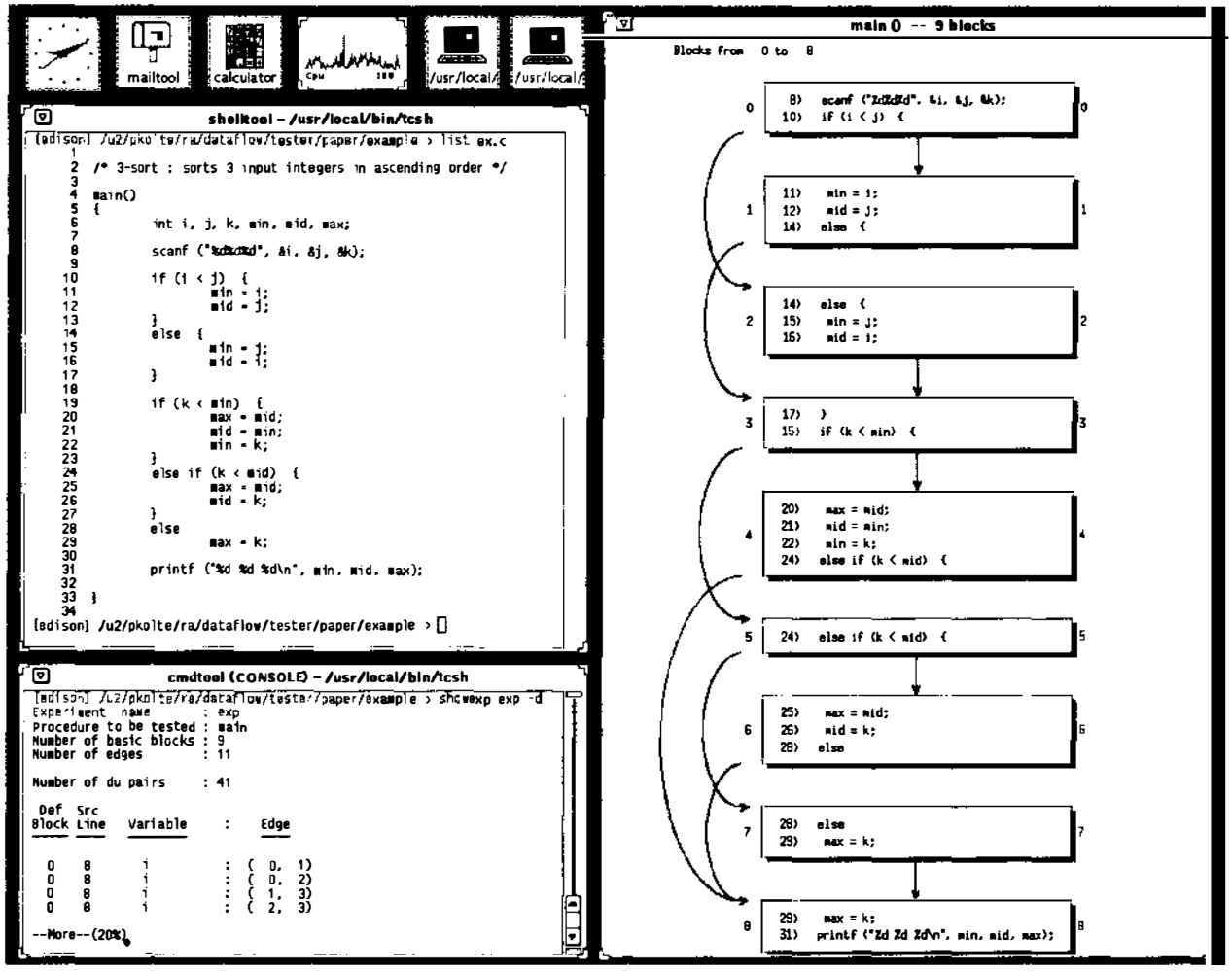


Figure 3: **Snapshot of screen**

We can view the unsatisfied basic blocks using **summexp exp -b** to get :

```
Test cases 0 to 0
number of blocks not satisfied = 3
block 2
block 4
block 6
```

Now the unsatisfied basic blocks using **summexp exp -b** are :

```
Test cases 0 to 1
number of blocks not satisfied = 2
block 4
block 6
```

We can instead view the unsatisfied edges using **summexp exp -e** to get :

```
Test cases 0 to 1
number of edges not satisfied = 4
from block 3 to 4
from block 4 to 8
from block 5 to 6
from block 6 to 8
```

We can also view the unsatisfied definition–use pairs using **summexp exp -d** to get :

```
Test cases 0 to 1

number of du-pairs not satisfied = 14

Def    Src
Block  Line   Variable  Edge


0      8      k       :  ( 3, 4)
0      8      k       :  ( 4, 8)
0      8      k       :  ( 5, 6)
0      8      k       :  ( 6, 8)
1      11     min     :  ( 3, 4)
1      11     min     :  ( 4, 8)
1      12     mid     :  ( 4, 8)
1      12     mid     :  ( 5, 6)
1      12     mid     :  ( 6, 8)
2      15     min     :  ( 3, 4)
2      15     min     :  ( 4, 8)
2      16     mid     :  ( 4, 8)
2      16     mid     :  ( 5, 6)
2      16     mid     :  ( 6, 8)
```

Since the execution path produced by test case 2 is identical to that produced by test case 1, test case 2 is redundant with respect to all testing criteria. Thus, executing **redexp exp -b** results in :

```
Experiment :  exp
Number of test cases :  3
Number of functional test cases :  0

All-Nodes Criterion
Number of redundant test cases :  1 (33.33 %)
Redundant test case numbers :  2
```

5 Conclusion

We have described **Combat**, our data flow testing system that is (1) based on the GNU C compiler **gcc**, (2) uses dynamic data flow analysis to accept the test cases, (3) identifies redundant test cases, and (4) provides tools for user interface. We have shown that dynamic data flow analysis is more efficient than the traditional acceptor method for identifying which definition-use pairs are satisfied by test cases. Since our testing system is incorporated into a compiler and operates on the program's intermediate code, it avoids costly translation and analysis required to compute the data flow information. Thus, implementing a data flow tester for other languages requires only a front end translation and moderate changes to the existing tester. In fact, we have recently extended our data flow testing system to handle C++ programs using a modified version of the GNU C++ compiler **g++**, which translates programs to the same intermediate form as **gcc**. Currently, we are extending **Combat** to perform testing of interprocedural definition and use relationships. We are also using the testing system to perform studies on program metrics.

6 Acknowledgements

We wish to thank Dick Hamlet for his review of this paper and helpful suggestions for its improvement.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers, Principles, Techniques, and Tools,* Addison-Wesley Publishing Company, Massachusetts, 1986.

[2] R.A. DeMillo, E.W. Krauser and A.P. Mathur, "Compiler-Integrated Program Mutation," *Proceedings of the IEEE Computer Society 15th International Computer Software and Applications Conference (COMPSAC 91),* pp. 351-356, September 1991.

[3] P. Frankl and S. Weiss, "An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria," *Proceedings of the 4th Symposium on Testing, Analysis and Verification (TAV4),* October 1991.

[4] P.G. Frankl, S. Weiss and E.J. Weyuker, "ASSET: A system to select and evaluate tests," *Proceedings of the IEEE Conference on Software Tools,* New York, April 1985.

[5] P.G. Frankl and E.J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering,* vol. SE-14, no. 10, pp. 1483-1498, October 1988.

[6] J.P. Gannon, P. McMullin and R. Hamlet, "Data-abstraction implementation, specification, and testing," *ACM Transactions on Programming Languages and Systems,* vol. 3, no. 3, pp. 211-223, July 1981.

[7] M.R. Garey and D.S. Johnson, "Computers and Intractability, A Guide to the Theory of NP-Completeness," ed. V. Klee, W.H. Freeman and Company, New York, 1989.

[8] R.G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering,* vol. 3, no. 4, pp. 279-290, July 1977.

[9] M.J. Harrold, R. Gupta and M.L. Soffa, "A methodology for controlling the size of a test suite," *Proceedings of the Conference on Software Maintenance 1990,* pp. 302-310, December 1990.

[10] B. Korel and J. Laski, "A tool for data flow oriented testing," *ACM Softfair Proceedings,* pp. 35-37, December 1985.

[11] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters,* vol. 29, no. 3, pp. 155-163, October 1988.

[12] B. Marick, "The weak mutation hypothesis," *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pp. 190-199 October 1991.

[13] B. Marick, "Generic Coverage Tool (GCT) User's Guide," Testing Foundations, Champaign, IL, 1992.

[14] S.C. Ntafos, "An evaluation of required testing element strategies," *Proceedings of the 7th International Conference on Software Engineering,* pp. 250-256, March 1984.

[15] R.M. Stallman, "Using and porting GNU CC (version 1.37.1)," Free Software Foundation, Inc., Cambridge MA, pp. 73-77, February 1990.

[16] E.J. Weyuker, "The cost of data flow testing: An empirical study," *IEEE Transactions on Software Engineering,* vol. 16, no. 2, pp. 121-128, February 1990.

THREE WAYS TO IMPROVE YOUR TESTING

Brian Marick
Testing Foundations

Everyone needs to improve quality. A few companies are in a quality crisis, where dramatic and expensive changes may be the only recourse. But most companies aren't. Most companies need to improve quality in ways that are largely non-disruptive, that don't jeopardize a steady stream of new releases and new products. They seek modest but sustainable improvements, with a minimal risk of wasted effort.

This paper discusses three inexpensive, continuous, incremental improvements that can be made in software testing. They are:

(1) Using coverage to measure what your tests do, in order to discover weaknesses in test design.

(2) Using defect analysis to record what your tests missed, then building on that experience with defect catalogs.

(3) Increasing testability via debuggability, thus reducing the cost of testing.

BIOGRAPHICAL SKETCH

Brian Marick worked in industry for ten years as a tester, developer, and line manager, mostly on operating systems and compilers. Joint research with the University of Illinois led to internal consulting and then the creation of Testing Foundations. It exists to help companies acquire essential testing skills and tools. Because practitioners are justifiably suspicious of those who talk about software development but never actually do any, he spends half his time building and maintaining tools, some freely available.

Author's address: Testing Foundations, 809 Balboa, Champaign, IL 61820
Phone: (217) 351-7228
Email: marick@cs.uiuc.edu

1. Measuring and Improving Effectiveness Through Coverage

A test suite would be perfectly effective if it revealed every bug[1] in a program. While we have no way of measuring this, we still need to judge test quality. One approach is to measure how tests exercise the code.

Suppose you're testing a large system, especially one that's undergone a lot of perfective and corrective maintenance. The following is a very real risk:

(1) Some part of the system, call it NS, is not used in normal testing.

(2) Changes to other parts of the system break NS. It doesn't work at all.

(3) A customer is the first person to use a feature of the system that exercises NS.

This should have a relatively low chance of happening (though I've seen it depressingly often), but it has a very high cost. (Having a basic feature simply not work is a wonderful way of losing repeat business.) High risk translates into a task for testing: make sure all of the system's features have been exercised. Of course, you do this based on the system's specification or user documentation, but it's easy to miss a feature in an "evolved" system. How can you detect your testing mistakes and omissions?

A tactic that works well is to *instrument* every routine in the system so that it records whether it's been entered. This can have very low runtime overhead. Run your test suite. Any unused function likely corresponds to an untested feature. You should deduce what feature that is and design tests for it.

This example establishes a pattern for testing:

(1) Design your tests however you do now.

(2) At small additional cost, measure what your tests do. This is referred to as *test coverage*.

(3) Use uncovered parts of the code as pointers to weak test design. Improve your tests as time allows.

This type of coverage, *routine coverage*, is appropriate for large-scale system testing. I find other types of coverage useful for system components.

Types of Path-Based Coverage Useful for Subsystems

All of the measures described below (and more) are implemented by my tool GCT, which is available via anonymous FTP. Fetch the file cs.uiuc.edu:pub/testing/GCT.README for instructions. I can also supply it on tape for a materials and handling fee.

Branch coverage requires every branch to be executed in both directions. For this code

```
if (arg > 0)
{
 counter = 0;      /* Reinitialize */
}
```

branch coverage requires that the if's test evaluate to both true and false. These two requirements are the two *coverage conditions* for branches. Taking the branch both ways *satisfies* them.

[1] Technically, I should use the terms *fault*, for an actual incorrectness in program text, *failure*, for the incorrect output resulting from a fault, or *error*, for the programmer's mistake that caused the fault. Often, the distinction is unimportant, so I'll use "bug" unless I mean one of the senses in particular.

Some kinds of faults may not be detected by branch coverage. Consider a test-at-the-top loop that is intended to sum up the elements of an array:

```
sum = 0;
while (i > 0)
{
 i = i - 1;
 sum = pointer[i];          /* Should be += */
}
```

This program will give the wrong answer for any array longer than one. This is an important class of faults: those that are only revealed when loops are iterated more than once. Branch coverage does not force the detection of these faults, since it merely requires that the loop test be true at least once and false at least once. It does not require that the loop ever be executed more than once. Further, it doesn't require that the loop ever be skipped (that is, that i initially be zero or less). *Loop coverage* [Howden78] [Beizer83] therefore requires that loops be iterated zero, once, and "many" times.

Multi-condition coverage [Myers79] is an extension of branch coverage. In an expression like

```
if (A && B)
```

multi-condition coverage requires that A be true in some test, A be false in some test, B be true in some test, and B be false in some test. Multi-condition coverage is stronger than branch coverage. These two inputs

A == 1, B == 1
A == 0, B == 1

satisfy branch coverage, but do not satisfy multi-condition coverage.

Multi-condition coverage provides a good illustration of how coverage can indirectly reveal bugs. Consider this common C fault:

```
if (A > 0 && B = 1)
```

The second clause is incorrect. Branch coverage does not force detection, since

A == 1, B == *anything*
A == 0, B == *anything*

will satisfy it. However, multicondition coverage is impossible to satisfy because of the fault - a fact that will be revealed when coverage results are examined. A similar observation was made by [Holthouse79].

Using Coverage

You would expect a thorough test suite to have very high coverage. After all, every branch in the program should correspond either to some system feature or some special case in the implementation, else why does it exist? Since any perfect test suite will test all the features and special cases, it must exercise all the branches.

However, high coverage alone does not necessarily mean the test suite is thorough. The "fault of omis-

sion" is a common type of bug ([Basili84], [Glass81], [Ostrand84]). It is exemplified by this code:

```
a = b / c;
```

which should read

```
if (c != 0)
        a = b / c;
```

Since you are testing the incorrect program, a branch coverage tool cannot tell you that you need a test where c=0 - it can't measure the coverage of a branch that doesn't exist. So a test suite that achieves perfect coverage may still not be good enough.

Using coverage as a test design technique will reduce your chance of finding this important class of bugs. Coverage should instead be a way of evaluating your existing tests, finding incompletenesses, and using those incompletenesses to focus attention on weaknesses in your test design. It is an incremental improvement to your normal design process. Here is what I do when GCT shows me a line like

> "lc.c", line 271: if was taken TRUE 0, FALSE 28 times.

(1) I ask if the branch is impossible (as, for example, in a sanity check or automatically generated code). If so, I ignore it.

(2) I ask what feature corresponds to this branch. Does it involve handling of quoted characters? Resource exhaustion? Collision handling in a hash table? I now know that that feature has been undertested in at least one way.

(3) If it's been undertested in one way, there's a good chance it's been undertested in other ways. I redo the design work I should have done in the first place, without worrying about the branch. Good tests will force the branch -- what I'm after are the tests that won't cause any increase in coverage but might find bugs. These are typically tests for omitted special-case code. This sounds like a lot of extra work, but it's rarely much harder than just satisfying the branch.

(4) I will sometimes decide that the branch is too much trouble; for example, it may be that the true case would be very hard to induce, the program's actions in that case are "obviously" correct, and I can think of no other related tests. I rarely choose to skip testing, though. In the past, I've almost done so, decided I'd test anyway, and found a bug.

What is the additional cost of measuring coverage and redesigning? In an experiment, I found that it increased the total cost of test design by 4% [Marick91]. Using coverage in my day-to-day testing shows a similarly small cost.

An Example

I spent roughly a month testing the essential core of GCT. The overall coverage of those tests was 86.44%. Of the 129 uncovered coverage conditions, 74 were impossible, 17 were from deliberately untested temporary code, 24 were due to a major feature that had never had regression tests written for it, and 14 corresponded to 10 untested minor features.

Completely testing the major feature required 4 hours of work for design and implementation. This testing discovered one bug. Note, though, that a test designed solely to achieve coverage would likely have missed the bug. (That is, the uncovered conditions could have been satisfied by an easy and obvious - but inadequate - test.)

Testing the minor features required 2 hours. Branch coverage discovered one pseudo-bug: dead code. A particular special case check was incorrect. It was testing a variable against the wrong constant. This check could never be true, so the special case code was never executed. However, the special case code turned out to have the same effect as the normal code.

After fixing the bugs, I reran the entire test suite to see if all expected coverage conditions had been satisfied. I found that one had not. A typo caused the test to miss checking what it was designed to test. Finding this test error and correcting it took 1/2 hour. (Running the test suite took none of my time, since

it's completely automated.)

This work - 6.5 hours - was a small part of the total testing effort. Of the two bugs, one could never cause a failure, and the other would cause a failure only in rather obscure C programs, so the effort of finding them was arguably not worth it. However, the knowledge that I had a thorough test suite made me quite confident that GCT produces correctly instrumented C code, and that confidence has been borne out.

Of course, the time spent was a small fraction only because the initial tests were so thorough. What do you do if your tests leave many test conditions unsatisfied -- 40%, say? You might decide to spend the time to achieve complete coverage. However, there's a considerable danger that an effort to add 40% coverage will degenerate into a blind writing of tests that satisfy coverage. Faults of omission will be missed.

It would be better to improve your test process so that each new testing effort naturally results in higher coverage than the one before it. But this begs the question of *how* you improve your test design so that these higher coverages are inevitable. Coverage results - more particularly, careful thinking about the types of coverage that are missed - can lead to improvements in test design. However, remember that coverage is only an approximation of effectiveness. For process improvement, you'll do better to deal directly with effectiveness by examining what your testing misses. This is the next topic.

2. Designing tests based on experience

This section describes several examples of how you can use the bug history of a system to learn how to design better tests and better measures of their quality. The basic process is always the same:

(1) LOOK at the bugs that testing missed.

(2) ASK why they were missed.

(3) INVENT an improvement to the testing process so that they're less likely to be missed next time.

Simple faults that allow coverage

Programmers often misuse relational operators. They write code like

```
if (A < 5)
  return 1;
else
  return 0;
```

when they should write

```
if (A <= 5)
  return 1;
else
  return 0;
```

In this particular case, the only value of A that causes a failure is A=5. For all other values, the incorrect program yields the same value as the correct one. Because testers know that this type of misuse is likely, testing "boundary conditions" is one of the basic test design rules [Myers79].

GCT can measure whether we've tested boundary conditions. Just as it can measure whether the branch was taken true or false, it can measure whether A=5 during the evaluation of the if's test. This type of coverage is called "relational operator coverage", and I've found it useful in testing. For example, after writing tests for part of GCT, I found that one relational operator was not covered. It turned out to be impossible to cover because the code was wrong. On my development machine, it caused no failure. On another machine, GCT might not have worked at all. That's a pretty severe bug; finding it paid for all the time spent on coverage.

Relational operator coverage is a subset of *weak mutation coverage* [Howden82]. GCT measures other types of weak mutation coverage, but relational coverage is the only one I'm sure is worthwhile.

You should use relational operator coverage just like other coverage: design your tests first, then measure coverage, then upgrade your tests.

Widely Applicable Test Condition Catalogs

"Testing for inequality" is included in every programming language because it's something every programmer wants to do. It's a universal cliche. Many other cliched actions are not built into every language. Some languages implement them, others provide versions of them as library routines, and some leave it to the programmer to implement them from scratch. Here are some examples:

(1) traversing an array.

(2) searching an array.

(3) sorting a list.

(4) using a temporary file.

The same is true of data types. All programming languages have integers; most have floating point numbers, a few have lists, a very few have hash tables. A data type gets put into a language (or defined by users) when enough people have found the abstraction useful - when it becomes a cliche.

The important thing about cliches is that they are mis-implemented or misused in cliched ways. We've already seen how the relational operator cliche is typically misused. That misuse often becomes a cliched mis-implementation of traversing an array - how many such traversals contain off-by-one errors? How many list operations fail on circular lists? How much floating point code uses exact equality when inexact is better? How many routines that take pointers fail when given a null pointer, or when given two pointers that happen to point to the same object?

Cliches can be catalogued, as in textbooks on algorithms. So can common faults in those cliches. [Johnson83] and [Spohrer85] are collections of such faults for novice programmers.

For purposes of testing, a fault catalog isn't quite what we want. For A<5, we don't care that A<=5 is the correct program - let the person fixing it worry about that. All we really need to know is that A=5 is a good *test condition* to use to find misuses of relational operators.

The distinction between faults and test conditions is useful. For elementary cliches like <, it's easy to exhaust the plausible alternatives. This is much more difficult when considering all the ways "searching" could be misused or incorrectly implemented.[2] Fortunately, relatively few test conditions suffice to detect almost all these faults, because people make cliched mistakes (and because one good test condition may detect many possible faults).

To find these few test conditions, you examine bug reports, asking what cliche the bug involves (if any) and what test condition would have detected it. Then you write that information down in a *test condition catalog*. [Marick92a] is such a catalog. I started it by writing down test conditions that are common knowledge (such as boundary conditions) or that I've absorbed in my experience as a tester. I then looked for new cliches in introductory programming texts and reference manuals for higher-level programming languages like Smalltalk. When I found a cliche, I asked myself how I'd test it. After that, I augmented the catalog by examining bug fixes (mostly those posted on Internet bug newsgroups).

Here's an example of an entry, one for a program that searches. (There are more specific entries for special kinds of searches.)

- Match not found
- Match found (more than one element, exactly one match)
- More than one match in the collection
- Single match found in first position (it's not the only element) INLINE
- Single match found in last position (it's not the only element) INLINE

Note that these conditions apply whether the search is forward or backward.

The first three check for misuses of "searching". They apply whether the search is done by a subroutine or an inline loop. The first is used because programs that search sometimes incorrectly assume that the search

[2] See [Morell90] and [Howden87], though, for analytical work on larger code fragments.

will never fail. The second just ensures that you design a test where the program does search. The third checks for another common bug, where a program should search for several matches but incorrectly searches for only one.

The last two test conditions are only useful for an inline implementation: they check boundary conditions. Most faults they'll discover are the direct result of relational operator misuses, but it's inefficient to worry about details if you can find the faults without them.

Some of these conditions do not apply to all programs - for example, it may truly be impossible for a match not to be found. Some may apply to a particular program, but not be useful. For example, if the program only uses a first match, why try a list with more than one match? However, it's usually easy to include such a test condition in a test you're writing anyway, so it costs almost nothing to use it.[3]

An expert tester has a catalog very like this in his or her head. By writing it down, you make it available to novices and experts with imperfect memories. The result is an improvement in test effectiveness. The catalog will grow as new bugs prompt additions.

Application-specific faults

Of course, some bug reports will have nothing to do with commonly used cliches. Instead, they'll report bugs particular to your system. Many such bugs will be due to subtle interactions between subsystems. Others will actually be errors in the specification of the system, not its implementation.

There's certainly every reason to think about these bugs as hard as cliched bugs. You will be able to use some of them to build an application-specific test condition catalog (though these faults seem to be harder to categorize). At the very least, you'll discover what parts of the system are bug-prone, so you know where to test harder. You'll also discover how customers use the system, so you'll be able to make system tests more representative.[4]

You may also discover that this focusing of attention suggests not only test designs, but ways of measuring the quality of tests. For example, in multiprocessor systems with locking, locking errors are a major risk (because they often cause unacceptable failures). Three types of bugs are:

(1) A processor fails to acquire a lock before updating a data structure. If another processor is simultaneously updating the data, it is likely to be corrupted.

(2) A processor fails to release the lock. The system will likely grind to a halt as other processors wait for that lock to be freed.

(3) The system may deadlock when one processor acquires lock 1 and attempts to acquire lock 2, while at the same time a second processor acquires lock 2 and attempts to acquire lock 1. Neither can proceed until the other gives up its lock.

The conventional approach to testing for locking errors is stress testing. By making sure that a shared datum is heavily used, you increase the chance that locking faults involving that datum will manifest themselves. The problem is with "making sure" -- despite your attempt to stress a particular part of the system, all the processors might be bottlenecked in some other part. Perhaps they are all waiting on a lock that's letting only one at a time through to the part of the system you're testing - and therefore you're not testing its locking at all.

This danger motivated *race coverage*. Race coverage counts the number of times a routine has had two or more processors (or threads of execution) in it at once. Race coverage thus provides direct evidence of whether stress tests are actually stressing particular routines. Those routines that have not been "raced" can be targeted with new stress tests.

Note that race coverage - like most coverage - is an indirect measure. It is not measuring contention for data structures, only entries to routines. (This allowed much faster implementation and tolerable runtime overhead.) Nevertheless, it was useful for generating tests that found locking bugs of all types.

---

[3] Test design - how to most effectively combine test conditions into tests - is outside the scope of this paper.

[4] [Petschenik85] contains a useful discussion of this type of system testing.

A note on inspections

Suppose your program allocates memory but fails to release it. Such "memory leaks" typically have no effect on the output, so they're not visible during a test. You only see them when the program runs out of memory after running for many hours. This is an example of a fault difficult to detect with testing. These are typically failures to honor what [Perry89] calls "obligations". That is, one part of a program uses the result of another part of the program. As a result, it incurs an obligation to perform some action later. The bug is that it fails to fulfill the obligation - it fails to free the memory, or it fails to close the opened file. (For the particular case of memory leaks, there are specialized tools that can make the failures more visible during a test, but they do not solve the general problem of obligations.)

Such faults may best be detected by inspecting the code. An inspection catalog can be built up alongside the test condition catalog, and in the same way - by examining faults. Here's an example from the "freeing data" section of [Marick92b]:

- Are you freeing already-freed storage?
- Does the freed storage still have pointers to it, pointers that could still be dereferenced?
- Are you freeing storage that is not supposed to be freed? (Often, allocated with a routine that says "do not free returned value" in a code header.)
- Is it possible for the pointer being freed to be null?
  (Note: POSIX-compliant implementations of **free** accept null pointers. Not all implementations conform.)
- Are you reusing the contents of freed data?
  (Some versions of **free** overwrite some parts of the freed data; most do not.)

Such a checklist is easily incorporated into whatever type of code reads or code inspections you do now.

Checklist items may also be specific to your application. For example, after a couple of years of working on GCT, I know to apply this checklist after I add a new type of coverage:

```
* 1. Have only copies of temporaries been used?
* 2. Has the original value of temporary_id been freed?
* 3. Does the instrumented tree have the same type as the original tree?
* 4. If locals are set pointing to parts of a tree, is the tree still the same when
*    the locals are used (else they might point to the wrong place)?
* 5. Are locals set to values derived from the root of an instrumented tree?
* 6. Do the mapfile entries read from left to right?
* 7. Is SELF built into the resulting instrumented tree?
* 8. Is any temporary_id that is an initialized non-static declared
*    OUTERMOST? (See gct-temps.c.)
*/
```

(Note that this checklist is a comment at the head of the source file to which it applies, a convention I recommend.)

Detection and Prevention

Of course, you'd rather prevent faults than create them, find them, and correct them. The same examination of reported bugs that improves test design can also improve fault prevention. [Mays90] is an excellent description of this process. They report that examining faults and implementing preventative procedures costs only about 0.5% of total development effort, whereas they estimate a direct benefit of at least double and possibly triple the cost.

One type of fault prevention is simply to use test condition catalogs during development. Many of the faults those test conditions find are the results of unexamined assumptions; if the developer simply reads the test condition catalog and wonders if her code handles a condition, the fault may be forestalled. This of

course then raises the question of why you should bother writing the test.[5]

A prolonged discussion of this topic would take us far afield from ways of gradually improving testing. The answer, in brief, is that our fault prevention skills are as yet too shallow and too narrow.

(1) **shallow:** We're not good enough at preventing even those errors we think of. The developer may try to handle the condition, but get it wrong. The cost of testing is less than the risk of trusting prevention alone.

(2) **narrow:** There are too many errors we don't even try to forestall, because we don't think of them. Because testing is rather good at serendipitously finding bugs it wasn't "supposed" to find, it's a useful safety net for error prevention. This is especially true if the tests are made as complex as is tractable. Each should probe many test conditions at once. This increases the chance that you will accidentally probe some unsuspected test condition. [Knuth89] has a nice description of this approach applied to the TeX text processor.

As our prevention skills increase, they will reduce the need for testing. More importantly, the focus of testing will shift toward faults harder to prevent, especially design and specification faults caused by faulty or unquestioned assumptions.

3. Reducing Test Cost

It's not enough to design better tests; you must also reduce the cost of test construction.

The key to long-term reduction of testing costs is an automated regression test suite. Manual tests are simply too expensive to run often enough. Bugs are found long after they're caused, so the cost of diagnosing and fixing them is much higher. Further, bugs may not be found at all: if the tests are judged manually, by a person looking at the output to see if it's correct, many failures will be missed ([Basili87], [Myers78]).

While the long-term benefits of an automated test suite are enormous, the startup cost can be high. For example, testing individual routines in isolation (unit testing) is too expensive except for the most critical routines. You spend too much time writing test drivers and the stubs that emulate the subroutines the routine-under-test calls. Initial test development is bad enough; maintenance becomes a nightmare, and the unit tests are often abandoned.

To reduce the cost, separate test design from test implementation. The same unit tests can often be implemented using the system itself as the driver.

(1) The design calls for certain inputs to be delivered to the routine. Deduce what input to the system will cause that delivery.

(2) The design calls for certain expected results from the routine. From these, predict the expected results from the system, and use those results to judge the routine's correctness.

(Of course, the test design may be based not on individual routines, but on interfaces to larger subsystems. If test design is backed up with careful use of coverage, we may not lose much in effectiveness. This discussion applies to any instance of testing some body of code via the system that contains it.)

This approach usually makes at least part of the test suite easier to automate. There are two potentially serious problems:

(1) You may make a mistake designing inputs. Perhaps your chosen input is a special case for the system, handled by special-case code and never delivered to the routine at all. This is the lesser of the two problems, especially since coverage often points to it.

(2) It may be hard to determine correctness of an interior routine from correctness of the system. For example, suppose the routine produces a complicated linked list structure and the system produces 0 or 1. Information is lost in that transformation ([Voas91a], [Voas91b]). A correct and incorrect linked list structure may both produce 1, so you can't test the interior routine via the system.

This difficulty is usually because systems are not designed to be testable - they are not designed to be easy to force into known states, and they do not provide internal results in a way that allows them to be compared to expected results. Of course, redesigning systems for testability is hardly the sort of gradual improvement this paper is about. What is needed is a way of "growing" the testability of the system, so

[5] Hamlet has considered related questions. See, for example, [Hamlet92].

that every month you have a better prototype of a truly testable system. This can be done by taking advantage of what the programmers are already doing:

They're writing debugging code.

Consider the GNU Make program. It maintains an internal hash table of files. The programmer wrote debugging code to print out the entries in the hash table. The output looks like this (edited for conciseness):

```
# Files

a::
# Command-line target.
# Implicit rule search has not been done.
# Implicit/static pattern stem: ''
# Last modified Mon Jul 27 10:37:55 1992 (712251475)
# File has been updated.
# Successfully updated.
# 18 variables in 23 hash buckets.
# average of 78.3 variables per bucket, max 2 in one bucket.
```

This is almost what you would need to test the hash table manipulation routines, but there are several problems:

(1) The file's location in the hash table is not printed (in particular, we want output that shows how hash table collisions have been handled).

(2) Variable values are translated into text useful to the debugger (such as "successfully updated"). For testing, we'd prefer to see the exact values.

(3) The date in the output will cause spurious differences when expected output is compared to actual output.

Revising the routine to solve these is simple. An example of testing output is:

```
aa:: (at bucket 194, bucket pos 0, duplicate pos 0)
#  (update_status = 1) (allow_dups = 1)
aa: (at bucket 194, bucket pos 0, duplicate pos 1)
#  (update_status = 2) (allow_dups = 0)
aa: (at bucket 194, bucket pos 0, duplicate pos 2)
#  (update_status = 3) (allow_dups = 0)
# 2 files in 1007 hash buckets.
# average 0.2 files per bucket, max 1 files in one bucket.
```

The hash table routines are now testable at very little expense.

Surprisingly, some organizations discourage the inclusion of debugging code in the finished product. The code may then be unavailable for testing; at best, it will be less useful (because it's "throwaway" code in the first place). Eliminate this policy.

The same organizations probably also resist "cluttering up the code with testing crud". You often don't want testing code in the executable, usually for performance reasons. (However, if the testing code is modified debugging code, that inclusion may save a lot of time when answering phone calls from the field.) But there is rarely a reason not to include testing code in the source and compile it out in the delivered version. There are two arguments you may face:

(1) "We should test what we ship". This is absolutely true - for the last two runs of the test suite. During earlier development, there is very little risk in using a testing version. The small risk of the testing code obscuring a bug, causing it to be discovered late, is counterbalanced by the other bugs that will be found because more tests are feasible.

(2) "Too much testing code makes the system unmodifiable." If your testing modifications cause every third line to be

#ifdef TESTING

you should certainly rethink them. Testing changes should be as modular as any other enhancement. But what this statement can also mean is that modifications that had to happen anyway are now visible. Whenever there is test support code, changes to the system can wreak havoc with it, causing wholesale reworking. But that havoc is usually invisible to the person making the change. When the test support code is inside the system, it causes more work for that person. That the *total* work required to effect a change is reduced will be no consolation. This is a situation requiring considerable tact. The greater the organizational separation between testing and development, the greater the tact required.

As converting debugging to testing code becomes more common, make it simpler by providing conventions, support code, and examples. Keep in mind that the purpose is not to make the debugging code harder to write, but to make it more adaptable for testing. As low-cost efforts prove their worth, move toward a more complete in-system debugger and test harness. I'll discuss an example.

I was the leader of a team of programmers porting a version of Common Lisp from an obscure workstation to an obscure minicomputer. The workstation was of the now-unfashionable type that allowed user-programmable microcode. That is, it did not exactly have a "native" instruction set; instead, each high-level language compiler could assume a machine language tailored to its requirements. The Common Lisp compiler assumed an instruction set much like that of the MIT Lisp Machines. For example, the Lisp Cons instruction was also a machine language instruction, implemented in microcode. The garbage collector was also implemented in microcode.

The first step of the port was to emulate all the microcoded instructions by C and assembly code on the minicomputer, turning it into a Lisp machine emulator. Garbage collection was the hardest instruction; I began with it. I implemented debugging code that allowed me to initialize the state of the imaginary Lisp machine, take a snapshot, garbage collect, take another snapshot, and compare the two states. For testing, all that was required was to allow the garbage collector debugger to be driven by a script. The remaining instruction emulations were debugged and tested using extensions of what I'd already done for garbage collection, mainly by adding more flexibility in setting up the starting state. As the system grew closer to completion, more debugging features were added as needed. (In particular, after every difficult bug fix, I asked myself what new debugging command would have made the problem more obvious. Then I added that command.) Because this increasingly capable harness was designed to be script-driven, each new debugging feature was also a new testing feature, if appropriate.

Of course, in other systems, such an elaborate test harness is not required. Almost all of GCT's tests use special testing code, but most of them depend only on one 10-line segment. Some of the others use relatively simple subsystem-specific routines that print out a portion of GCT's internal state. In a few cases, parts of GCT have been unit tested with stubs and drivers.

At some point, a gradual approach may be insufficient; eventually, you may find that you need an overhaul of the system to test as thoroughly as you like. But the costs are modest, the benefits immediate, and success at small changes can give you the experience you'll need to justify a more designed, less "grown", test harness.

REFERENCES

[Basili84] V. Basili and D. Weiss, "A Methodology for collecting valid software engineering data", *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 728-738, November, 1984.

[Basili87] V. Basili and R.W. Selby. "Comparing the Effectiveness of Software Testing Strategies". *IEEE Transactions on Software Engineering*, vol. SE-13, No. 12, pp. 1278-1296, December, 1987.

[Beizer83] Boris Beizer. *Software Testing Techniques.* New York: Van Nostrand Reinhold, 1983.

[Glass81] Robert L. Glass, "Persistent Software Errors", *Transactions on Software Engineering*, vol. SE-7, No. 2, pp. 162-168, March, 1981.

[Hamlet92] Richard Hamlet. "Are We Testing for True Reliability?". *IEEE Software*, July, 1992.

[Holthouse79] Mark A. Holthouse and Mark J. Hatch. "Experience with Automated Testing Analysis". *Computer*, Vol. 12, No. 8, pp. 33-36, August, 1979.

[Howden78] W. E. Howden. 'An Evaluation of the Effectiveness of Symbolic Testing'. *Software - Practice and Experience*, vol. 8, no. 4, pp. 381-398, July-August, 1978.

[Howden82] W. E. Howden. "Weak Mutation Testing and Completeness of Test Sets". *IEEE Transactions on Software Engineering*, vol. SE-8, No. 4, pp. 371-379, July, 1982.

[Howden87] W.E. Howden. *Functional Program Testing and Analysis*. New York: McGraw-Hill, 1987.

[Johnson83] W.L Johnson, E. Soloway, B. Cutler, and S.W. Draper. *Bug Catalogue: I*. Yale University Technical Report, October, 1983.

[Knuth89] Donald Knuth. "The Errors of TeX". *Software - Practice and Experience*. Vol. 19, No. 7, July 1989, pp. 607-686.

[Marick91] Brian Marick. "Experience with the Cost of Test Suite Coverage Measures". *Pacific Northwest Software Quality Conference*, October, 1991.

[Marick92a] Brian Marick. *Test Condition Catalog*. Testing Foundations, 1992.

[Marick92b] Brian Marick. *A Question Catalog for Code Inspections*. Testing Foundations, 1992.

[Mays90] R.G. Mays, C.L Jones, G.J. Holloway, and D.P. Studinski, "Experiences with Defect Prevention", IBM Sys. J., Vol 29, No 1, 1990.

[Myers78] Glenford J. Myers. "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections". *Communications of the ACM*, Vol. 21, No. 9, pp. 760-768, September, 1978.

[Morell90] L.J. Morell. "A Theory of Fault-Based Testing". *IEEE Transactions on Software Engineering*, Vol. SE-16, No. 8, August 1990, pp. 844-857.

[Myers79] Glenford J. Myers. *The Art of Software Testing*. New York: John Wiley and Sons, 1979.

[Ostrand84] Thomas J. Ostrand and Elaine J. Weyuker, "Collecting and Categorizing Software Error Data in an Industrial Environment", *Journal of Systems and Software*, Vol. 4, 1984, pp. 289-300.

[Perry89] Dewayne E. Perry. "The Inscape Environment". *Proceedings of the 11th International Conference on Software Engineering*, pp. 2-12, IEEE Press, 1989.

[Petschenik85] Nathan H. Petschenik, "Practical Priorities in System Testing". *IEEE Software*, September, 1985.

[Spohrer85] J.C. Spohrer, E. Pope, M. Lipman, W. Scak, S. Freiman, D. Littman, L. Johnson, E. Soloway. *Bug Catalogue: II, III, IV*. Yale University Technical Report YALEU/CSD/RR#386, May 1985.

[Voas91a] Jeffrey Voas, Larry Morell, and Keith Miller. "Predicting Where Faults Can Hide from Testing". *IEEE Software*, March 1991.

[Voas91b] Jeffrey Voas. "Factors That Affect Software Testability". *Pacific Northwest Software Quality Conference*, October, 1991.

COMPLETE TESTING AND PROGRAM ABSTRACTIONS

William E. Howden
Yudong Huang
Suehee Pak

Software Engineering Laboratory
University of California at San Diego
La Jolla, CA 92093

Abstract

We describe how the systematic use of abstractions can be used to construct exhaustive test sets, and to characterize the assumptions upon which a set of program system tests is said to be complete.

Biographies

William E. Howden is a Professor of Computer Science at the University of California at San Diego. He has written many papers, authored one and co-edited two books in computer software testing. He has served on the editorial board of IEEE Transactions on Software Engineering and has been program chair of the International Conference on Software Engineering. His current research includes the ASPECTS abstract testing and reverse engineering project, and the QDA-Ada project in which a practical Ada verification system is being developed.

Yudong Huang is a Ph.D. student in the Department of Computer Science and Engineering at the University of California at San Diego. He received the B.Sc. and M.Sc. degrees in Computers and Automation from the Polytechnic Institute of Bucharest, Romania. He is currently working on testing methods for expert systems.

Suehee Pak is a Ph. D. student in the Department of Computer Science and Engineering at the University of California at San Diego. She received the B.A. degree in German Language and Literature, and the B.Sc. degree in Computer Science and Statistics from Seoul National University in Seoul, Korea. She received the M.Sc. degree in Computer Science from the University of California at San Diego. Her current research is on abstractions and reverse engineering methods for large data processing systems.

1 INTRODUCTION

The development of practical verification methods continues to be one of the difficult unsolved problems in software engineering. Over the last two decades a variety of different methods have been proposed. Those that are practical, in the sense that they can be easily used without impossible overhead costs, are usually inconclusive in that they do not directly prove that the software is any more reliable once they have been successfully applied. Those methods that are conclusive are usually impractical.

Practical methods include various kinds of test coverage approaches. The most common is statement coverage, in which it is required that each statement be executed on at least one test. A variety of extensions to this idea have been proposed. They include data flow coverage measures in which it is required to test each pair of statements together on at least one test, if there is a flow of data from the first to the second [1,2]. Others include path coverage and path segment coverage measures [3,4]. All of these methods have a certain intuitive appeal, but all that can be concluded when they are used is that every statement, or every path segment or some other elementary program component has been exercised. From this it is not possible to conclude that the program is correct or reliable, or what kinds of faults have been eliminated.

Examples of more conclusive testing methods include algebraic [5], weak mutation [6] and mutation [7] testing. The general idea in these approaches is to define a set of tests to be complete if they will provably detect all of the faults in some fault class. The approach may be useful for unit testing, but it appears to be impractical for larger program units.

Perhaps the most conclusive verification method is proofs of correctness. This method is conclusive in the sense that it proves that a program has certain well defined functional properties, described in a formal specification. Its disadvantage is that proofs can become hopelessly bogged down in details. They are difficult to construct and may themselves have errors. Proofs may be a useful way of verifying the design of an algorithm, but they are often impractical for programs, especially above the unit program level.

The ideal situation, in testing, would be to be able to test a program exhaustively over its entire input space. This is, of course, in general impossible. What would also be ideal is if it were possible to divide up the input space of the program into equivalence classes, so that if the program were correct on one element of the input class, then it would be correct on all members of the class. If the number of classes were small enough, exhaustive testing would be possible. A similar approach would be to introduce abstractions into the code in such a way that we could test the program at some abstract level, at which there were only a small number of abstract program paths. If we could then carry out an abstract evaluation of each path we would be able to exhaustively test the program. This is the approach that is examined in this paper.

2 ABSTRACTIONS AND TESTING

Different kinds of abstractions can be used to simplify a program. The most common kind is a subroutine or procedure module. When the module is considered within the context of the whole program, it is represented by its abstract function name, resulting in an abstract program.

The construction of a module may involve preconditions on its input data, properties they are expected to satisfy in order for the module to have its listed abstract effect. These can be types, in which case there is the expectation that the module's parameters will contain certain types of data. They can also be properties specified in some object property language.

In previous studies, it was determined that it is useful for the purposes of testing to identify sections of code in a program that are not identified as separate modules, but correspond to readily identifiable abstract functions[8]. These implicit modules may also have (unwritten) precondition expectations that, like those for explicit modules, must be satisfied in order for them to have their expected abstract functional effect.

In addition to functions that correspond to modules or explicit sections of code, programs may contain functional threads that are blended in with other functional aspects of the program. These may correspond to *slices*, or the subparts of the program that compute the values of a subset of a program's variables.

We have found it useful to identify two kinds of abstractions: *intensional* and *extensional.* An intensional abstraction is one for which there exists a formal, complete definition. For example, the abstract function Matrixmultiply(A,B,C) can be given a formal definition. An extensional aspect is known by example. For example, the abstract concept of *initialization* can be associated with many different kinds of parts of programs. There is no exact definition of what it means, but we can point to some example, and recognize it as initialization code.

Extensional abstractions are important for several reasons. One is so that general terms can be used for describing the properties of a variety of different kinds of programs, without being uniquely defined by any one of them (or by its specifications). Another has to do with the formation and use of abstractions during the design and programming process. These are made up on the fly and the idea is that some aspect of the program will be given some abstract name having an intuitive meaning in order to better understand the program and keep track of what is going on. The meaning of these abstractions becomes progressively defined as they are used to document different sections of code.

The use of abstractions may require the use of an *abstract test oracle.* In ordinary testing methods, we must have an oracle that can be used to determine if the output (or some other behavior) produced by some input is correct. If the abstraction approach is used, we must have an oracle that can answer the question "if the following abstractions are made, is this abstract behavior correct?"

Abstractions can occur in a variety of ways, and may be different in different kinds of programs and systems. In the following sections we will study abstractions and testing for three different kinds of programs: low level process control programs written in assembly language, data processing programs, and expert systems.

3 ASSEMBLY LANGUAGE AND PROCESS CONTROL PROGRAMS

3.1 Application area

In the QDA(Quick Defect Analysis) project we studied the functional modules in the avionics operational flight program (OFP) for the Navy's AV-8B aircraft. This program is written largely in assembly language, with the data declarations in CMS2. It is divided into

functional modules, each of which consists of a large number of small subroutines. The modules communicate using a global data base.

3.2 Program comments

Like most well written assembly programs, the OFP is heavily commented. Each of the subroutines has a header that describes the abstract functional effects of the routine, and the expected properties of the input to the routine. In addition, the expected output is abstractly described. The abstract functional effects are often described extensionally, using words whose meaning intuitively corresponds to what the code computes. The exceptions are the mathematical subroutines, which are named with formally defined mathematical concepts. Suppose that F is some extensional abstract function name associated with a routine R, and C is the set of conditions on the input. Then the documentation associated with the routine is thought of as declaring that, provided that C is satisfied, R is an example of the abstraction F.

Most of the comments in the code are concerned with properties of intermediate states of the program. They describe the abstract functional effects of sections of code or individual statements. In examining these comments we found that many of them fall into two categories: *hypotheses* and *events*. Events describe the abstract effects of some statement or piece of code. Hypotheses describe expected properties of the system. They are like working hypotheses, formed by the programmer during programming, and on the basis of which certain coding decisions are made. Hypotheses and events can be thought of as fitting together in the following way. When a programmer is working at some location in the code, the goal is to perform some abstract action. Assumptions are made about the state of the system at that point, and on the basis of the assumptions code is written that will produce the desired effect. We can then say that the written code produces the event provided that the hypothesis is correct.

3.3 Object properties and the QDA comments language

Event and hypothesis comments in the OFP usually concern properties of objects. Objects may include variables and data structures in the OFP data base, registers, the clock rate, and more abstract objects created by the programmer during commenting. Because of this, an abstract description language was defined in which the programmer can document the state of a program in terms of logical expressions whose basic terms are of the form

x is P, x iss P, x nis P, and x niss P,

where x is an object and P is a property or list of properties. The first term means that x has the property (or property list) P, the second means it has the property P and no other, the third that it does not have the property P and the fourth that its properties are not exactly the property (ies) P, but some other property or list of properties. Property expressions are used to construct hypotheses and are of the form

?property-expression.

Events are or the form

!property-expression.

In addition to the hypotheses, events and specifications described above, QDA also includes various other kinds of comments that were found to be useful.

3.4 Abstract program execution and testing

A program can be executed at some abstract level by executing its events. Abstract execution of some program path starts with an empty program state. Then events that encountered along the path are executed to produce the desired effect on the state.

Property expressions can also be used to give a set of abstract specifications for a program and are of the form

input: IPE1, IPE2, ... IPEn
output: OPE1, OPE2, ... OPEm,

where IPEi and OPEi are all property expressions. The specification means that whenever the routine is called, one of the input expressions must be satisfied, and when it terminates, one of the output expressions must be satisfied.

Abstract testing of a QDA annotated program is carried out on a routine by routine basis, and involves "executing" the events along program paths to form abstract output expressions. These are then compared with the routine's output specification to see if they are correct.

In order to apply some event abstraction it is required that any hypotheses that were made while the program was being commented, on the basis of which the event was asserted, must be true. This can be done during abstract program testing. Each time an hypothesis is encountered along some program path it is checked against the current abstract program state. If it cannot be satisfied in the current program state, an abstraction error message is generated.

QDA testing can detect two kinds of errors. One occurs when the abstract output produced by some test does not satisfy an output specification. The second is that there is an abstraction specification error because some hypothesis is not satisfied along some program path. The first is the usual kind of error discovered by testing, but at an abstract level. The second is similar to a type error, when a subroutine is called with actual parameters that are not of the expected types, i.e. the types of the formal parameters. The differences are as follows. Types hold over a well defined fixed scope. QDA object properties can be associated with objects at arbitrary points in the code, and de-associated whenever no longer appropriate. Similarly, hypotheses can be inserted at arbitrary points when needed as assumptions to support the use of some abstract event, documenting the effect produced by some piece of code.

3.5 Experience with QDA analysis

Although QDA has been described as an abstract testing tool, its most powerful feature was found to be its use in checking abstraction hypotheses. Hypotheses are used at the beginnings of each documented code segment to record assumptions about what has occurred in previous segments and checking these hypotheses had the effect of checking the validity of the abstract reasoning that was used during code construction. This is where we discovered errors, rather than by looking at the abstract test output for a routine. Hypotheses which have been described as abstraction conditions, can also be thought of as intermediate state assertions, so that checking them corresponds to an abstract testing process in which intermediate as well as final state values are checked.

One of the criticisms of QDA has to do with the validity of the comments in describing the code. The program may be found to "test out" at the abstract level, but it may be incorrect

because the abstractions are incorrect. There are two ways in which this can happen. The first is when a comment is missing, when code occurs that changes the abstract value of some abstract object but this is not documented. This problem can be ameliorated using an analysis tool that can determine when changes are made to objects about which assumptions occur but for which there are no corresponding abstract property description events. The second kind of error is when a comment is wrong, that it declares that one kind of abstract object property changing event has occurred and some other property change should have been documented. This cannot be dealt with by QDA since in QDA the meanings of abstractions used in comments are extensional in the sense that the concrete properties established by the code are assumed to be examples of the abstraction; the code in fact defines (at least partially) the meaning of the abstraction so that by definition the abstraction cannot be considered incorrect.

A QDA abstract testing tool was built and applied to the functional modules of the current operational version of the AV-8B OFP. Many dozens of coding and documentation faults were found. A concise summary of the results of the QDA empirical studies can be found in [9]. Examples include the following.

3.6 Examples of errors discovered using QDA

Subroutine interface fault. The routine MSLMT in the math module expects registers R1 and R2 to be set to high_limit and low_limit. In the routine DSLRT there is a path along which these register flavors are not established before the call is made. In verifying the input specification for MSLMT, QDA discovered that there was a calling state in DSLRT in which this flavor expectation was not satisfied.

Intermediate flavor fault. At one point in the code, the original program comments indicate that a program jump should be made if GIREL>1. This led to the introduction of a QDA hypothesis

?GIREL is gt_1

at the target location of the jump. This assumption is presumably a prelude to the abstract operation to be performed at that or a later location. The code was then examined to insert events documenting the effects of the jump instruction. On the branch that takes the jump the instruction that is used is such that the event

!GIREL is lt_2

was inserted. This leads to the detection of the hypothesis error.

4 DATA PROCESSING PROGRAMS

4.1 Application area

The success of the QDA project encouraged us to attempt to use analogous methods for other kinds of programs. One area under investigation is large data processing programs. These are typically programs consisting of hundreds of thousands of lines of code. Like the OFP analyzed in the QDA project they often involve large global data bases so that even though they are highly modularized from a functional point of view, they are not as modularized with respect to data definition. In addition, some functions involving control aspects of the code may "thread" their way through many if not all of the modules. A tool

called ASPECTS is currently under construction that will allow abstract testing of Cobol programs.

4.2 Abstract operations

There are several basic concepts used to develop the ASPECTS approach to abstractions that are different from those used in QDA. One difference is the abstraction language, which is functionally oriented, as opposed to state oriented. Abstract operations in ASPECTS correspond to the assignment of an abstract value to a variable, as opposed to changes in a list of object properties. Abstract values are represented using functions and function composition. A related difference is the use of symbolic evaluation for doing abstract testing. Symbolic testing is an idea that was first described in [4] and has reappeared a variety of times since then (e.g. [10]). The advantage of symbolic evaluation is that abstract, undefined data, predicates and functions can be incorporated into the program evaluation process in such a way that abstractions and concrete operations can be intermingled with each other. This makes it possible to test code with embedded abstractions, as opposed to the QDA approach in which it is necessary to abstract out all code, and test entirely at the abstract level.

Another feature of the ASPECTS work is the identification of two broad classes of abstractions: problem domain abstractions and logical abstractions. Problem domain abstractions are concepts and ideas that are unique to the problem or problem class at hand. The abstractions are often inherently extensional, and are introduced into the abstract testing process by the programmer who recognizes the occurrence of these abstractions. Logical abstractions are simplifications that can be used to replace one set of abstract test output with another, simpler set. Logical abstractions are intensional and are defined by the rules of logic that make one logical expression equivalent to or implied by another.

A third feature of the ASPECTS work is the use of executable abstract test output. This feature is shared with QDA. Abstract test output in QDA corresponds to object property expressions. As we saw above, these can be both hypotheses or events. The semantics of event execution in QDA makes it possible to treat these expressions as events and execute them. ASPECTS differs from QDA in that abstract test output is represented as sets of pairs (C,V) where C is a predicate expression on symbols representing the initial values of variables, and V is a set of value pairs of the form (x = f(y)) where x is a variable, f is a functional expression, and y is an input variable value. In the terminology of symbolic evaluation, C is a path condition and V is a symbolic value output set. Abstract (symbolic) test output can be executed as follows: simply add the symbolically evaluated predicates to the path condition and interpret the symbolically evaluated functional part as a set of assignment statements.

The substance of most of the work in ASPECTS involves the development of problem domain and logical abstraction methods. The idea is that the programmer will introduce problem abstractions into the code which isolate some parts of the program and suppress others, replacing them with simple abstractions, so that the resulting program can be exhaustively tested over symbolic data. Logical abstractions are also applied to simplify and reduce the abstract test output produced from the abstracted program.

4.3 Problem domain abstractions

Several kinds of problem domain abstraction techniques are used in ASPECTS. One kind is derived from QDA, and depends on the use of comments. The comments language is different from that developed in the QDA project, reflecting the fact that we are using

different abstract evaluation semantics. There are two basic kinds of comments. The first consists of equalities of the form

$$!x = f(y)$$

where x and y are variables and f is an abstract function. When this is encountered in a program it is treated as an abstract, or symbolic assignment statement. The other kind of basic comment is of the form

$$!P(x)$$

where x is a variable and P is an abstract predicate. When this is encountered during symbolic evaluation, it replaces the previous path predicate that was generated from the code.

The above comments have the effect of implicitly replacing some section of code with an abstract program fragment. The equality comment overrides and replaces the previous assignment to x. The predicate comment overrides and replaces the conditional part of some statement.

In some cases we found it desirable to introduce more drastic abstractions in which more complex pieces of code are replaced with more complex abstractions. Facilities are included that also allow this.

4.4 Logical abstractions

In addition to our research on problem domain abstractions, work has been completed on logical abstractions. Since the purpose of logical abstractions is to reduce the number of different abstract test output cases that are generated, they can be more effective if they can be incorporated into the abstract testing process in such a way that they perform their simplifications before and while test output is being generated instead of after. We have identified two general kinds of logical abstraction procedures. The first can be used with loops and the second with branching structures. Both kinds depend on some form of dependency analysis, and can be used to perform reductions on a program graph during abstract program execution. Because of their graphical nature we earlier referred to them as *structural abstractions* [11].

Loop abstraction. If the programmer fails to abstract out a loop, or abbreviate it with a small finite path set, then abstract symbolic testing may result in the generation of an infinite number of paths (only *may* since if the evaluator has the ability to do infeasible path detection then it may be able to recognize situations where only a finite set can logically occur). In order to alleviate this situation without requiring action on the part of the programmer it is desirable to have some way of automatically generating loop abstractions.

In one approach that we have examined, the abstract program interpreter is assumed to have a way of determining if an assignment in a loop could possibly assign an infinite set of different values to a variable. The value set could be infinite, for example, if the assignment is of the form

$$x := x + y$$

where the initial values of x and y are either symbols or numeric values. If an assignment could result in an infinite number of values, then an abstract value of the form

F(x0,y0)

is formed, where x0 and y0 stand for the symbolic input values that are used to construct the values of x (in the above case they will be the initial values of x and y). This abstract value is then used for the value of x and stands for "the value computed for x" after completion of the loop.

Techniques have been developed for both scalar variables and data structures. The resulting abstractions involve functions that can be defined using functional equations.

The development of loop abstractions is an active area of research, and additional work needs to be done to determine the facilities required for each of the different kinds of situations that can arise. Our approach has been to characterize different kinds of loops, identify the kinds of abstract test output that would be useful for those kinds of loops, and to develop abstraction methods that would produce those kinds of abstract output.

Conditional structures abstraction Two kinds of logical conditional structures abstraction have been developed: *value* and *condition redundancy*. In the first we reduce abstract test output by eliminating separate test outputs that have the same value parts, using disjunction to replace them with a single test output case. Suppose that we have some abstract test output containing the two cases

(condition-C1, values-V) (condition-C2,values-V).

Then this can be reduced to the single test output

(condition-C1 or condition-C2, values-V).

The second kind of redundancy occurs when two test outputs have the same condition part, but different value parts. It can be used to replace two test outputs of the form

(condition-C, values-V1) (condition-C, Values-V2)

with the single abstract test output

(condition-C, Values-V1 union values-V2).

Value redundancy abstraction is particularly important in dealing with exponential blow-ups that occur when a program consists of a sequence of n conditional substructures, each of which is independent. In this case there are an exponential number of program paths and hence an exponential number of abstract test outputs. Value redundancy can be used to reduce this to a set of only 2n instead of 2**n outputs. In the case where the substructures are not completely independent, problem domain abstraction may be used to abstract out dependencies.

4.5 Abstract testing strategies

Abstract testing can be done with respect to a single output variable, a subset of the program's output variables, or with respect to the whole set of a program's output variables all at the same time. The choice will affect the volume of test output in the following way. If a set of variables is highly interdependent, then each variable may receive a different value for each path condition. Listing each possible variable value independently, instead of grouping them together by common condition will increase the number of test outputs by

a factor equal to the number of variables. Condition redundancy abstraction can be used to avoid this extra, unnecessary abstract test output.

Algorithms have been developed for performing value and condition redundancy abstraction during abstract test output generation. In the case of condition redundancy, we have an approximate abstraction algorithm that will group together variables if they depend on the same set of program conditional statement conditions. We have also included a facility for user direction of logical abstraction to allow experimentation with different reductions involving different variable groupings.

5 EXPERT SYSTEMS

5.1 Application area

For the purposes of abstract testing, expert systems are different in several ways from the kinds of control programs studied in the QDA project, and from the ASPECTS data processing programs. In expert systems, the problem domain abstractions that are used to build the system are directly represented in the system in the form of predicates, attributes and rules. This implies that there may be no need for an additional facility for introducing abstractions into the testing process, like the comments language and other problem abstraction facilities in the QDA and ASPECTS work.

Previous research on expert system verification has included work on determining the consistency and completeness of the knowledge base, or rule set. Various kinds of static analysis methods have been used to determine anomalies such as loops in rule chains or unreachable states. Our view of this work is that it is the kind of preliminary analysis that is necessary to carry out prior to testing, to confirm that the system is testable.

Other research has attempted to incorporate types in expert systems. Our view of type checking for rule based systems is that it is analogous to hypothesis checking in QDA. When abstractions are introduced, their meaning is dependent on assumptions about the kinds of data they involve. These assumptions can be documented in the form of types of variables. We have investigated other kinds of hypotheses and have demonstrated that some of the type checking inadequacies in previous approaches can be solved using a QDA-like flavor feature [12]. Flavors can be used, for example, to document the assumption that some lower level rule in a hierarchical system returns a consequent whose meaning contains information expected to have been determined by antecedent conditions in its rules.

In our analysis of expert systems we will distinguish between *data driven* and *goal driven* systems.

5.2 Data driven systems

In systems of this kind, there are a number of observation attributes, each of which can take on a finite set of values. Similarly, there are a number of derived attributes, which take on values in ways that are defined by rules. Modules consist of sets of rules that can be grouped together on the basis of the consequent part of the rule. For example, a bird classification system might have several rules of the form

IF feet(webbed) AND density(floats) THEN order(waterfowl)

Each rule in the module will give a different criterion for a bird to belong to some order, based on observations about its feet, bill and other basic characteristics. Additional rule modules might contain the "order" abstraction in their definition and be of the form

IF order(waterfowl) AND bill(narrow) THEN family(surface-feeder)

where bill is a basic observation.

Naive exhaustive testing of a data driven system would involve constructing all possible combinations of observation attribute values. This will often result in too many tests, most of which may be meaningless combinations that do not result in any classification/output. This implies the need for some kind of abstract testing.

Abstract testing of an expert system can be implemented using the system's modular structure. In the above example, this would entail testing the order module and the family module separately. In the case of the order module, we would be choosing different possible values for the basic observations that can appear on the left of the rules in an order module rule. A similar approach would be used for the family module, adjusted to take into account the fact that the consequent predicate from some other module can appear as an antecedent in its rules.

The use of modules in reducing test set sizes depends on assumptions about the relevancy of attributes and attribute values. It is implied in module testing that test sets are limited to the construction of combinations of attribute values for the attributes that appear on the left hand sides of the rules in the module. We call this an *attribute relevancy assumption*. Note that this does not mean testing each rule over input constructed only from its antecedent attributes since all rules in a module may not have the same attribute sets. Additional restrictions can be imposed using *attribute value relevancy assumptions*, in which it is assumed that the relevant possible attribute values for a module are restricted to some subset of the different possible attributes, possibly to those contained in occurrences of the attribute in the module's rules.

Attribute and attribute value relevancy are examples of *data dependency assumptions*. When an abstraction is created, it is formed with the idea that the abstraction depends on certain data. In addition, an abstraction may be constructed under the assumption that the data will have certain properties. We call these additional assumptions *interface dependency assumptions*. In the QDA project interface dependency assumptions were of major importance and, as in the case of the ASPECTS work, data dependency was implicitly defined by the data items in the code that was being abstracted. In our work on expert systems, data dependency appears to be the more important consideration, and needs to be made more explicit.

Even when modularization with attribute relevancy and attribute value relevancy assumptions are used, there still may be too many possible combinations of attribute values, and hence too many tests. One solution that can be used is to restrict what occurs in a module on the basis of its output. A module could be defined to be a set of rules with the same predicate and the same predicate output value in the consequent parts of the rules. We could call these *strong* as opposed to *weak modules*.

Each of the above kinds of data dependency assumptions depend on an expert system's modular structure. An alternative to module data dependency, called *output data dependency* can be used. It is based on the idea that for each possible predicate output value, for a module or the system as whole, that an expert will be able to say which basic observation attributes, values or value combinations are relevant to the production of that

output. It is assumed that the expert will be able to do this independently of the use of attributes and attribute values in the system or system modules.

It might appear that knowing all relevant input attribute value combinations for output values is equivalent to being able to tell by inspection if a system's rules are correct, but this is not the case. For example, it may be that some output x is such that two attributes A1 and A2 always have values a1 and a2 for x. But both attributes might not be used in the expert system because they are not needed to distinguish between x and the other output possibilities included in this particular expert system, although they could be necessary in some broader system designed to make more precise distinctions. We may wish to verify the system to make sure that the rules, as a matter of economy, do not mistakenly leave out information that is necessary. Errors like this can occur in a system that is evolving, in which at some point the attributes necessary to distinguish between different outputs become insufficient.

5.3 Goal driven systems

In a goal driven application, the expert system is given a predicate and value expression, and it is desired to determine if the state modelled by that expression can be derived from an initial state using the rules of the expert system. We will consider the kind of goal driven system in which there are a number of objects, each of which has a set of attributes that can take on a set of values. Each possible combination of attribute values is thought of as a state. Such a system could be used, for example, to build semi-autonomous vehicles. The human controller would send a command to the object consisting of a final state description. The expert system would then attempt to find a sequence of actions that would produce that state. If the system were unsuccessful, the controller would have to devise and carry out the necessary sequence of elementary commands.

The number of possible states in such a system can be very large, so that exhaustive testing of the system as a whole is not possible. Output data dependency may not be useful since the user will not know for each desired output state what the different possible input states are that can produce that output. Even if this were not the case, the number of different possible output and input states is too large to consider this possibility. This indicates that some form of module data dependency may be the only hope.

We will illustrate our discussion of this kind of system using the "monkey and bananas" problem . In this example there is a monkey in a room with a banana, a blanket, several toys, and a ladder. The goal is for the monkey to hold the banana. Each object has attributes such as "location", "weight" and "on". The monkey has the additional attribute "holds".

One possibility for modularization is to group rules together into clusters that accomplish the same subgoal, starting from the same class of states. For example, a subgoal might be for the ladder to be moved to some location, given the condition that the monkey is on the floor. The expert system would contain rules that would model the situations and ways in which the ladder could be moved. This could result in modules for which the testing problem was still intractable, if we try to test the rules in the module over all possible input states for all objects, unless we identify relevant assumptions associated with the formation of the cluster abstractions that limit the possibilities.

The first assumption is similar to the kinds of module relevancy assumptions defined above in the discussion of data driven systems. It will be called object relevancy. We will assume, that for each cluster, that it is possible to determine if an object is relevant to the kinds of actions incorporated in the rules contained in that cluster. We may automatically

exclude, for example, all objects that are not contained in any of the rules in the cluster. If the cluster is contained in a super-cluster of clusters, we may wish to still allow the consideration of objects in related clusters that are part of the same super-cluster.

Other kinds of relevancy assumptions can be defined for goal oriented systems that are the same as those for data driven systems. In the case of attribute relevancy, we assume the ability to recognize whether or not object attributes are relevant to the cluster abstraction. In the testing of the rules in the cluster, for example, we may wish to restrict our consideration of input state characteristics to only those attribute value combinations for attributes that appear in at least one rule in the cluster. In a similar way, attribute value relevancy can also be defined for goal driven systems.

5.4 Classes of expert system faults and criticality of system errors

In our work on expert systems, we found it interesting to distinguish between two different kinds of system faults, and to relate these to the seriousness of different kinds of system errors. A data driven system may or may not recognize some input set of observations. One kind of incorrect behavior is to report recognition of the wrong kind. Another is to report recognition of some kind when the input data case is not meaningful, and the system should not succeed in recognizing anything. It may be programmed to report having recognized a dummy situation "inconclusive" in this case. It can also falsely report an inconclusive result when it should recognize the input observation set.

Similar kinds of incorrect behavior can occur in goal driven systems. A system might not satisfy some goal when it should, and vice versa. We have to distinguish between when the expert system "fails" in the sense that it (correctly) does not recognize some input set or does not achieve some goal, and "fails' in the sense that it acts incorrectly due to design or programming faults. We will refer to a system as *succeeding* or *failing* in the following sense. If a data driven system recognizes some input then it succeeds, otherwise it fails. When a goal driven system satisfies its goal for some given state, then it succeeds, otherwise it fails. Note that this has nothing to do with whether or not the system has behaved incorrectly. In this case we say that it produces an error, or that it has *erred.*

Three broad classes of expert system error producing program faults can be identified: *generality*, *specificity*, and *mixed* faults. In the first case, something is left out of a rule. The result will be that the system is less discriminating than it should be. This can produce incorrect output, in the case where it allows an extra path to success through the rule system. It can allow the system to report success when it should not. It does not, however, allow the system to report failure when in fact it would otherwise succeed. In testing a rule containing a generality fault, if we do not consider the missing attribute we may or may not choose tests where it has the correct value for the rule to succeed, so that we are not guaranteed to test it using data that will cause it to succeed when it should fail.

The second kind of fault, specificity, occurs when extra terms are introduced into rules, making them too discriminating. In this case, a rule will not be successful in situations where it should be. Such faults can result in systems failing when they should succeed. They do not result in a system succeeding when it should fail, and except for reporting failure falsely, do no return with incorrect answers.

The third class of faults can occur in which specification and generality faults are combined, and in which a system can both succeed when it should fail and fail when it should succeed.

Different kinds of system errors may be more critical than others. For example, for some systems it may be critical if the system fails when it should succeed, and for others, critical if it succeeds when it should fail. The first case is illustrated below for a hazard detection system.

We consider the case of a data driven system that has been constructed to do hazard detection. In the simplest case there is only one kind of output, that a hazard has been detected. In this case, it is critical error if the system fails to recognize a critical situation (i.e. system failure rather than success), but not critical if it falsely succeeds (i.e. reports success when it should fail). For such a system, generality faults do not produce a critical error. Specificity or mixed faults can produce critical incorrect behavior. Output relevancy testing, or module attribute or attribute value testing are sufficient to detect such faults.

Generality faults for the data driven system will be discovered if we use module attributes relevancy testing, and the missing attribute in a rule is present in the module. To detect generality faults in which an attribute is completely missing from a module we have to use some other mechanism to detect relevant attributes. One possibility is to use output relevancy instead of module relevancy. In this case we assume that when programmers/experts are constructing input cases relevant to some module they will know to include the attribute that is missing from the faulty module. Another is widen the domain of attributes that are considered in abstraction formation using "super-abstractions" of some kind, from which this abstraction inherits all of its attributes.

6 SUMMARY AND FUTURE WORK

6.1 Abstractions and abstract programs

One common feature in each of the three problem areas discussed above is the formation of abstractions whose validity depends on assumptions about the part of the code being abstracted. Abstractions may include the identification of the input data upon which the abstracted code may depend, together with assumptions about the data, such as its types and/or flavors. In each case the resulting abstractions form executable operations, resulting in an abstract program model that can be completely tested. There are two factors in the execution of a complete set of tests. The first is to test over all allowable input data. In some cases the abstract program is non-deterministic with respect to branching conditions, so it is necessary to add that all paths should be tested over all data.

In the case of the QDA examples, input data is represented very abstractly with a single input set in which each input variable may have one or more abstract input properties. The abstraction process produces abstract operations (events) and complete testing corresponds to testing all paths through the abstract program. All paths can be executed because the abstraction replaces the original program with a representation in which there only a finite number of possible states, so that all loops "converge".

In the assembly programs analyzed using QDA, abstract versions of programs were created that identified the abstract meanings of short sections of assembly code. The formation of these abstractions depended on interface dependency assumptions about what had previously been achieved, documented in the form of hypotheses to be checked as part of the abstraction formation process. Once the abstractions were created, using events, we could execute the code at the abstract level. Such abstracting events may have to accumulate previously computed information so that when a terminal point in the code is reached, at which an output specification is to be checked, all of the abstract conditions necessary to satisfy the specification are present. This results in a heavy emphasis on the checking of

the conditions (hypotheses) that are documented as being necessary to the formation of the property accumulating operation abstractions (events).

In the ASPECTS work, complete testing also corresponds to the idea of testing all paths through an abstract program over a set of abstract input data, which in this case consists of symbolic input data. The abstractions used to represent sections of code include assignments that have abstract functions on the right, and abstract predicates which replace concrete path branch conditions. In order to be able to execute all paths through the abstract program, techniques like "loop abstraction" are used to replace iterative sections of code with an in-line abstract representation. The resulting abstract program paths are symbolically executed to produce symbolic test output consisting of path conditions and symbolic variable values.

In the data processing programs analyzed in the ASPECTS project, we were dealing with high level code. This makes it more desirable to retain some of the code and mix it in with program and data abstractions. The use of symbolic evaluation makes this possible. In this case it is still important for users to ask, whenever they are checking the output of some aspect of the code, what the implied interface assumptions were in the formation of the abstraction corresponding to that section. But the approach is more operational, and the output less predicable, so that the checking of abstract output, as opposed to abstraction preconditions, is more important. Since we generate potentially large numbers of abstract outputs we also perform logical abstraction to simplify the output.

In expert systems, the abstractions correspond to modules, and the abstraction assumptions correspond to data dependency assumptions about input objects, attributes and attribute values. These assumptions make it possible to exhaustively test programs by testing all modules over all input data. In this case the abstract programs are deterministic with respect to the "branching" conditions, and the enforced testing of all "paths", which will occur automatically when testing over all "input", does not have to be explicitly forced in the testing process.

When programmers test a system, they know that they cannot test all the details. Yet testing is considered essential and it is necessary to understand why it is important and what it can do. One thing that is important in system testing is to make sure "all the pieces" fit together correctly. This can be done through analysis by requiring the documentation of and verification of interface dependency assumptions. Testing at the abstract level also checks to see if the pieces fit together. In the case of the QDA and of the ASPECTS data processing programs we can see if all of the abstracted code segments fit together by completing abstract tests and verifying the abstract output. In the case of the data driven, hierarchical expert systems, testing of the top level module determines if the next lower level abstractions and the rules occurring at that top level module all fit together. This process can then be carried out down to the lowest levels. In the case of the goal driven system, each cluster is a kind of integrating point for other clusters that interact with its rules. The system can be considered to be hierarchical, with each cluster taking a turn as the top of the hierarchy, and testing it tests if the pieces fit together at some abstract point of view.

6.2 Future work

Future work will proceed in all three areas of interest, including the development of more sophisticated abstraction/reduction methods for data processing programs, and a more detailed analysis of different kinds of expert systems and expert system faults. We are also developing interface dependency analysis methods for expert systems that are like those used in the QDA project. This will enable us to attack the problem of finding generality

faults that cannot be found by data dependency based testing. In addition, work is proceeding on the development of a QDA derivative for Ada.

Acknowledgements

The research and development involving the QDA system was supported by the Office of Naval Research and the Naval Weapons Center. The expert systems research was supported by Bendix Field Engineering. The research on data processing programs is carried out in association with the Software Maintenance Assistant project at the Software Engineering Laboratory at the University of Hawaii.

References

[1] J.W. Laski , and B. Korel, A Data Flow Oriented Program Testing Strategy, *IEEE Transactions on Software Engineering*, SE-9, 3, May, 1983.
[2] S. Rapps and E.J. Weyuker, Selecting Software Test Data Using Data Flow Information, *IEEE Transactions on Software Engineering*, 11-4, April 1985.
[3] M.A. Hennell, M. R. Woodward, and D. Hedley, Towards More Advanced Testing Techniques, *Workshop on Reliable Software*, Bonn Germany, 1978.
[4] W.E. Howden, Methodology for the Generation of Program Test Data, *IEEE Transactions on Computers*, 24-5, May,1975.
[5] W.E. Howden, *Algebraic Program Testing*, Acta Informatica, 1978.
[6] W.E. Howden, Weak Mutation Testing and the Completeness of Program Test Sets, *IEEE Transactions on Software Engineering*, 8-4, July, 1982.
[7] R. DeMillo, R.J. Lipton, and F.G. Sayward, Hints on Test Data Selection: Help for the Practicing Programmer, *Computer*, 11-4, April 1978.
[8] W.E. Howden, *Functional Program Testing and Analysis*, McGraw-Hill, 1987.
[9] W.E. Howden and Cheron Vail, An Informal Verification of a Critical Program, Software Engineering Laboratory, UCSD, La Jolla, CA, 1992.
[10] C. Ghezzi and M. Jazayeri, Syntax Directed Symbolic Execution, *Proceedings of COMPSAC 80*, Chicago, 1980.
[11] W.E. Howden and Suehee Pak, Abstractions and Problem Dependency in Reverse Engineering, Software Engineering Laboratory, UCSD, La Jolla, CA, 1992.
[12] W.E. Howden and Yudong Huang, Total Testing Methods for Expert Systems, Software Engineering Laboratory, UCSD, La Jolla, CA, 1992.

A MODEL FOR IMPROVING THE TESTING OF REUSABLE SOFTWARE COMPONENTS[1]

Jeffrey M. Voas
Reliable Software Technologies Corporation
1001 N. Highland Street, Suite PH
Arlington, VA 22201 USA
Phone: (703) 276–1219

Keith W. Miller
Department of Computer Science
College of William & Mary
Williamsburg, VA 23185

Abstract

This paper briefly describes the problem changing input distributions cause to software reliability estimates for reused software components. This paper then presents a state-based technique for performing additional testing if it is determined that a change in the input distribution requires reverification.

Index Terms

Software testing, data state, software testability, software reuse, software reliability estimation.

Jeffrey Voas is currently Vice President of Advanced Research at Reliable Software Technologies Corporation in Arlington, VA. In 1992, Voas completed a National Research Council resident research associateship at the National Aeronautics and Space Administration's Langley Research Center. His research interests include techniques for predicting software testability, new software metrics based on semantic rather than syntactic information, debugging, design techniques for improving software testability, software testing, and software reliability estimation. Voas received a PhD in computer science from the College of William & Mary in 1990. He is a member of ACM and IEEE.

Keith Miller is an associate professor of computer science at the College of William & Mary. His research interests include software engineering, software reliability, and computer ethics. Miller received a PhD in computer science from the University of Iowa in 1983. He is a member of ACM and IEEE.

[1]Supported by a National Research Council NASA-Langley Associateship and NASA-Langley Grant NAG-1-884.

A Model for Improving the Testing of Reusable Software Components

1 Introduction

Reusable software components have been extolled as a way of increasing software reliability; reusable hardware components and the ability to quickly replace faulty hardware components have greatly increased the reliability and reduced the down-time of hardware systems, and it is hoped that similar improvements can be achieved in software systems. However it is not necessarily possible to quickly replace a faulty software component. Object-oriented languages and libraries of mathematical and scientific procedures are outgrowths of the desire for software reuse. Intuitively, a software component that is trustworthy in one application can enhance the reliability of a different application. Unfortunately, this is not always the case.

Reused software components present significant difficulties in achieving and assessing reliability, and these problems are particularly significant for critical software systems. There are many different problems associated with the reliability of software components; perhaps the most fundamental problems are ensuring that a given component is reused in a context where its specified behavior is precisely what is required. In this paper we describe a different problem: assuming that the component's specification *is* the proper one for this application, and assuming that the component has undergone extensive testing, how does the reliability prediction change when the component is reused in a new environment?

The testing that originally establishes a reliability prediction for the component must be based on some *input distribution*. When the component is reused in a new hardware environment, the input distribution will (in general) change. A changed input distribution may require that the reliability estimate be lowered and more testing be done to obtain the same confidence in the component using the new distribution. This additional testing can be considerable, decreasing cost benefits of software reuse.

In this paper, we present the problem that changing input distributions cause to software reliability for reused software components. We will also present a state-based technique for performing additional testing if it is determined that a change in the input distribution requires it.

2 Preliminaries

We assume that the software component C that is going to be reused has had a reliability R_1 under input distribution D_1. The goal is to show that C has a reliability of R_1 under a new input distribution, D_2. We assume that exhaustive testing is infeasible for both input distributions, and further assume that the reliability of the component will be established via random black box testing, where the inputs are drawn according to the appropriate input distribution. Given that the required testing has been accomplished for C under input distribution D_1, we wish to ascertain the reliability of that same component (C).

At one extreme we could perform all the testing necessary to establish reliability R_1 under D_2 ignoring previous testing done with D_1; this extreme wastes testing resources. At the other extreme, we could simply assume that the reliability will be identical under the two distributions; this will not be true in general. For example, assume that only one possible input, x, causes C to fail. Assume further that x is very unlikely to be chosen using D_1, but that x is very likely to be chosen using D_2. In this case C will be more reliable under distribution D_1 than under D_2. Somewhere between the two extremes of ignoring previous testing and applying testing blindly, we think there are more productive ways of reducing the amount of new testing using D_2 while maintaining confidence that we still have a reasonably accurate prediction of the reliability.

One method of reducing the new testing required is to make a third input distribution, D_3, such that new testing using D_3 will accumulate additional tests that, when added to the tests already executed with D_1, will create a distribution compatible with D_2 [6]. If the additional tests reveal no failures, the resulting reliability assessment will be greater than the assessment established with the original tests for D_1. Using this method, no tests are ignored and the added tests are incorporated into a more precise reliability estimate. This method of supplementing the testing strictly on the basis of the distributions is particularly attractive when the number of tests necessary to augment the original testing, is relatively small. However, for some seemingly minor distributional changes, D_3 may require significant retesting, sometimes surpassing the number required to simply begin anew. In the rest of this paper we describe a more theoretical model-based approach to reusing testing information when the input distributions change.

3 A Simplified Reliability Prediction Model

Before we provide our theoretical model in §4, we begin by providing the reader with a reliability prediction model. The purpose of using a reliability prediction model with reusable components is to get an understanding of how reliable a particular component will be in a new environment. This knowledge will allow the developer to better assess whether a component can be deployed in a new environment without additional testing, and whether a component should not be considered for reuse at all. Note that we are not necessarily advocating the

use of this model; we provide it as background for a better understanding of the problem of predicting the reliability of a component in a new environment.

One simple way of assessing the reliability of a software component under a new distribution D_2 is to assume that D_2 is partitioned into histogram bins, each bin i having both a probability of having an element from it selected $p_{D_2 i}$ and each bin having a reliability estimate r_i [3, 1]. Each r_i represents the proportion of inputs in bin i on which the program will succeed. In this model, we further assume that each histogram bin for D_2 has a corresponding histogram bin in D_1, where both bins represent the same elements of the domain. Thus, the only difference between the corresponding bins in these two histograms is the values assigned to each $p_{D_2 i}$. If by chance $\forall i \ (p_{D_2 i} = p_{D_1 i})$, then $D_1 = D_2$. In this simple model with the previous assumptions, we let R_d, the reliability of the program under distribution d, be:

$$R_d = \sum_{i=1}^{n} p_{D_d i} \cdot r_i$$

where n is the number of histogram bins and d is some input distribution.

The assumption that distributions are based on identical histograms and that r_i's are readily available for the bins are not always realistic. It is more likely that we will not have r_i's and only "best guesses" concerning the input distributions. Frequently not only will the distributions change, but the input elements themselves will also change as the distributions change. This makes reliability reassessment more difficult. In this paper we focus on the simpler problem when the assumptions hold; the solutions we suggest can be generalized, but the complications require separate development.

4 Our Reverification (Testing) Model

From this point on, we will assume that the need for reverification of a reused component exists, i.e., the reliability reassessment technique that was applied before reverification began has failed to produce a reliability prediction that is acceptably high. Therefore, additional testing of the component will occur. In this section, we will provide a model for saving time on any additional tests that will be performed according to D_2.

In our reverification model for reusable software components, it is necessary to uniquely identify specific syntactic program constructs as well as the internal data states created during execution. To uniquely identify syntactic constructs, we define a *location* to be either an assignment statement, an input statement, an output statement, or the <condition> part of a **if** or **while** statement. Our definition for location is based on Korel's [5] definition for a single instruction. We admit that this is a toy language–no procedures, single input integer, and single output integer.

It is important to be able to determine whether a particular variable at some specific location has any potential impact on the output computation of the program; a variable is termed *live* at a particular location if this potential exists. Determination of whether a

variable is live is made statically from a control flow diagram that is augmented with def-use information. Admittedly, certain variables defined as live via static analysis of the data flow diagram might not be defined as live if our definition were based on the dynamic behavior of the program [4].

A program *data state* is a set of mappings between all live variables (declared and dynamically allocated) and their values at a point during execution. For the purposes of this paper, we will add three other values to each data state during execution: (1) the program input used for this execution, (2) the program output that results, and (3) the value of the program counter (**pc**).

A data state is only observed between two dynamically consecutive locations. The execution of a location is considered here to be atomic, hence data states can only be viewed between locations. As an example, the data state

$$\{(\text{input}, 3), (\text{a}, 1), (\text{b}, 50), (\text{c}, ?), (\text{pc}, 10)\}$$

tells that variables **a** has a value of 1, **b** has the value 50, the next instruction to be executed is at address 10, variable **c** does not yet have a value, and the program input that started this execution was a 3. Before program execution on an input begins, all variables are undefined.

We assume that when testing was performed according to D_1, each input on which the program halted produced some output. In our model, both the input value and output value are part of each data state; the output value however cannot be added to the data state until after test input was processed. This requires that after an execution occurs for some input x, each data state that was created has the resulting output added to it. If for our example data state the output was 5, the data state would be modified to look like:

$$\{(\text{input}, 3), (\text{a}, 1), (\text{b}, 50), (\text{c}, ?), (\text{pc}, 10), (\text{output}, 5)\}$$

When the software that executed under D_1 is reused, the input distribution under which it is newly executed may not be D_1. If the distribution is still D_1, then no additional testing is needed. However if the new distribution is D_2, where $D_1 \neq D_2$, then in order to predict that we still have reliability R, we will have to perform additional testing. This model presents a scheme by which we can lessen the costs (in terms of time) of this additional testing, by spending memory (storing data state information) when the distribution changes.

To demonstrate how this model can be applied to testing software under a new distribution, suppose that when we sampled according to D_1 we never selected a program input of 2 since it was very unlikely, but when we sampled according to D_2, we did sample a 2 because it had a much greater likelihood of being sampled. Further, suppose that with the input = 2 we have a data state that looks like:

$$\{(\text{input}, 2), (\text{a}, 1), (\text{b}, 50), (\text{c}, ?), (\text{pc}, 10)\}$$

We know where in the source code this state occurs (the program counter, pc = 10), and we can (at least theoretically) search for a data state generated during a previous test execution,

where that data state has that same program counter and same values for the variables. Note that the input and output values need not match. Assume that we discover such a matching data state with input = 3 and output = 5. From this point on, the computation of the program must be identical for the two inputs, so we know immediately that an output of 5 will result from the input 2. This allows us to "short-circuit" execution on this input, halting the input and recording the output. If the proper output for an input of 2 is 5 (according to a human or automated oracle), then the program has passed this test. This scheme of storing output values in data states trades the time and space required for storing and searching for data states with the time necessary for running the program being tested.

If this scheme of saved data states were fully implemented, it could be used to short-circuit processing in general. However, we believe it unlikely that this will be a practical technique in general because of the great number of data states that would have to be stored and searched. We think that testing certain types of programs may be an application where this technique can be successfully utilized.

5 When To Apply This Reverification Scheme

The technique of storing and searching data states (tagged with input/output pairs) is only practical at locations in programs where only a relatively few distinct data states occur. A recent study by Protzel demonstrates that some programs have locations where such a situation occurs, locations where a handful of states are the only ones to occur [7]. In order to more efficiently locate these places, we have examined the *domain/range ratio* metric. The domain/range ratio (or *DRR*) is the ratio of the number of possible inputs to the number of possible outputs. A program, a module, and a location can each be given a *DRR*.

Code that has a relatively high DRR is more likely to have a small number of possible data states than code with a lower DRR. Instances of high DRR code can be located by static analysis, and then monitored dynamically during initial testing. If a location that occurs early in a program's execution is found to have a relatively small number of data states, then the store and search technique described above may be practical.

In [8, 9], there is a conjecture that when identical program states are created by many program inputs, the testability of the software is decreased. This data state collapse makes it easier for programs to tolerate faults, because when the states collapse, evidence of the fault (an incorrect data state) can be removed before the program outputs any values that reflect the error. If this is true, then a testing strategy based on this data state model should be most beneficial for testing reusable software that suffers from low testability, where the lower testability is due to a greater ability to cancel data state errors. However as Hamlet has pointed out [2], when there is a large amount of data state collapse, then there are many resulting common computation segments, and it may be enough to test these segments with only a few tests. If this is true, then these locations with high data state collapse may only appear to suffer from a greater ability to hide faults, but in actuality they are quite easy

to test if we can find the tests to exercise the computation segments. (The reader should note that there are other factors that can contribute to low software testability, and we do not yet know how well this testing scheme might work to counteract those factors.) These issues require further research before we will understand the complete impact of data state collapse on testing and reliability.

6 Concluding Remarks

Decreasing testing costs that are incurred when a piece of software is placed in a new environment is one way to encourage software reuse. Our scheme may reduce the costs of testing in terms of time for particular types of programs; this reduced cost is at the expense of memory. For other programs, little testing benefit would be derived from this scheme. The likelihood of many different program inputs mapping to the same program output is the determining factor in the value of this strategy, and the DRR can help us locate code fragments where the strategy may be effective.

We also feel that this strategy (if forcefully implemented during design) can serve as a "design-for-reverification" of reusable software components. That is, it might be possible to force a program to collapse to a few internal states at some early point in execution if it is known that the number of potential distinct outputs of the program is very small. (A boolean function would be an example of this). This essentially suggests that we might be able to reduce the overall amount of software written, by inserting a table lookup on the output once we get to a location l with very few internal data states. This would be implemented at such a location by some mechanism similar to:

```
table_lookup_location_l(current_internal_state, eventual_output)
    write(eventual_output)
```

And of course we will need ways to certify that look-up is being used and that it is being used correctly. The proposed benefit of programming via table lookup will suffer if programmers modify a lookup table when they should leave it alone.

The obvious criticism of this proposed scheme is getting the outputs (`output_1, output_2, ...`) for the table lookup. In practice it will be necessary to program the software to find the outputs, verify that they are correct, and then at a later time replace the (now) unnecessary code with a lookup table. Only if the location where this phenomenon occurs is very early on in the execution sequence will any practical benefit be derived. The design method described here may not be immediately applicable for most applications; but for some applications, the payoff could be immense, changing computationally expensive programs into O(1) table lookups. Being alert to the possibility of such a conversion may reveal that the possibility exists in more applications than might be at first expected. Furthermore, even if very few applications are found which make this conversion possible, the theoretical possibility offers insights into software design, software reusability, and software testing.

Acknowledgements

The authors express thanks to D. Hamlet and to the other referees who have made insightful and complimentary comments on previous drafts of this manuscript.

References

[1] J. W. Duran and S. C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, SE-10:438–444, July 1984.

[2] D. Hamlet. Are We Testing for True Reliability?. *IEEE Software*, pages 21–27, July 1992.

[3] Richard Hamlet and Ross Taylor. Partition Testing Does Not Inspire Confidence. *Second Workshop on Software Testing, Validation, and Analysis*, pages 206–215, July 1988.

[4] Bodgan Korel. Dynamic Program Slicing. *Information Processing Letters*, October 1988.

[5] Bodgan Korel. PELAS-Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering*, SE-14(9), September 1988.

[6] K. Miller, L. Morell, R. Noonan, S. Park, D. Nicol, B. Murrill, and J. Voas. Estimating the Probability of Failure When Testing Reveals No Failures. *IEEE Trans. on Software Engineering*, 18(1):33–44, January 1992.

[7] Peter W. Protzel. Automatically Generated Acceptance Test: A Software Reliability Experiment. In *Proc. of the Second Workshop on Software Testing, Validation, and Analysis*, pages 196–203, July 1988.

[8] J. Voas, K. Miller, and R. Noonan. Designing Programs that Do Not Hide Data State Errors During Random Black-Box Testing. In *Proc. of the 5th Int'l Conf. on Putting Into Practice Methods and Tools for Information System Design*, Nantes, France, September 1992. Université de Nantes/I.U.T.

[9] J. Voas. Factors That Affect Program Testabilities. In *Proc. of the 9th Pacific Northwest Software Quality Conf.*, pages 235–247, Portland, OR, October 1991. Pacific Northwest Software Quality Conference, Inc., Beaverton, OR.

Index

1992 PROCEEDINGS ORDER FORM

PACIFIC NORTHWEST QUALITY SOFTWARE CONFERENCE

To order a copy of the 1992 proceedings, please send a check in the amount of $35.00 to:

PNSQC
PO Box 970
Beaverton, OR 97075

Name _____

Affiliate _____

Mailing Address _____

City _____

State _____

Zip _____

Daytime Phone _____