TWELFTH ANNUAL

PACIFIC NORTHWEST

SOFTWARE QUALITY CONFERENCE

October 17 - 19, 1994

Oregon Convention Center
Portland, Oregon

TABLE OF CONTENTS

QUALITY ASPECTS TRACK - October 18

TESTING TRACK - October 18

MANAGEMENT TRACK - October 19

QUALITY ASPECTS TRACK - October 19

TESTING TRACK - October 19

Preface

Hilly Alexander

Welcome to the **Twelfth Annual Pacific Northwest Software Quality Conference**. Over the past year we have watched high-tech companies change their strategic focus to put more emphasis on software. Customers expect ever more usable and reliable software products. As software professionals we must continuously search for ways to deliver higher quality software in less time.

This Conference again offers an excellent opportunity to learn from successful practitioners and well known experts. During the two day technical program speakers from North America and Europe explain how they are meeting the challenges of the mid-nineties. We invite you to meet the speakers and share information with your peers in the "Birds of a Feather" sessions during lunch.

The keynote speaker on the first day of the conference is **John Zachman**, an international consultant on Information Systems Architecture. He stresses the importance of understanding the business enterprise before selecting development methodologies and tools.

Dr. David Lorge Parnas, a Professor at McMaster University, will present the keynote address on the second day. He draws on his academic and industry experience to find a "middle road" between theory and practice. He emphasizes theory that can be applied to improve the quality of our products.

We are pleased to publish these Proceedings, which contain the papers presented during the technical program. The 22 papers were selected from 65 abstracts. In addition to the refereed papers, we have invited ten speakers to share their expertise and insights.

I would like to thank Sue Bartlett and Peter Martin, the Program Committee Co-Chairs, for putting in the hard work and many hours it takes to organize the program. Many thanks also go to the Abstract Selection Committee for their efforts in reviewing and selecting these papers from over sixty abstracts and to the Program Committee members who reviewed the draft papers.

The members of all the committees have volunteered many hours and I thank them for their contributions to the success of this Conference. Feel free to talk to any of the committee members (they wear red committee ribbons attached to their name tags). We all welcome your input.

Finally, a special thank you to our conference manager: Terri Moore of Pacific Agenda. She has handled the organizational and administrative tasks very efficiently and kept all the committees on track and on schedule.

Mark your calendars for next year's events: Spring Workshops on May 22, Fall Workshops on September 27, Annual Conference on September 28 and 29. The keynote speakers at the Conference will be Mary Shaw, Carnegie Mellon University, and Dr. Charles Engle, Florida Institute of Technology.

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Hilly Alexander - President/Chair
ADP Dealer Services

John Burley - Secretary
Credence Systems Corporation

Ray Lischner - Treasurer/
Sofware Excellence Award
Rogue Wave

Sue Bartlett, Program Co-Chair
Tektronix, Inc.

Peter Martin, Program Co-Chair
Taligent, Inc.

Miguel Ulloa - Workshops
Mentor Graphics

James Mater - Exhibits
Revision Labs

G.W. Hicks - Publicity
Summit Design

Karen King - Birds of a Feather
Sequent Computer Systems

CONFERENCE PLANNING COMMITTEE

Judy Bamberger
Sequent Computer Systems

John Burley
Credence Systems Corporation

Martha Chamberlin
PenMetrics Inc.

Curtis Cook
Oregon State University

Margie Davis
ADP Dealer Services

Dave Dickmann
Hewlett-Packard

Erin Fassio
ADP

Lynne Foster
Motorola

Cynthia Gens

Warren Harrison
Portland State University

Dan Hoffman
University of Victoria

Connie Ishida
Mentor Graphics

Mark Johnson
Mentor Graphics

Bill Junk
University of Idaho

Wendy Kellberg
Attachmate Corporation

Karen King
Sequent Computer Systems

Randolph King
FLIR Systems

Ray Lischner
Rogue Wave

James Mater
Revision Labs

Joe Maybee
Tektronix Inc.

Howard Mercier
Intersolv

Shannon Nelson
Intel Corporation

Ian Savage
CFI Pro Services

Eric Schnellman
Boeing Commercial Aircraft

Barbara Siefken
Tektronix Inc.

Spass Stoiantschewski
Credence Systems Corporation

Bill Sundermeier
Quality Checked Software, Ltd.

James Teisher
Credence Systems Corporation

Don White
Cascade Solutions

Scott Whitmire
Advanced Systems Research

Barbara Zimmer
Hewlett-Packard

PROFESSIONAL SUPPORT

John Bistolas
Bistolas and Associates

Linda Cordelia
Graphic Design

Pacific Agenda, Inc.
Conference Management

PRESENTERS

Ed Amoroso
AT&T Bell Labs
67 Whippany Road, Room 15A-345
Whippany, NJ 07981

Frank J. Armour
American Management Systems Inc.
4050 Legato Road
Fairfax, VA 22033

Judy Bamberger
Sequent MS COL2-860
15450 SW Koll Parkway
Beaverton, OR 97006-6063

Michael W. Cohn
3700 Ocaso Court
Cameron Park, CA 95682

Dr. James S. Collofello
Arizona State University
Computer Science and Engineering Dept.
Tempe, AZ 85287-5406

Ward Cunningham
Cunningham & Cunningham
7830 SW 40th
Portland, OR 97219

Michael Deck
Cleanroom Software Engineering
Associates
6894 Flagstaff Road
Boulder, CO 80302

Paul Dittman
Tektronix Inc.
PO Box 500, MS 46-115
Beaverton, OR 97077

Michael F. Dunn
Industrial Software Technology
HC1, Box 93
Earlysville, VA 22936

Mark A. Fleming
Texas A&M University
H.R. Bright Bldg., Room 425
College Station, TX 77843-3112

Dick Hamlet
Portland State University
Computer Science
PO Box 751
Portland, OR 97207-0751

Roger Harrison
3M Health Information Systems
575 W. Murray Boulevard
Murray, UT 84157

Douglas Hoffman
Software Quality Methods
24646 Heather Heights Place
Saratoga, CA 95070

Hubert Hofmann
Universitat Zurich-Irchel
Winterthurerstrasse 190
Zurich, Switzerland CH-8057

Capers Jones
Software Productivity Research
1 New England Executive Park
Burlington, MA 01803-5005

Edgar Knapp
Purdue University
Dept. of Computer Science
West Lafayette, IN 47907-1398

Dr. Lech Krzanik
CCC Software Professionals Oy
Lentokentäntie 15
SF-90460 Oulunsalo, Finland

Rosemary Marchetti
Hewlett-Packard Co.
1501 Page Mill Road, MS 5MS
Palo Alto, CA 94304

Joe Maybee
Tektronix Inc.
PO Box 1000, MS 63-424
Wilsonville, OR 97070-1000

Bill McCarty
Azusa Pacific University
901 East Alosta Avenue
Azusa, CA 91702-7000

John McHugh
Portland State University
Computer Science
PO Box 751
Portland, OR 97207-0751

Michael Migdoll
Bank of America
c/o 1254 Panorama Drive
Lafayette, CA 94549

Kimberlyn C. Mousseau
Idaho National Engineering Lab
PO Box 1625
Idaho Falls, ID 83415-3730

J. Audery Neal
ACTC Technologies Inc.
350, 6715 - 8th Street N.E.
Calgary, Alberta, Canada T2E 7H7

David Parnas
Middle Road Software Inc.
551 Old Dundas Road
Ancaster, Ontario L9G 3J3

Andy Podgurski
Case Western Reserve University
Cleveland, OH 44106

Neil Potter
The Process Group
PO Box 870012
Dallas, TX 75287

Michael Sellers
New World Designs
2913 N. Aspen Way
Newberg, OR 97132-6903

Dr. A. Serra
ASIC Srl
Via Stefan Clemente, 6
Torino, Italy 10143

AA (Tony) Templeton
IBM Consulting Group
3600 Steeles Avenue East
Toronto, Ontario, Canada L3R 9Z7

Thomas Vagoun
University of Maryland/College of Business
Information Systems Department
College Park, MD 20742

John Zachman
Zachman International
2222 Foothill Boulevard, #3378
La Canada, CA 91011

Wayne M. Zage
Ball State University
Computer Science Department
Muncie, IN 47306

Andreas Zamperoni
Leiden University, Dept. of CS
Niels Bohrweg 2, NL-2333 CA
Leiden, The Netherlands

(award winner--name to come)
Credence Systems Corporation
9000 SW Nimbus Ave.
Beaverton, OR 97005

EXHIBITORS

Sabrina Rankin
Atria Software, Inc.
24 Prime Park Way
Natick, MA 01760

Steve Cornett
Bullseye Software
5124 24th Avenue NE, Suite 9
Seattle, WA 98105-3230

Mark Smith
Cadre Technologies, Inc.
10900 NE 8th Street, Suite 900
Bellevue, WA 98004

Dina Hunnum
CaseWare
108 Pacifica, 2nd Floor
Irvine, CA 92718

Roy Conant
Conant & Conant Bookseller
1001 SW 10th Avenue
Portland, OR 97205

Joan Brown
McCabe & Associates, Inc.
5501 Twin Knolls Road, Suite 111
Columbia, MD 21045

Alex Ellis
Mercury Interactive Corp.
333 Octavius Drive
Santa Clara, CA 95054

Mike Nelson
Northwest Software, Inc.
PO Box 91396
Portland, OR 97291-0396

GW Hicks
PNSQC
PO Box 10142
Portland, OR 97210

Bill Sundermeier
Quality Checked Software, Ltd.
PO Box 6656
Beaverton, OR 97007-0656

Lawrence Markosian
Reasoning Systems
3260 Hillview Avenue
Palo Alto, CA 94304

James Mater
Revision Labs
15262 NW Greenbrier Parkway
Beaverton, OR 97006

Teresa Harrison
SET Laboratories
PO Box 868
Mulino, OR 97042

Tricia McCormick
SQA, Inc.
10 State Street
Woburn, MA 01801

Nancy Bennett
Scopus Technology, Inc.
1900 Powell Street, Suite 900
Emeryville, CA 94608

John Zimmerman
Software Productivity Research
1750 Montgomery Street
San Francisco, CA 94111

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

Michael Shenker
Technical Solutions Inc.
12990 NW Sue Street
Portland, OR 97229

Neil Potter
The Process Group
P.O. Box 870012
Dallas, TX 75287

Information Systems Architecture: A Context for Understanding the Issues

John A. Zachman
Zachman International
2222 Foothill Blvd. Suite 337
La Canada, California 91011

Abstract

The basic concepts of the Framework for Information Systems Architecture serve as a context within which some observations can be made about things like: managing change, CASE, repositories, client/server, object-oriented, application development life cycles, etc. These technologies are useful, helpful and actually essential, but they are neither "silver bullets" nor substitutes for actual work. Models (as identified in the Framework) have to be built and managed and it doesn't happen by accident or magic.

It is imperative that we be honest with ourselves and with the user community and acknowledge that an enterprise is a complex entity and that there is no one simplistic solution to any complex problem. Information Systems as well as the Enterprise need to resist the temptation to be distracted by technological "silver bullets" and to commit to the real work of building the set of models that constitute a comprehensive description of the enterprise. The enterprise must have an inventory of these architectural models to serve as a baseline for managing change if it intends to survive the dynamics of the "Information Age."

Biographical Information

John A. Zachman is the author of the "Framework for Information Systems Architecture" which has received broad acceptance around the world as an integrative framework, or "periodic table" of information systems architectures. He is not only known for this work on information systems architecture, but is also known for his early contributions to Business Systems Planning, IBM's widely used, information planning methodology as well as to "intensive planning," the basis for IBM's executive, team planning techniques.

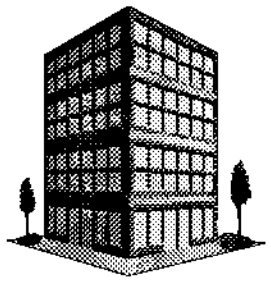
Mr. Zachman has been focusing on planning and information strategies and architecture since 1970 and has written a number of articles on those subjects. He has facilitated innumerable executive, team planning sessions. He travels nationally and internationally, teaching and consulting, and he is a popular conference speaker, known for his motivating messages on information issues. He has spoken to many thousands of information professionals on every continent.

Mr. Zachman retired from IBM after 26 years. He currently operates his own education and consulting business, Zachman International. He serves as a Special Advisor to the School of Library and Information Management at Emporia State University, Emporia, Kansas; is a member of the Information Resource Management Advisory Council of the Smithsonian Institution, Washington, D.C.; the Advisory Council to the School of Library and Information Management at Rosary College in River Forest, Illinois; the Board of Directors for the Repository / Architecture / Development Users Group; the Advisory Board of the Data Administration Management Association (DAMA) International and listed in "Who's Who in the West."

Prior to joining IBM, he served as a line officer in the United States Navy and is a retired Commander in the U.S. Naval Reserve. He chaired a panel on "Planning, Development and Maintenance Tools and Methods Integration" for the National Institute of Standards and Technology (formerly, the National Bureau of Standards). He holds a degree in Chemistry from Northwestern University, has taught at Tufts University and has served on the Board of Councilors for the School of Library and Information Management at the University of Southern California.

Information Systems Architecture

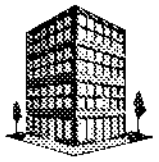
An Infusion of Honesty



John A. Zachman
Zachman International
2222 Foothill Blvd. Suite 337
La Canada, Ca. 91011
818-244-3763

Overview

A Framework for Information Systems Architecture



Different Perspectives

Architecture	Manufacturing	Info. Sys.
	OWNER	
Architect's Drawings	Wk. Bk. Dwn. Structure	Model of Business
	DESIGNER	
Architect's Plans	Engineering Design	Model of Info. Sys.
	BUILDER	
Contractor's Plans	Mfg. Eng. Design	Technlgy Model

Different Abstractions

WHAT	HOW	WHERE
Material	Function	Location
Bill of Materials	Functional Specs	Drawings
Data Models	Functional Models	Network Models

A Framework

	WHAT	HOW	WHERE
OWNER			
DESIGNER			
BUILDER			

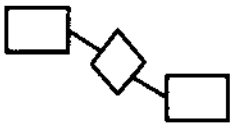
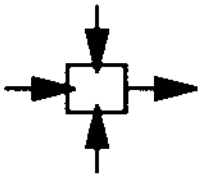
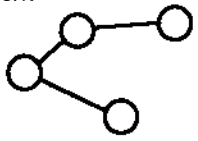
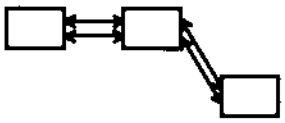
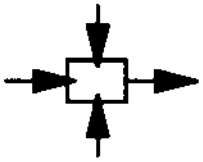
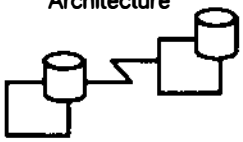
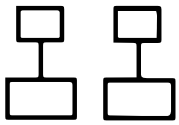
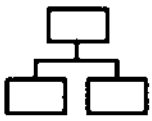
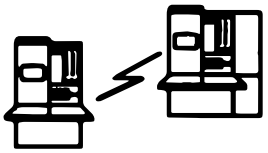
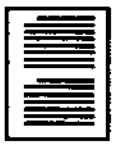
A Framework

	WHAT	HOW	WHERE
SCOPE			
OWNER			
DESIGNER			
BUILDER			
OUT OF CONTEXT			
PRODUCT			

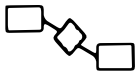
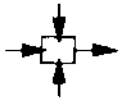
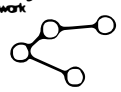
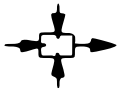
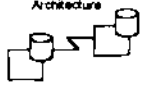
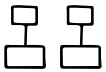
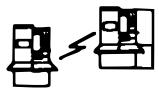
A Framework

	DATA	FUNCTION	NETWORK
SCOPE			
BUSINESS MODEL			
INFO SYS MODEL			
TECH MODEL			
DETAIL RPSNTNS			
SYSTEM			

INFORMATION SYSTEMS ARCHITECTURE - A FRAMEWORK

	DATA	FUNCTION	NETWORK
SCOPE *Planner*	List of Things Important to the business  ENTITY = Class of Business Thing	List of Processes the Business Performs  Function = Class of Business Process	List of Locations in Which the Business Operates  Node = Business Location
ENTERPRISE MODEL *Owner*	e.g., "Ent/Rel Diagram"  Ent = Business Entity Reln = Business Constraint	e.g., Business Process Model  Funct. = Business Process Arg. = Business Resources	e.g., Logistics Network  Node = Business Unit Link = Business Linkage
INFORMATION SYSTEM MODEL *Designer*	e.g., "Data Model"  Ent = Data Entity Reln = Data Relationship	e.g., "Data Flow Diagram"  Funct.= Application Function Arg. = User Views	e.g., Distributed System Architecture  Node = I/S Function (Processor, Storage, etc) Link = Line Characteristics
TECHNOLOGY MODEL *Builder*	e.g., Data Design  Ent = Segment/Row Reln = Pointer/Key	e.g., "Structure Chart"  Funct.= Computer Function Arg. = Screen/Device Formats	e.g., System Architecture  Node = Hardware/System Software Link = Line Specifications
COMPONENTS *Sub-Contractor*	e.g., Data Definition Description  Ent = Fields Reln = Addresses	e.g., "Program"  Funct.= Language Stmts Arg. = Control Blocks	e.g., Network Architecture  Node = Addresses Link = Protocols
FUNCTIONING SYSTEM	e.g., DATA	e.g., FUNCTION	e.g., NETWORK

INFORMATION SYSTEMS ARCHITECTURE - A FRAMEWORK

	DATA	FUNCTION	NETWORK
SCOPE *Planner*	List of Things Important to the business  ENTITY = Class of Business Thing	List of Processes the Business Performs  Function = Class of Business Process	List of Locations in Which the Business Operates  Node = Business Location
ENTERPRISE MODEL *Owner*	e.g., "Ent/Rel Diagram"  Ent = Business Entity Rein = Business Constraint	e.g., Business Process Model  Funct. = Business Process Arg. = Business Resources	e.g., Logistics Network  Node = Business Unit Link = Business Linkage
INFORMATION SYSTEM MODEL *Designer*	e.g., "Data Model"  Ent = Data Entity Rein = Data Relationship	e.g., "Data Flow Diagram"  Funct.= Application Function Arg. = User Views	e.g., Distributed System Architecture  Node = I/S Function (Processor, Storage, etc) Link = Line Characteristics
TECHNOLOGY MODEL *Builder*	e.g., Data Design  Ent = Segment/Row Rein = Pointer/Key	e.g., "Structure Chart"  Funct.= Computer Function Arg. = Screen/Device Formats	e.g., System Architecture  Node = Hardware/System Software Link = Line Specifications
COMPONENTS *Sub-Contractor*	e.g., Data Definition Description  Ent = Fields Rein = Addresses	e.g., "Program"  Funct.= Language Stmts Arg. = Control Blocks	e.g., Network Architecture  Node = Addresses Link = Protocols
FUNCTIONING SYSTEM	e.g., DATA	e.g., FUNCTION	e.g., NETWORK

Less than Enterprise Scope

	DATA	FUNCTION	NETWORK
SCOPE *Planner*			
ENTERPRISE MODEL *Owner*			
INFORMATION SYSTEM MODEL *Designer*			
TECHNOLOGY MODEL *Builder*			
COMPONENTS *Sub-Contractor*			
FUNCTIONING SYSTEM			

Re: *Any* Cell

Less than Excruciating Detail

	DATA	FUNCTION	NETWORK
SCOPE *Planner*			
ENTERPRISE MODEL *Owner*			
INFORMATION SYSTEM MODEL *Designer*			
TECHNOLOGY MODEL *Builder*			
COMPONENTS *Sub-Contractor*			
FUNCTIONING SYSTEM			

Re: *Any* Cell

End State Vision

	DATA	FUNCTION	NETWORK
SCOPE *Planner*			
ENTERPRISE MODEL *Owner*			
INFORMATION SYSTEM MODEL *Designer*			
TECHNOLOGY MODEL *Builder*			
COMPONENTS *Sub-Contractor*			
FUNCTIONING SYSTEM			

Enterprise - Wide Horizontal Vertical Integrated Architecture at Excruciating Level of Detail

The End State Vision

The end state vision:

Full set of models ...

enterprise - wide,
horizontally and vertically integrated,
at excruciating levels of detail,

on record (authorized),

being maintained (dynamically reflecting reality),

with design characteristics that allow them to accommodate:

orders of magnitude more change
with
orders of magnitude less impact,

this would constitute an enterprise positioned to respond to the dynamics of its environment.

Observation

Re: The "End State Vision"

If you had all the models ...
enterprise-wide,
horizontally integrated,
vertically integrated,
at an excruciating level of detail;

not only would you have a base-line for managing enterprise change

BUT

you would have an inventory of assets from which you could

assemble - to - order!

Information Systems Architecture

An Infusion of Honesty

Environment

I submit:

The credibility of I/S (in general) is declining dramatically.

A. The market has shifted.

For 50 years we have been unknowingly electing the short term option (make-to-order) and now we are paying for it.

("Pay me now - pay me later"
... it is now "later.")

Current demand is for "custom systems, mass produced in quantities of one for immediate delivery" ... (*assemble*-to-order.)

Environment (cont.)

B. The mystique is gone.

- Everybody has a PC on their desk.

"I can do this myself ... What's the problem?!"

(Note: Typical *Stage* 2 perspective.)

C. Complexity has become a critical factor.

- Price/performance escalation changes the problem space.

(Hundred story buildings vs log cabins)

- Rate of change continues as forcast:

"Future Shock"
Alvin Toffler 1970

Environment (cont.)

D. We all have this indefatigable penchant for "silver bullets."

The users like "silver bullets" - they'd rather throw money at the problem than hear about actual work.

I/S likes "silver bullets" - they'd rather have new technology than have to change behavior.

The vendors like "silver bullets" - they'd rather generate revenue this quarter than risk none at all.

It's a vicious cycle that creates enormous expectation gaps that fuel the credibility problem.

"Anybody who buys a 'silver bullet' bites the bullet!"
Warren Selkow

Recent"Silver Bullets"

1. Client/Server
2. Data Warehouse
3. Object Oriented
4. Out-Sourcing
5. Packages
6. Rapid Application Development
7. Repository
8. Prototyping
9. CASE
10. Artificial Intelligence
11. ISP (Info. Sys. Plng.)
12. "AD Cycle"

etc.

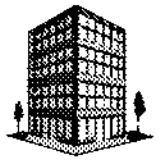
Old "Silver Bullets"

1. IOCS ... Operating Systems
2. COBOL ... RPG
3. Random Access
4. Structured Programming
5. Structured Design ... Analysis
6. DataBase
7. Relational DataBase
8. Virtual Memory
9. Time Sharing
10. BSP (Bus. Sys. Plng.)
11 Mini Computers ... Micro
12. Distributed Systems
13. SNA ... SAA ...

etc.

An Infusion of Honesty

Rapid Application Development



Rapid Application Development

	DATA	FUNCTION	NETWORK
SCOPE *Planner*			
ENTERPRISE MODEL *Owner*			
INFORMATION SYSTEM MODEL *Designer*			
TECHNOLOGY MODEL *Builder*			
COMPONENTS *Sub-Contractor*			
FUNCTIONING SYSTEM			

What can be done to make application development rapid?

Rapid Application Development

	DATA	FUNCTION	NETWORK
SCOPE *Planner*			
ENTERPRISE MODEL *Owner*			
INFORMATION SYSTEM MODEL *Designer*			
TECHNOLOGY MODEL *Builder*			
COMPONENTS *Sub-Contractor*			
FUNCTIONING SYSTEM			

1. Don't build any models.

(i.e. Go directly to write the code and forget about all this architecture stuff.)

(Faster yet ... generate the code!)

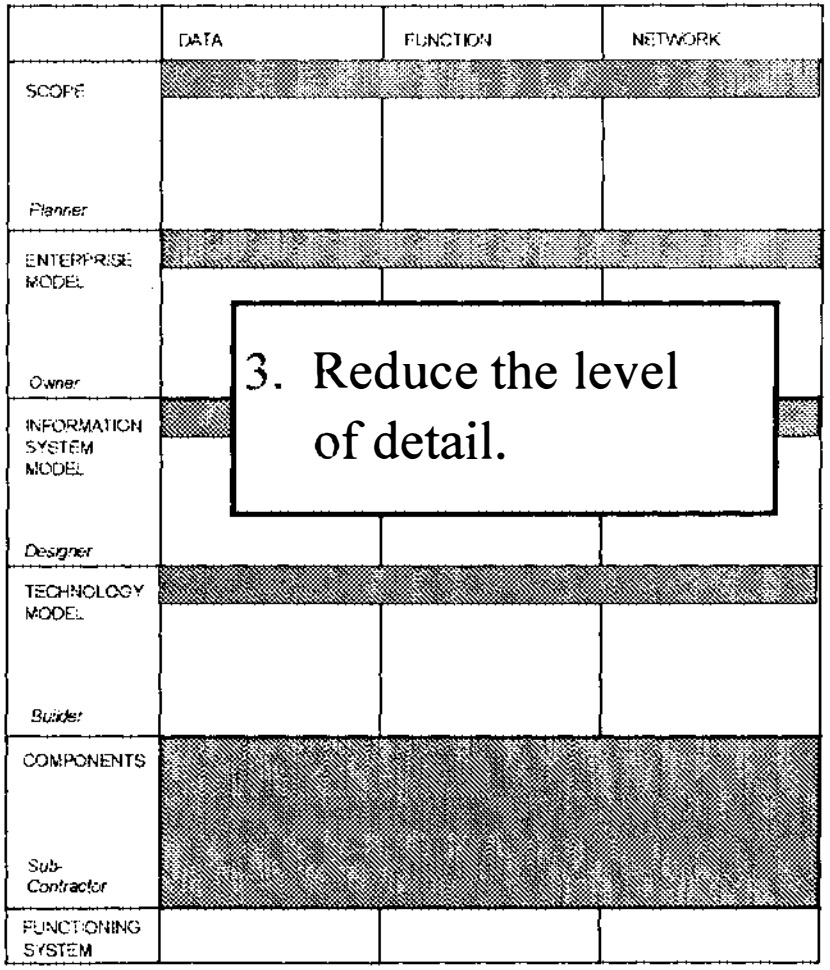
(Or, at least, only build a sub-set of the models ... the fewer the faster.)

Rapid Application Development

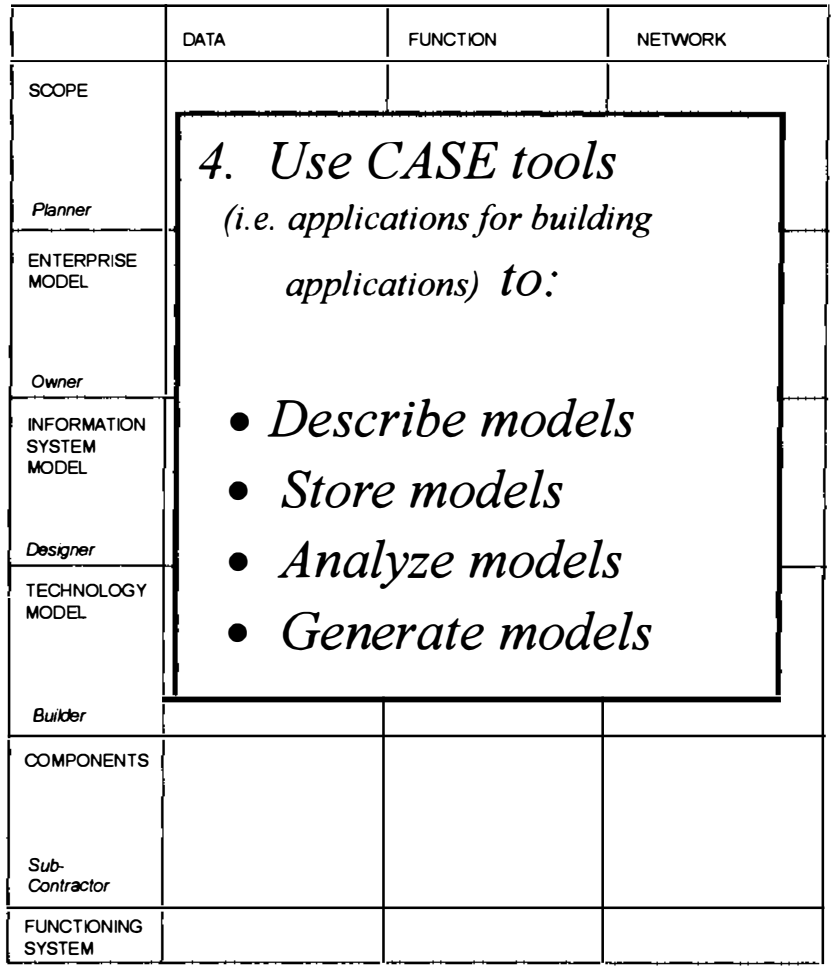
	DATA	FUNCTION	NETWORK
SCOPE *Planner*			
ENTERPRISE MODEL *Owner*			
INFORMATION SYSTEM MODEL *Designer*			
TECHNOLOGY MODEL *Builder*			
COMPONENTS *Sub-Contractor*			
FUNCTIONING SYSTEM			

2. Reduce the scope of the cells (models) you build.

Rapid Application Development



Rapid Application Development



Rapid Application Development

	DATA	FUNCTION	NETWORK
SCOPE *Planner*			
ENTERPRISE MODEL *Owner*			
INFORMATION SYSTEM MODEL *Designer*			
TECHNOLOGY MODEL *Builder*			
COMPONENTS *Sub-Contractor*			
FUNCTIONING SYSTEM			

5. "Concurrent Engineering"

(i.e. Produce multiple rows simultaneously, e.g. JAD)

Concurrent Engineering

Regarding

Concurrent Engineering,

Two Questions:

- ✓ Do you end up with one model or multiple models?
- ✓ Is it really possible to do design in a committee?

Rapid Application Development

	DATA	FUNCTION	NETWORK
SCOPE			
Planner			
ENTERPRISE MODEL			
Owner			
INFORMATION SYSTEM MODEL			
Designer			
TECHNOLOG MODEL			
Builder			
COMPONENT			
Sub-Contractor			
FUNCTIONING SYSTEM			

6. Buy someone else's models.
   (i.e. Packages ..." Provide from stock.")
7. Use previously built components.
   (i.e. " Assemble to order.")
8. Some combination of (any or all) of the above.

Rapid Application Development

	DATA	FUNCTION	NETWORK
SCOPE			
Planner			
ENTERPR MODEL			
Owner			
INFORMA SYSTEM MODEL			
Designer			
TECHNO. MODEL			
Builder			
COMPONENTS			
Sub-Contractor			
FUNCTIONING SYSTEM			

Use RAD, not because it magically eliminates application development time ...
but because it modifies the application development process based on a different set of values.

Summary Advice

Implement client/server, not because it magically eliminates mainframes or saves money ... but because it puts the right work on the right platforms and serves the Enterprise better.

Implement Data Warehouse not because it magically solves the legacy system problem ... but because it satisfies short term demand for data visibility and provides a migration path to an architected environment.

Implement Object Oriented, not because it magically delivers "reusability" or replaces analysis or design or COBOL programming ... but because it will force a better way of thinking about enterprise modeling and building systems.

Outsource, not because it magically saves money or replaces "incompetent" people ... but because it fulfills some specific strategic objective and/or takes advantage of specialized capabilities not otherwise available to the enterprise.

Summary Advice

Buy packages, not because they magically meet requirements ... but because they satisfy a short term demand. (Then, select them such that you minimize changes to the business rules!)

Use RAD, not because it magically eliminates application development time ... but because it modifies the application development process based on a different set of values.

Install Repositories, not because they magically integrate CASE tools ... but because they are a data base management system for models.

Use Prototyping, not because it magically eliminates Application Development ... but because you can validate various models and improve the relevance and quality of the product.

Summary Advice

Implement CASE, not because it magically produces code ... but because it improves application development through the use of the information technology itself. (Applications for building applications.)

Do ISP, not because it magically solves business problems ... but because it gives the enterprise an appreciation for architecture issues and establishes a context for segmenting and prioritizing implementations while minimizing the adverse effects of sub-optimization.

By all means, build reusable assets ... but understand there is nothing mystical about reusability. It only happens when there is a well-defined context (i.e. architecture) and when the "engineer" begins with the specific intention to design something (a part) that can be used for more than one thing (sub-assembly or product), which is a lot more complicated than designing something to be used for only one thing.

An Infusion of Honesty

I am not minimizing any of these concepts or technologies .. In fact, I endorse all of them! However, I am saying that we must understand their implications and limitation and set realistic expectations to:

1. preserve our credibility

and

2. protect the technologies.

We are dealing with an extremely complex Enterprise that is changing at an unprecedented rate.

It is lunacy to expect or to suggest that there are simplistic solutions, technology or otherwise.

I think we need a massive "Infusion of Honesty."

Honesty with ourselves.
Honesty with the Enterprise.

There is no such thing as a free lunch!!!

Our Heart's Desire

We all want:

Orders of magnitude reduction in cycle time for Application Development.

Orders of magnitude improvement in Information Systems (product) quality.

Orders of magnitude increases in system (product) flexibility, that is, orders of magnitude increases in responsiveness to change.

A Little Wisdom

I submit:

If you don't have an explicit, architecture-based strategy for achieving "your heart's desire," you are either:

A. Banking on more of the same, or
B. Hoping beyond hope for some kind of a technological "Silver Bullet."

Regarding "more of the same:"
"If you stake your hopes for a breakthrough on trying harder than ever, you may kill your chances for success."
Price Pritchett: You(Squared)
(Remember the fly.)

Regarding "Silver Bullets:"
Science and history have pretty well proven that:

- The universe is infinitely logical .. even in "Chaos."
- Nothing is mystical, not even technology.
- If you don't thoroughly understand the logic .. keep your hand on your wallet.

My Suggestion

My suggestion is,

given the inclination and a little time,

it is possible to map

any technology strategy, or

any potential "silver Bullet"

against the Framework

to help understand the implicit trade-offs,

to make knowledgable choices, and

to develop considered courses of action.

Friendly Advice

1. Don't get mesmerized by the "silver bullet" syndrome.

2. Be discerning and get creative.

3. Roll up your sleeves and go to work on the **"End State Vision."**

Software Aging

David Lorge Parnas
Communications Research Laboratory
Department of Electrical and Computer Engineering
McMaster University
Hamilton, Ontario, Canada L8S 4K1
email: parnas@triose.crl.mcmaster.ca

Abstract

Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable. A sign that the Software Engineering profession has matured will be that we lose our preoccupation with the first release and focus on the long term health of our products. Researchers and practitioners must change their perception of the problems of software development. Only then will Software Engineering deserve to be called Engineering.

Biographical Information

Dr. David Lorge Parnas is a Professor in the Department of Electrical and Computer Engineering at McMaster University in Hamilton, Ontario. He is a member of the Communications Research Laboratory and Principal Investigator for the Telecommunications Research Institute of Ontario. He has been Professor at the University of Victoria, British Columbia, the Technische Hochschule Darmstadt, and the University of North Carolina at Chapel Hill. He has held non-academic positions, including Philips Computer Industry, the United States Naval Research Laboratory in Washington, D.C. and the IBM Federal Systems Division. At NRL he instigated and led the Software Cost Reduction Project, which developed and applied software technology for avionics systems, for several years.

Dr. Parnas is the author of more than 160 papers and reports. His special interests include precise abstract documentation, real-time systems, safety-critical software, program semantics, language design, software structure, and synchronization. He has advised the Atomic Energy Control Board of Canada on the use of safety-critical software. Dr. Parnas seeks to find a "middle road" between theory and practice, emphasizing theory that can be applied to improve the quality in our products.

Professor Parnas received his Ph.D. in Electrical Engineering from Carnegie-Mellon University, and an honorary doctorate from the ETH in Zurich. Deeply concerned that computer technology be applied to the benefit of society, Dr. Parnas was the first winner of the "Norbert Wiener Award for Professional and Social Responsibility". He was recently elected a Fellow of the Royal Society of Canada.

2M: A Framework for Process and Technology Maturation

Lech Krzanik[1,2] and Jouni Similä[1,2]

[1]*CCC Software Professionals Oy*
Lentokentäntie 15, FIN-90460 Oulunsalo, Finland

[2]*University of Oulu*
Department of Information Processing Science
Linnanmaa, FIN-90570 Oulu, Finland
krzanik@rieska.oulu.fi

Abstract

Software process improvement methodologies such as the SEI's method, BOOTSTRAP or SPICE, pay limited attention to technology maturity - its development and exploitation. After 2 years of practical experience with BOOTSTRAP, this paper proposes a framework aiming at assuring balanced maturation of the software process and the software process technology. Both ultimately contribute to software process improvement. The paper describes the 2M approach and models behind it.

Keywords and Phrases: software process improvement, software process maturity, technology maturity, technology transfer, software metrics.

Biographical: Lech Krzanik and Jouni Similä divide their responsibilities between hands-on software industry (CCC Software Professionals Oy) and academia (University of Oulu, Department of Information Processing Science). Recently they have been involved with capability assessments and risk management of software processes. They were co-developers of BOOTSTRAP, the European software process assessment and improvement methodology, and currently take part in developing RISKMAN, a software project risk management tool interface and toolbox. They co-authored a book on BOOTSTRAP, to be published shortly, as well as a number of related papers.

1. Introduction

Just throwing tools at the problem doesn't work. The technology should be adequately mature, and adequate information about the technology should be available to those deciding about process improvement. This paper addresses software process improvement. An integrated view is taken that links software process maturity with technology maturity. In general terms such a viewpoint has already been exercised, e.g., [Leonard-Barton 88]. Now two elements are new. First - the emphasis on software process maturity has been balanced with technology maturity. Second - the technology maturity information has been associated with the technology itself rather than with a separate organization-oriented process-improvement process, as, for example, in the SEPG (Software Engineering Process Group) approach [Fowler and Rifkin 90].

We looked for software process and technology maturity approaches which were known, in a sense practically validated, and suitable for a wide range of software organizations (developers or maintainers). For assessing software process maturity, the BOOTSTRAP methodology was selected [Bicego et al. 93, Kuvaja et al. 94]. BOOTSTRAP declares compatibility with the SEI's method*, ISO 9001/9000-3, and the ISO's SPICE draft standard. For assessing technology maturity we combined several models, with notable constituents being: the IDA timeline model (for an overall technology level), the commitment model (for the organization level), the layered behavioral model, and the receptiveness model. The ultimate shape of 2M is due to the introduced metrics framework used as a common base for both process and technology measurement.

Section 2 outlines the 2M framework. Sections 3 and 4 describe the process and technology maturity frameworks respectively. Section 5 describes the process and technology requirements within these frameworks. In each case, first a number of representative methods are described, and then the 2M approach is presented. Section 6 discusses further details of the current state of 2M, and Section 7 enumerates future development directions.

2. 2M

"2M" ("Two Maturity Processes") is the name for a software process improvement program and an integrated framework designed at CCC Oulu and the University of Oulu, aimed at effective improvement of software processes. 2M has been developed to support:

---

* See the list at the end of the paper for explanation of abbreviations.

- strategic development - maturation - of an organization's software process,
- operational tuning of the software process within software development projects.

2M originated from practical observations of three types. One group of observations regarded cases of insertions of new technology into software processes which were attempted "too early". Either the process was unable yet to make effective use of the technology or the technology had not yet been validated or otherwise was not suitable for the target users. These synchronization problems had usually been discovered too late. There were also cases of "too late" insertions. Another group of observations regarded difficulties with analysis of the maturity synchronization status. This was caused by lack of appropriate information about the available technology or about the software process (required information exceeded the software process maturity models used). The third group of observations regarded inability to reuse developed technology status information for further decisions about technology maturation. The need for careful synchronization of the two maturation processes - that of technology maturation and that of software process improvement - has been evident from these observations. Unsynchronised interaction of the two processes led to harmful effects of budget overruns, schedule slippages and decreased morale of project teams. The synchronization of process maturity was not possible without access to the technology maturity information, and the synchronization of technology maturity was not possible without access to process maturity information.

The development of 2M included defining a framework consisting of the two maturity concepts - of an organization's software process and of software process technology - and providing a common user interface to both parts. 2M has been designed as a front-end to the BOOTSTRAP process assessment and improvement methodology. Hence the process maturity aspect has been reused from BOOTSTRAP. The technology maturity information has been associated with the technology itself, unlike in the SEPG (Software Engineering Process Group) approach [Fowler and Rifkin 90]. Technology information acquisition has been facilitated by the improvement function already included with the BOOTSTRAP methodology. The figure below illustrates how process maturation decisions are supported by technology maturity information. The following figure demonstrates the 2M view on technology transfer.

Procedure

An example of the 2M improvement procedure:

(Step 1a) A process maturity assessment methodology is used to determine the maturity level of the process. This gives an indication of generic software process areas for which *the process is ready* for technology transfer. The indication is general and gives the necessary (not sufficient) conditions only.

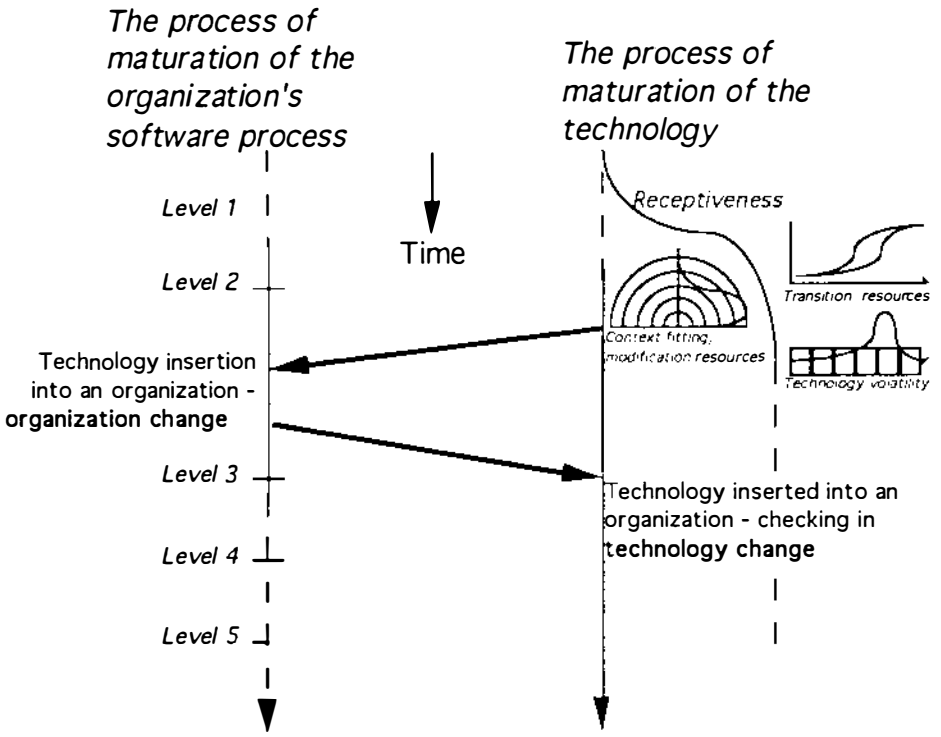
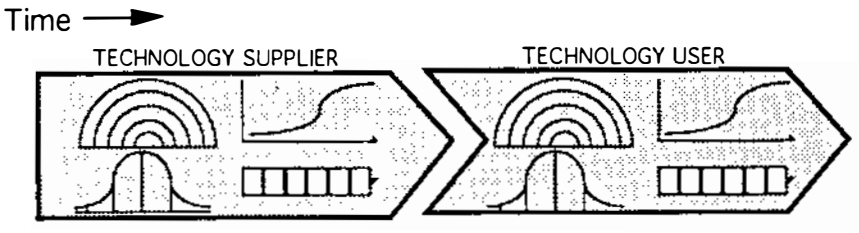
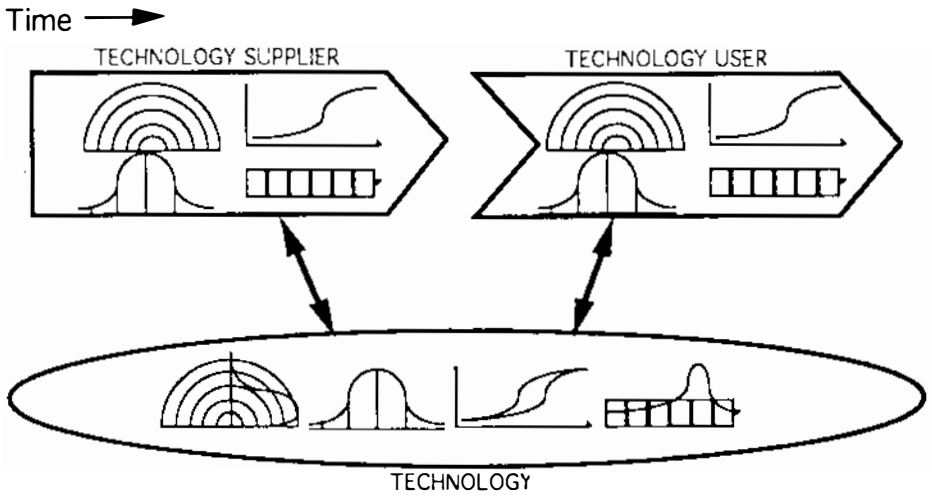


Figure 1. The two maturity processes of 2M. The process of maturation of the organization's software process (left) uses the BOOTSTRAP five-level maturity scale. The technology development process (right) uses the receptiveness maturity scale and the timeline maturity scale, and provides more information on technology context, resources needed for technology transition, resources needed for technology modification, and technology volatility. All this information is critical when deciding about technology implementation. Technology implementation is marked with the thick arrow heading left. The icons in top-right corner mark the various models, described later in this paper, supporting the information. Specific technology data are reported on the performed (or not) technology insertions (thick arrow heading right).



(a)



(b)

Figure 2. A traditional (a) and the 2M approach (b) to technology transfer. 2M allows for explicit standards of process and technology maturity, simpler maturation decision processes, and explicit interpretation of technology diffusion. The icons mark the various models described later in this paper. The arrows mark interactions between the process and technology maturation processes.

(Step 1b) A metrics framework, based on technology models, supports further formulation of the requirements for process maturation.

(Step 2a) A technology maturity methodology is used to determine whether there exist a corresponding *instance of technology*

that is ready for technology transfer, i.e., with a respective maturity in the sense of its receptiveness, organizational context, resources needed for technology transition, resources needed for technology modification, and technology volatility. If the answer is positive than process improvement steps (hopefully leading to its maturation) are proposed. Otherwise technology maturation steps are proposed. The outcome depends on the tradeoffs linking the two. An interleaved process of organizations and technology maturation is possible.

(Step 2b) A metrics framework, based on technology models, supports further specification of the technology requirements for technology maturation.

The technology maturity methodology is parametrized by the results obtained from (Step 1a-b), hence the steps have to be performed in the indicated order.

3. Process maturity

Below a selection of approaches to an organization's software process maturity is outlined: SEI's CMM, value-added maturity (Lean Enterprise Management), BOOTSTRAP, ISO's SPICE, and the 2M approach. More attention is given to BOOTSTRAP, from which 2M originated.

CMM

The SEI Capability Maturity model [Paulk et al. 93a, Paulk et al. 93b] is a staged process improvement including five consecutive levels of capability of the organization of large government/contractual software developments: (1) Initial, (2) Repeatable, (3) Defined, (4) Managed, (5) Optimizing. The Optimizing level represents a continuous process improvement. The goal of the approach is to achieve the Optimizing level through defect prevention, technology innovation, and process change management. The model is supported by an assessment procedure used to determine the organization's current level of software process maturity and to help set improvement areas. The levels are expressed in terms of key process areas - clusters of process functions. Improvement is achieved by action plans for poor key process areas. The approach is based on the organizational maturity model by Likert [Likert 67] and the quality management maturity model by Crosby [Crosby 79]. A key point of this approach is that there are process areas that will improve the process.

Value-added maturity (Lean Enterprise Management)

The concept of the value-added maturity, also known as Lean Enterprise Management [Womack et al. 90], requires that production concentrates on value-added activities and eliminates or reduces not-value-added activities. The goal is to tailor the process to the product needs, and to use the minimal set of activities needed. The particular methods used with the approach include technology management, quality management, human-centered management, decentralized organization, supplier and customer integration, internationalization, regionalization. A key point of this approach is that the process can be tailored to a class of problems.

BOOTSTRAP

The BOOTSTRAP process assessment and improvement methodology [Bicego et al. 93, Kuvaja et al. 94] combined the key-process-area and the value-added approaches to offer a balanced process maturity structure. It was created by an ESPRIT project of the same name, especially focused onto the European software industry. BOOTSTRAP took into account a selection of methodologies and software process types and especially international software standards applied in Europe. The mission of BOOTSTRAP was to lay the groundwork for European technology transfer standards and common practices. The main goal was to speed up the application of software engineering technology in the European software industry. The project took the SEI maturity model as the basic reference and extended it in several ways. The methodology was then applied to a number of European companies in order to define their maturity levels and capability profiles, and to generate the improvement plans.

The BOOTSTRAP assessment methodology describes the assessment process, determines where an organization stands (maturity level), identifies its strengths and weaknesses, and offers improvement guidelines (action plans). The BOOTSTRAP methodology evolved from the conventional and general approaches of the SEI maturity model and the ISO 9001 quality standard, to be then extended with elements, based on other standards, allowing for more satisfaction of users' needs and for more integration of assessment with improvement. The process development scale is 5-level, similar to the CMM's, though with finer granularity, extended to quartiles. The key area structure created in this way is implementation-independent and directly referred to the process standards. BOOTSTRAP provides 2M with more flexibility which is necessary for the value-added interpretation. Apart from the finer scale, more granularity has been achieved by the decomposition of process areas into standard subareas.

The project was concluded in 1993. Since then its members founded the BOOTSTRAP Institute as a new EEIG (European Economic Interest Group)

type non-profit organization. The purpose of this new legal body is to take care of the continuous development of the BOOTSTRAP methodology, promote, support, manage and control it's use and disseminate status information of the European software production for helping in its improvement. The main objectives in the development and use activities are to keep the methodology up to date, spread its use as wide as possible, manage automatic data collection and database services, and train the assessors in order to guarantee that all the assessments will fulfill the same quality standards. The dissemination activity will mainly produce periodically published summaries and trend reports based on statistical analyses of the assessment data base.

ISO-SPICE

The Software Process Improvement and Capability dEtermination (SPICE) initiative of the International Standards Organization (ISO) [Paulk and Konrad 94] is building on experiences of BOOTSTRAP as well as other methodologies. Its draft versions introduce, for instance, a two-dimensional scale of conformance and effectiveness which may be used to combine the process area and value-added views of process maturity in a way corresponding to the BOOTSTRAP approach. The initiative is in its preliminary stage.

2M

2M employs the BOOTSTRAP process assessment and improvement methodology. Interfaced with the technology maturity dimension, 2M receives more improvement flavor, and improvement plans are more adequate for the organization's state and technology availability. The process area aspect of BOOTSTRAP is more suitable for the second- and third-party process evaluations. The lean management aspect is more suitable for the first-party assessments.

4. Technology maturity

BOOTSTRAP, as well as other common process maturity frameworks, includes a primitive technology representation in the form of key process areas. In 2M, this part of the methodology has been considerably extended with the technology maturity framework described here.

Two classes of technology transition models are considered here, both supporting concepts of technology maturation used in 2M. These are specific and generic models. The former show how a technology is developed or adopted in the context of a particular organization. The latter

show how a technology develops in the context of its entire life cycle and in the context of many organizations and their software processes.

Generic models directly support the concept of technology maturity used in this paper. Four models are presented below and each of them constitutes certain technology maturity model. The figure below lists the models along with the icons used throughout the paper.

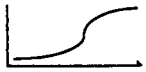
Specific models of technology maturity	Generic parametrized models of technology maturity
 IDA timeline model of technology maturation [Redwine 84]	 Technology maturation across different social contexts and industry types
 The commitment model of new technology adoption [Rogers 83]	 A family of commitment models for a class of technology adopters
 Receptiveness to new technology [Rogers 83]	 Receptiveness to new technology - for a narrow class of technology adopters
 The "rainbow" layered contexts behavioral model [Curtis et al. 88]	 The layered model with layer impact estimation

Figure 3. Technology maturation and transition models.

The IDA timeline model

The timeline model by the Institute for Defense Analysis (IDA) [Redwine 84] defines a sequence of stages of technology maturation: (A) Awareness, (B) Basic Research, (C) Concept Formulation, (D) Development and Extension, (E) Internal Enhancement and Exploration, (F) External Enhancement and Exploration, (G) General Popularization. Technology maturity states are determined against this scale. Stage (G) represents 40% to 70% market penetration. The goal is to achieve the General Popularization stage. The stages are expressed in terms of methods, media,

scope and commitment in sharing technology by different organizations. Particular critical factors - facilitators and inhibitors of technology maturation - are used as additional indicators. A key point of this approach is that there are ways of sharing technology by different organizations that avoid redeveloping it across different contexts.

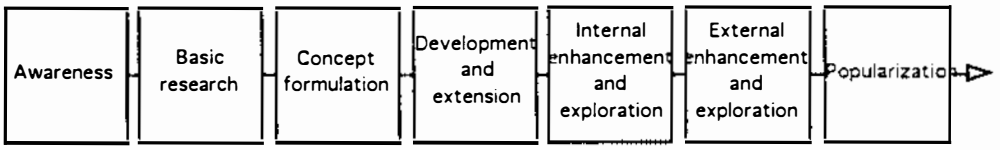


Figure 4. Technology maturation stages of the IDA timeline model.

The IDA timeline model has been used for general technologies such as software engineering, compiler construction technology, software metrics, etc. 2M uses it also in a parametrized version, to represent technology maturation across different social contexts and industry types.

The commitment model of technology adoption

The commitment model [Rogers 83] considers the commitments which individuals and groups make to the adoption of new technologies. The following phases are distinguished: Contact, Awareness, Understanding, Trial Use, Adoption, Institutionalization. The phases are marked on an S-shaped curve demonstrating the differences in commitments vs. the relative speed of phase transitions. There are differences in orientations, motivations, and responsibilities between different persons, groups, and organizations. These differences impact the shape of the commitment curve (degree of commitment and relative speed of phase transitions), and may be linked to the reinvention effect.

In 2M, the model is used in its original version for an individual technology and individual adoption context, and in a parametrized version showing the adoption process for different technologies, social contexts and industry types.

The "rainbow" layered behavioral model

The differences between analytic focuses of persons and groups within an organization are exemplified by the "rainbow" layered behavioral model [Curtis et al. 88]. The model indicates analytic focuses for different levels

within the organization: person - cognition and motivation; team and project - group dynamics; organization and business milieu - organizational behavior. For example, technologies may require different

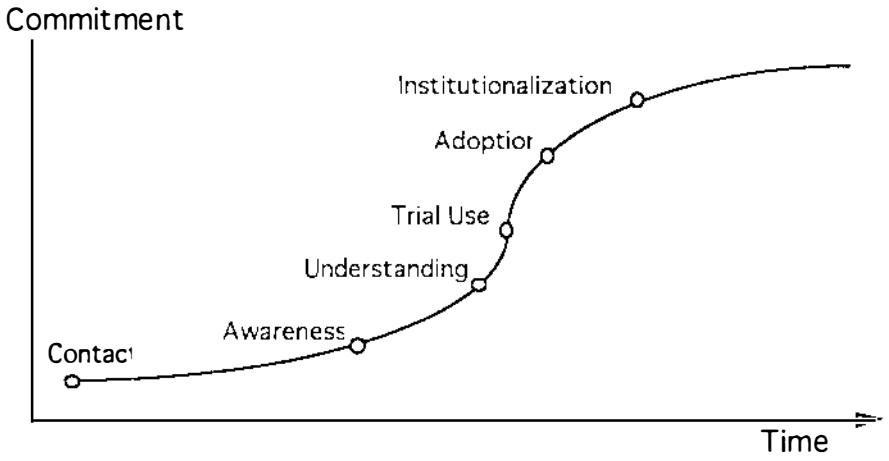


Figure 5. The commitment model.

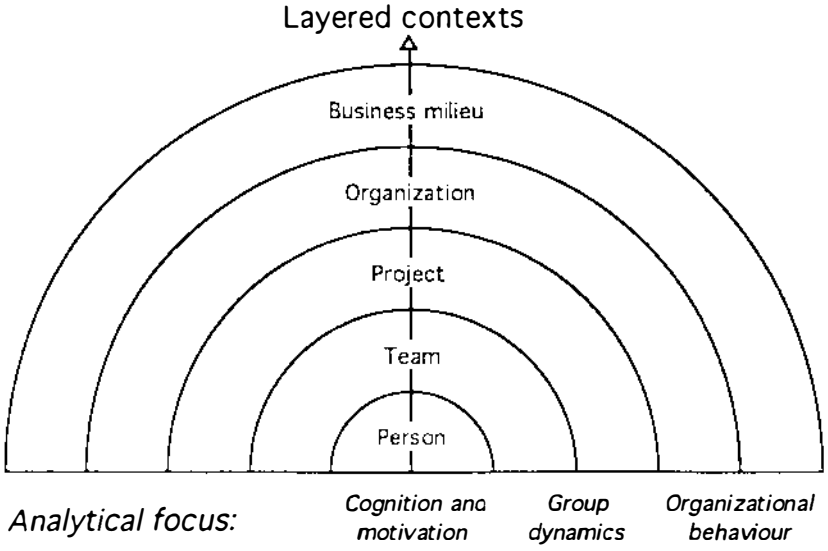


Figure 6. The "rainbow" layered behavioral model.

learning scales [Adler and Shenhar 90] and advocacy patterns, in terms of the commitment of different levels: broad-based support, technical staff, middle management, top management [Bayer and Melone 89]. The participants in the technology development life cycle (persons, groups, and organizations) perform value-adding activities which are critical for technology maturity. Depending on group or organization context the following may be different for different technologies: information needs, time frames, skills and procedures, strategy and culture requirements, balance between team independence and organizational buy-in, success criteria.

As with other models, 2M offers a parametrized version of the model. The version shows how different technologies and industry types affect the different social layers. In particular, the model shows whose work is affected most by the technology change.

Technology receptiveness models

The differences in reactiveness to new technologies are exemplified by the "bell curve" model of technology receptiveness [Rogers 83] distinguishing the following groups: Innovators (2% of population), Early Adopters (14%), Early Majority (34%), Late Majority (34%), Laggards (16%).

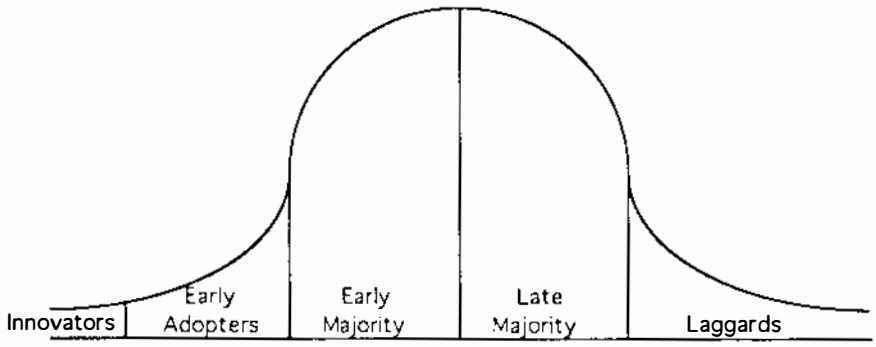


Figure 7. The "bell curve" model of technology receptiveness.

2M does not parametrize this model. Nevertheless the adjectives "early" and "late" of group labels receive different meanings for different technologies, technology transition contexts, etc.

2M

2M employs the generic models to define overall technology maturity in a parametrized way. The specific models are used for assessing technology maturity on an organization scale. Both models are important to evaluate the risk associated with technology transition.

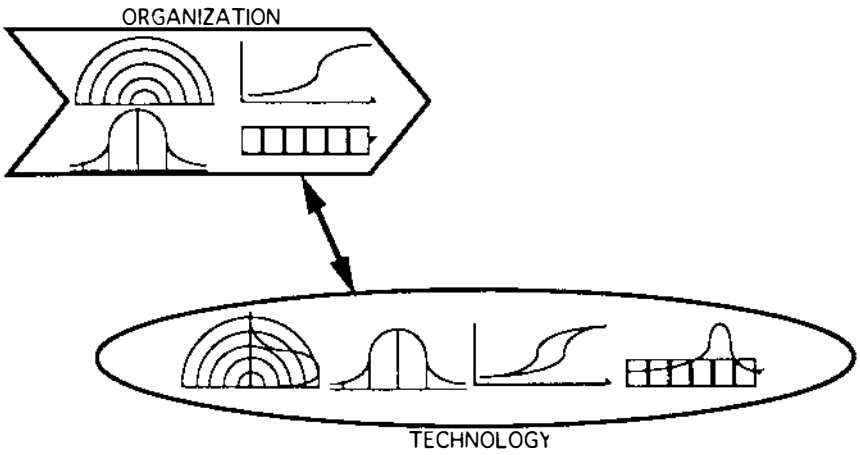


Figure 8. The 2M approach to technology maturity modeling: A generic technology is represented in a parametrized way covering multiple contexts of technology use. Organizations are represented with specific maturity models.

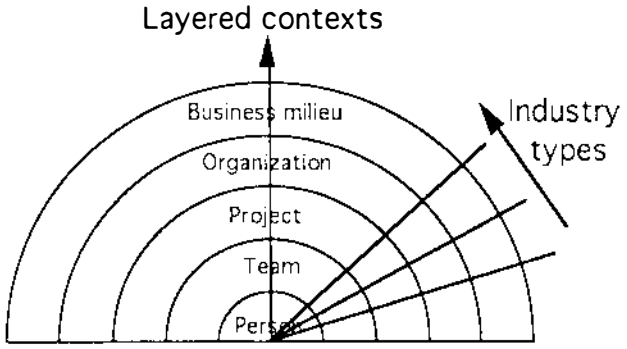


Figure 9. Technology coordinates in 2M. Industry types distinguish between various organization sizes, process types and product types.

The generic models include also mechanisms for parameter selection. In this way the combined technology model can represent a wide collection of phenomena of technology transition between different organizations. The "push-pull" transaction model and the multiple context "food chain" model [Rogers 83, Przybylinski et al. 91] are examples of descriptions of such phenomena. They are focused onto the roles of advocates-producers and receptors-consumers of technology, and the evolutionary selection of maturing technologies. Technology parameters include layered social contexts as well as organization sizes, process types and product types (cf. the figure above).

5. Metrics for process and technology requirements

Metrics play important role in 2M. They refine and extend the process and maturity models described above. It is stressed that metrics used in process specification are explicitly related to those used with technology specification, and that appropriate assessments are given of the risks associated with technology solutions.

Process requirements

A process maturity assessment constitutes a general reference for specifying process needs. Detailed process requirements must be obtained with more refined methods discussed in this section. Requirements can be divided into functions and nonfunctional metrics. The required process functions are most often set with enough detail by the process maturity assessment. In 2M the functional decomposition comes from BOOTSTRAP as illustrated in the following figure.

Metrics are used to specify preferences, compare and optimize solutions. Some overall "strengths and weaknesses" measures of the indicated functions are given by the software process assessment. Further metrics must be delivered from the maturity data of both the process and technology. 2M requires that process specifications refine the BOOTSTRAP process attributes. A default set of metrics is proposed by 2M, based on the introduced technology models. Other metrics may be identified, in an interactive way. They will satisfy the need for refining, modifying, and extending the default metrics to cover the organization's specifics. Methods outlined below next paragraphs may be used.

For instance, climbing up the commitment curve involves a number of effects addressing the human side of technology change, such as the defreezing/freezing effect or human reaction profile [Bridges 88]. The

effects motivate the need for resources and support. It is required that the necessary resources and support are evaluated and planned for before a technology insertion takes place. For mature technologies the insertion resources and support can be assessed with satisfying certainty. Maturity metrics sensitive to the above effects must be used if the effects have critical impact on technology transition. For immature technologies tradeoffs between process and technology maturation should be analyzed. Other important metrics may represent effects of concurrent technology insertion, economies and diseconomies of scale, etc.

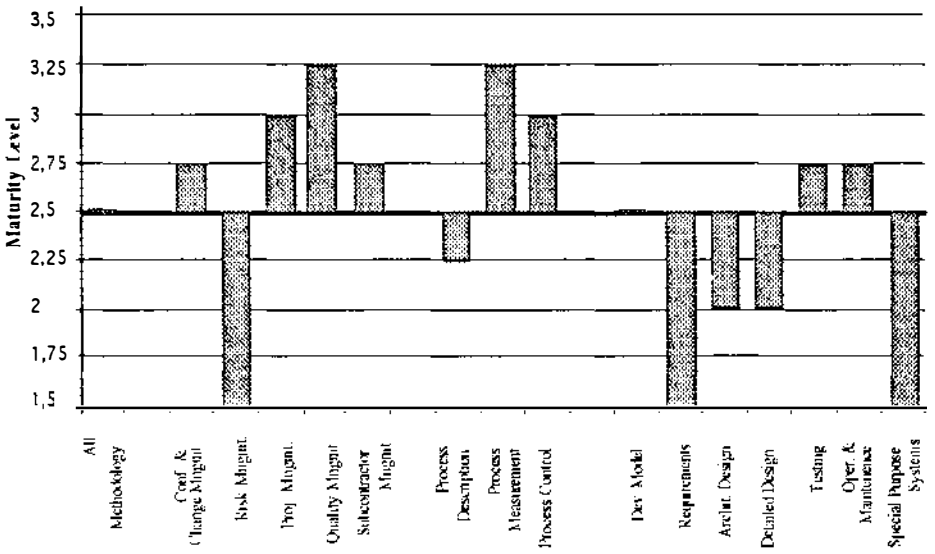


Figure 10. An example of the "strengths and weaknesses" profile resulting from a BOOTSTRAP process maturity assessment.

GQM

GQM (Goal/Question/Metric) [Basili and Rombach 87, Basili and Weiss 84] is a known approach to metrics indentification. It is based on the assumption that only an integrated approach, in the context of appropriate models, can effectively contribute to the body of knowledge about the quality of the product or process. If provided with appropriate models, GQM is known to create meaningful and reliable metrics. GQM is a systematic approach for defining and evaluating operational goals, for tailoring and integrating goals with models, and for defining measurements. The models represent software processes, products, and quality perspectives of interest. They are based on the specific needs of the project, the customer, and the organization. The goals, defined in an

operational way, are refined into a set of quantifiable questions. The questions are then used to extract the appropriate information from the models. The questions and models, in turn, define a specific set of metrics and data for collection, and provide a framework for interpretation. As a part of the Quality Improvement Paradigm/Experience Factory (QIP/EF) [Basili and Rombach 88], GQM has been used for building a continually improving organization based on evolving goals and assessments of the status against the goals.

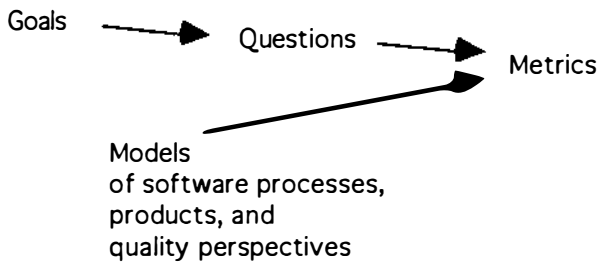


Figure 11. GQM metrics derivation.

DbO

The DbO methodology [Gilb and Krzanik 87, Gilb 88] emphasizes the role of the underlying models, validity and reliability of metrics. As opposed to GQM, DbO represents goals directly in terms of metrics, and requires that a risk assessment is included with each metric specification. It also emphasizes the obligation of consistent use of the metrics for expressing characteristics of technology as well as of the process. DbO has been created as an integrated methodology supporting continually improving organizations.

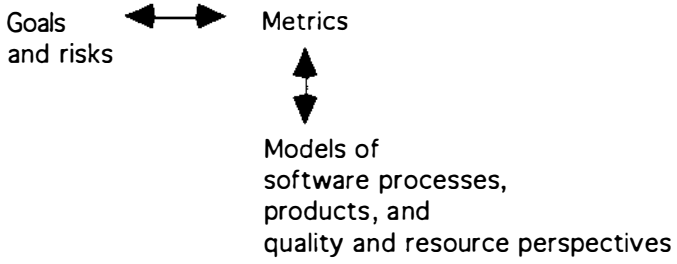


Figure 12. Interactive DbO metrics derivation.

Technology requirements

In the same way as process requirements detail the required process maturity development, technology requirements detail the technology maturity development. Explicit, measurable technology specifications are necessary to resolve differences between the requirements between technology users and technology providers. Both cases are served by 2M. The default strategy for users is TQM. Other different cases to resolve include [Przybylinski et al. 91]: single technology insertion into one organization's unit, inserting a technology into many units concurrently or sequentially, inserting many technologies into one or many units concurrently or sequentially, inserting the same technology into a different organization. Technology maturation requirements will generally be different for these various types of technology insertion.

The success of transferring technology into the process is determined by process change characteristics such as the required resources and support. By supplying explicit technology maturation metrics, 2M attempts to associate as many of these characteristics as possible with the technology in order to minimize the modeling work on the process side and to force the identification of consistent specific models. We observe that this leads to simpler and more effective process maturation decisions. The approach is different from, for example, the SEPG approach [Fowler and Rifkin 90]. In 2M the information is practically available through the reporting facilities of BOOTSTRAP database (BootBase).

TQM

TQM [Feigenbaum 91, Ishikawa 85] is a style of management aimed at achieving long-term success by linking quality with customer satisfaction. All members of the organization take part in the improvement process. The controls are opportunistic and triggered by the organization's culture. The management role is to commit and empower all staff in most of their activities. Small technology increments are preferred since impacts may be uncertain. In cases of more sizable increments, experimentation with improvement methods is necessary and development of predictive models, for example, according to the Plan-Do-Check-Act approach [Deming 86].

6. Current state of 2M implementation

The 2M framework includes the following constituents:

- the BOOTSTRAP process maturity methodology, enforcing a framework for process requirements specification and verification,
- the technology maturity methodology derived from models of technology maturity,
- the metrics methodology for specifying process and technology characteristics,
- an interface to BootBase (the BOOTSTRAP database) storing and manitaining specific process and technology data.

Apart from those constituents, the framework supports a database of technology maturity and metrics not available in present configuration of BootBase. BootBase is used under an appropriate security policy. The framework also includes training and information distribution services.

There are two basic activities of 2M. One is permanent collection of process and technology data to support the process and technology maturity models. The other is matching the requirements against the available technology and maturation alternatives, and developing improvement plan outlines. Both activities are risk-conscious.

Currently the method covers the TQM policy only, supporting the technology users. Provider policies are not supported. Among the different transition policies by the number of technologies and of organizational units, only the simplest are currently supported. It is believed that more advanced policies shall require new explicit maturation models to be included into the framework.

7. Future directions

The framework will be developed to cover for a more extensive selection of technology maturation policies. Most of the available resources will be invested in user interface, to allow more effective user feedback. Further experimentation will be performed to verify the design assumptions of the framework.

8. Conclusions

At the age of 2 of its exploitation, the BOOTSTRAP methodology gains popularity. Requests are raised for advancing the process improvement practice. The 2M front-end to BOOTSTRAP supports process

improvement by helping in understanding and implementation of the improvement processes.

References

[Adler and Shenhar 90] Adler, P.S., and Shenhar, A., Adapting your technological base: The organizational challenge. *Sloan Management Review* Vol. 32, No.1, pp. 25-37 (Fall 1990).

[Basili and Rombach 87] Basili, V.R., and Rombach, H.D., Tailoring the software process to project goals and environments. *Proceedings of the Ninth Intl. Conf. on Software Engineering* , Monterey, CA, pp. 471-485. (March 30-April 2, 1987).

[Basili and Rombach 88] Basili, V.R., and Rombach, H.D., The TAME project: Towards improvement-oriented software environments. *IEEE Trans. on Software Engineering* Vol. SE-14, No. 6, pp. 758-773 (June 1988).

[Basili and Weiss 84] Basili, V.R., and Weiss, D.M., A methodology for collecting valid software engineering data. *IEEE Trans. on Software Engineering* Vol. SE-10, No. 6, pp. 728-738 (November 1984).

[Bayer and Melone 89] Bayer, J., and Melone, N., Adoption of software engineering innovations in organizations. *Technical Report* CMU/SEI-89-TR-17, ADA211573. Software Engineering Institute (April 1989).

[Bicego et al. 93] Bicego, A., Cacchia, R., Haase, V., Koch, G., Krzanik, L., Kuvaja, P., Lancellotti, R., Maiocchi, M., Messnarz, R., Saukkonen, S., Schynoll, W., and Similä, J., BOOTSTRAP: Europe's assessment method. *IEEE Software*, Vol. 10, Nr. 3 (May 1993), pp. 93-95.

[Bridges 88] Bridges, W., *Surviving Corporate Transformations*. Doubleday, New York, NY (1988).

[Crosby 79] Crosby, P.B., *Quality is Free: The Art of Making Quality Certain*. McGraw-Hill Book Co., New York (1979).

[Curtis et al. 88] Curtis, B., Krasner, H., and Iscoe, N., A field study of the software design process for large systems. *Comm. ACM*, Vol. 31, No. 11, pp. 1268-1287 (Nov. 1988).

[Deming 86] Deming, W.E., *Out of the Crisis*. Center for Advanced Study, Massachusetts Institute of Technology, Cambridge Ma (1986).

[ESA PSS-05-0] *ESA Software Engineering Standards ESA PSS-05-0*. Issue 2. ESA Board for Software Standardisation and Control, European Space Agency, Paris (February 1991).

[Feigenbaum 91] Feigenbaum, A.V., *Total Quality Control*. McGraw-Hill Book Co., New York (1991).

[Fowler and Rifkin 90] Fowler, P., and Rifkin, S., Software Engineering Process Group Guide. *Technical Report* CMU/SEI-90-TR-24, Software Engineering Institute (September 1990).

[Gilb 88] Gilb, T., *Principles of Software Engineering Management*. Addison-Wesley, New York (1988).

[Gilb and Krzanik 87] Gilb, T., and Krzanik, L., Design by Objectives: A basis for automated software engineering design. *Software Engineering Notes*. Vol. 12, No. 2 (April 1987), pp. 42-50.

[Ishikawa 85] Ishikawa, K., *What is Total Quality Control? The Japanese Way*. Prentice-Hall. Inc., Englewood Cliffs, NJ (1985).

[ISO 9001] *ISO 9001. Quality Systems. Model for Quality Assurance in Design/Development, Production, Installation and Servicing*. International Organization for Standardization, Geneve (1989).

[Kuvaja et al. 94] Kuvaja, P., Similä, J., Krzanik, L., Bicego, A., Koch, G., and Saukkonen, S., *Software Process Assessment and Improvement: The BOOTSTRAP Approach*. Blackwell, Oxford, UK, and Cambridge, MA (1994).

[Leonard-Barton 88] Leonard-Barton, D., Implementation as mutual adaptation of technology and organization. *Research Policy* Vol. 17, No. 5, pp. 251-267 (October 1988).

[Likert 67] Likert , R., *The Human Organization: Its Management and Value*. McGraw-Hill Book Co., New York (1967).

[Paulk and Konrad 94] Paulk, M.C., and Konrad, M.D., An overview of ISO's SPICE project. *American Programmer* (Feb. 1994).

[Paulk et al. 93a] Paulk, M.C., Curtis, B., Chrissis, M.B., and Weber, C.V., Capability Maturity Model for Software, Version 1.1. *Technical Report* CMU/SEI-93-TR-

024. Software Engineering Institute (February 1993).

[Paulk et al. 93b] Paulk, M.C., Weber, C.V., Garcia, S.M., Chrissis, M.B., and Bush, M., Key Practices of the Capability Maturity Model, Version 1.1. *Technical Report* CMU/SEI-93-TR-025. Software Engineering Institute (February 1993).

[Przybylinski et al. 91] Przybylinski, S.M., Fowler, P.J., and Maher, J.H., *Software Technology Transition*. Tutorial presented at the 13th Intl. Conf. on Software Engg, Software Engineering Institute (May 12, 1991).

[Redwine 84] Redwine, S.T., Jr., Becker, L.G., Marmor-Squires, A.B., Martin, R.J., Nash, S.H., and Riddle, W.E., DoD related software technology requirements, practices, and prospects for the future. *Technical Report*, IDA Paper P-1788, IDA (June 1984).

[Rogers 83] Rogers, E.M., *Diffusion of Innovation*. Free Press, New York, NY (1983).

[Womack et al. 90] Womack, J.P., Jones, D.T., and Roos, D., *The Machine that Changed the World*. Rawson Associates, New York (1990).

List of abbreviations

2M	The Double-Maturity framework of process and technology maturity
BootBase	The BOOTSTRAP database of maturity information
BOOTSTRAP	A process assesment and improvement methodology of Bootstrap Institute
CMM	Capability Maturity Model of SEI
DbO	Design by Objectives
EEIG	European Economic Interest Group
ESA	European Space Agency
GQM	Goal/Question/Metric
IDA	Institute for Defense Analysis
ISO	International Standards Organization
QIP/EF	Quality Improvement Paradigm/Experience Factory
SEI	Software Engineering Institute
SEPG	Software Engineering Process Group
SPICE	Software Process Improvement and Capability dEtermination, a methodology of ISO
TQM	Total Quality Management

"Overcommitment - Escaping Chaos"

Neil Potter and Mary Sakry
The Process Group

Description

Overcommitment may be causing many of your quality and process improvement problems. Organizations exhibiting the characteristics of SEI level 1 usually have little time available to do anything but coding and some testing.

When commitments are made to customers and these commitments cause unrealistic schedules, there is often no time to improve. Organizational stress increases, expensive mistakes are made and customers become dissatisfied.

The commitment process is used by an organization to make and alter commitments. When followed it allows an organization to maintain its sanity and manage customer expectations. It is one of the most essential elements of successful long-term process improvement and SEI level 2.

About the presenters

Neil Potter and Mary Sakry of The Process Group specialize in software engineering process improvement. They are both authorized by the SEI to conduct licensed SEI Vendor-assisted process assessments.

Mary has 18 years of software development, project management and software process improvement experience. She was the first person authorized by the SEI to conduct process assessments and is one of the most experienced in the world.

Neil has nine years of experience in software design, engineering and process management. He has consulted in process improvement in the U.S. and overseas. He created and managed a Software Engineering Process Group (SEPG) for a Fortune 500 company.

Quality Improvement and Organizational Change

J. Audrey Neal

ACTC Technologies Inc.
350, 6715 - 8th Street N.E.
Calgary, Alberta, Canada T2E 7H7
Business: (403) 295 - 5900
FAX: (403) 295 - 5999
Internet: audrey.neal@actc.ab.ca

Abstract

Standards such as ISO 9000 are becoming international yardsticks by which company quality management is measured. Increasingly, companies without certification to these standards will be excluded from the marketplace. Implementing programs to achieve quality standards is a major organizational commitment, and often involves organizational change. This paper explores both the benefits and the challenges of standards-driven quality-improvement efforts, as compared to more open-ended efforts such as Total Quality Management (TQM) or continuous improvement. Current theories on personal motivation, organizational development, and organizational culture and cultural change are cited as they apply. Practical application is presented based on the author's experience in leading an organization-wide process improvement effort based on the SEI Capability Maturity Model. The conclusion is drawn that, regardless of the program selected, consideration must be given to the personal and cultural aspects of the change program.

Keywords and Phrases: ISO 9000, TQM, SEI CMM, quality improvement, continuous improvement, organizational development, cultural change, flow, personal motivation, quality standards.

Biographical: Audrey Neal is the Director of Process Management at ACTC Technologies Inc., a custom software development company in Calgary, Alberta, Canada. For the last three years Audrey has been involved in leading the organizational initiative to improve software process according to the SEI Capability Maturity Model. Prior to that time, Audrey spent several years as a project administrator for software development, and before that was the head of technical documentation at ACTC. Audrey holds a Bachelor's degree in education, a Bachelor's degree in science, and is currently completing a Master's degree in business administration at the University of Calgary.

Introduction and Perspective

For the last three years I have been the manager in charge of quality improvement programs for ACTC Technologies Inc. Our program is based on the Software Engineering Institute's Capability Maturity Model (commonly referred to as the SEI CMM). I have watched with interest as the International Standards Organization's (ISO) 9000 quality standards have gained in prominence, and as pressures have increased on companies to compete for quality awards such as the Malcolm Baldrige in the U.S. and the Canada Award for Business Excellence in Canada. Many roads to quality improvement are being offered, and there is significant pressure on companies to adopt one or more approaches.

My personal experience in this area leads me to be very interested in the whole phenomenon of quality improvement – particularly in what works, what doesn't, and what realistic and beneficial outcomes can be expected of such efforts. During the last year I had the opportunity to take a leave of absence from work to pursue an MBA degree at the University of Calgary. Hence, I had the opportunity to step back from *doing* quality program implementation, to *reflecting* on how it has gone and analyzing it from some theoretical perspectives. This paper is, thus, an attempt to marry my academic background and personal experience.

In this paper I discuss three main topics. First, I present the various approaches to quality improvement and attempt to distinguish between them on the basis of their roles and underlying philosophies. I discuss the pros and cons of programs that are aimed at meeting standards, and compare them with approaches that are more open-ended (such as Total Quality Management or TQM). Secondly, I present our experience at ACTC in implementing a quality improvement program based on the SEI CMM framework. Finally, I look at quality improvement programs as instances of organizational change, and discuss theories of personal motivation and organizational culture as they apply. I draw the conclusion that, regardless of the program selected, consideration must be given to the personal and cultural aspects of the change program.

Finally, I would like to acknowledge and thank the many people who have helped me shape these concepts and capture them in this paper: Dr. Michael Waterhouse, Dr. Michelle Fraser, Karen Sharpe, Gillian Hobbs, Brian Westcott, Al Hussein, Margo Elliott, Mary Jo Skeet, Vivian Hindbo, Fiona Koether – and all the staff of ACTC Technologies Inc.

Alternative Approaches to Quality Improvement

Many approaches to quality improvement have been developed in recent (and not so recent) years. All of them share the goal of improving quality, although the motivation to pursue some programs may be based on strategic or competitive reasons rather than loftier quality goals. The ISO 9000 series of standards, for example, has been described as serving as a non-tariff barrier to trade with Europe – a concept which has nothing to do with quality. Increasingly, however, certification to standards of one sort or another is becoming an entry criteria for companies wishing to be in the marketplace at all.

The pursuit of quality is not only the right thing to do, but a necessary thing in today's world of intense global competition. The goal of being able to demonstrate quality, particularly in software, is also very important. The software industry has been notorious for its stories of out-of-control costs and vast schedule overruns. This has led to strong pressures within the industry for ways to improve not only quality, but predictability in terms of final cost and schedule. The philosophy of "building quality in" and "doing it right the first time" is all the more important for software due to its intangible nature. Consequently, standards and process improvement models and criteria share a common assumption: that by improving the *process* by which software is created, a higher quality *product* will result, with the additional benefits of allowing for more accurate predictions of schedule and cost.

What are the approaches and standards available to guide process improvement? Perhaps most widely known is Total Quality Management (TQM), which is not a standard, but a set of structured methods for achieving quality improvement in line with customer needs. Quality standards that apply generally (not just to software) include the International Standards Organization (ISO) 9000 series of standards published in 1987. These standards cover an organization's quality management system. Awards such as the Deming Award and the U.S. Malcolm Baldrige National Quality Award attempt to recognize companies that show excellence in comprehensive quality management by defining a set of qualifying criteria. Other awards such as the Canada Award for Business Excellent in Total Quality are similar. These award criteria can be thought of as a kind of standard for Total Quality Management. Finally, standards or frameworks that apply to software include ISO 9000-3, the SEI Capability Maturity Model, Department of Defense Standards (DOD-STD 2167A, 2168, 7935A, and SDD498), IEEE/ANSI standards, and ISO/SPICE for process assessment.[1]

With this wide range of options, how is one to determine the best approach to quality improvement? What philosophy do they subscribe to? What outcomes do

[1] *Software Process Standards.* Software Quality & ISO 9000, July 1993, p.3.

they promise? The temptation is to get caught up in the panic to jump on the quality improvement bandwagon without really examining the premises and the potential results of these programs. What follows is an attempt to put them into perspective.

TQM

TQM can be defined as "an organization-wide, structured approach to managing the continuous improvement of processes and systems that enables an organization to meet or exceed the demands of its customers."[2] It is based on the premise that the desire of the customer is paramount, and that with a customer satisfaction focus and basic tools for continuous improvement, the organization can meet or exceed quality requirements. Another basic tenet of TQM is that it is the capability of the *system* that determines the level of quality that is produced, not the motivation of the individual worker. The system has a certain level of capability, characterized by statistical quality control ranges. If quality improvements are to be made, the system's capability must be enhanced. Thus, from the TQM point of view, it is the gap between the capability of the system and the needs of the customer that drives the improvement efforts, rather than an external standard. It is an open-ended program of system improvements, designed by the people who are working in the system.

Quality Standards

If the goal of quality improvement programs were strictly to address the gap between the quality desired (and defined) by the customer, and the quality capability of the current system, there would be no need for an external standard. In fact, the Japanese, who are meeting customer expectations admirably, are scrambling to qualify for the ISO quality standards – they have good systems for meeting the needs of customers, but they have not met the documentation requirements of the standard.[3] Therefore there must be additional motivations driving interest in the various standards:

1) They serve to tell *others* where you stand on the range of quality management practices – what I'll call certification and qualification, and
2) They serve to give a firm *target* for quality improvement programs.

Certification and Qualification

The attractiveness of a standard such as ISO is that it is a way to compare companies on important dimensions that are not necessarily readily visible to the purchaser. The advantage of independent certification such as ISO is that the purchasers no longer need to go to the time and expense of evaluating the qualifications of the competing firms themselves; they can rely on the ISO certification process to do that for them.

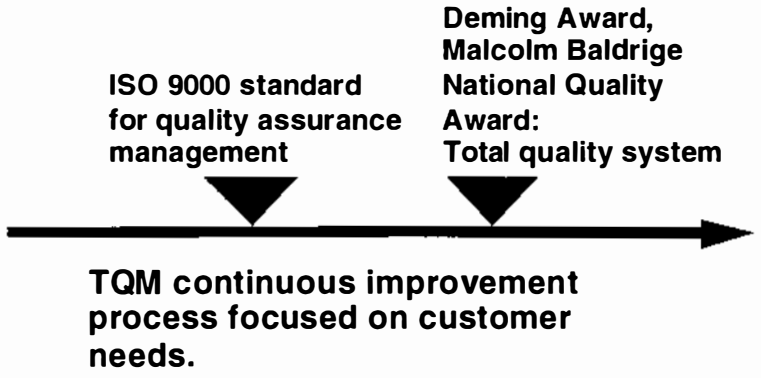
---

[2]Rankin, Lloyd, TQM Coordinator, Mount Royal College, Calgary, Alberta, in presentation to the Calgary Quality Council, June 1994.

[3]Hulme, D. *Japan Takes Standards to Heart.* Machine Design, August 27, 1993, p. 77.

Guidance for Quality Program Implementation
Aside from the external recognition that a firm receives from certification to a standard such as ISO, the standard itself can provide a guide to implementation of sound quality management systems and practices. This description of *what* needs to be in place gives a good target for organizations embarking on quality improvement programs, yet gives latitude for organization-specific implementations. The SEI CMM and Malcolm Baldrige Award criteria are also valuable guides for quality program implementation.

The following diagram shows a conceptual relationship between TQM, ISO and quality awards such as the Malcolm Baldrige. The idea is that whereas TQM gives tools and techniques for continuous improvement, the standards and award criteria can serve as targets along the way.[4]



Where does the SEI CMM fit in?
The SEI CMM attempts to combine the philosophy of TQM with the guidance of a framework. I see it as less of a standard and more of a model to guide continuous improvement. In contrast to the ISO 9000 standards which are static, the SEI CMM incorporates an emphasis on continuous improvement. It does, nonetheless, prescribe key practice areas that are fundamental to good software management and software engineering.

The Pros and Cons
As I mentioned above, there is a strong temptation to dive into one or more of these quality programs, if for no other reason than compelling business considerations. Often the program chosen is dictated by business necessity, yet it is still important to understand the dangers and potential pitfalls as well as the promised benefits of the various approaches.

---

[4]A caveat to this: it is not *necessarily* the case that the standards such as ISO are in line with the direction TQM would take a company. At the very least, companies undertaking initiatives shaped by standards need to keep a close watch on their customer satisfaction level to make sure that no disconnects are occurring as the programs unfold.

I will cite examples to illustrate where standards-driven or open-ended continuous improvement (TQM) programs may better fit with theory. In general, however, the following cautions apply:

- There is an ever present danger to start to focus on passing the test rather than making genuine process improvement. The pressures to get the recognition in the market can be acute. The section on personal motivation deals with the possible consequences of a superficial commitment.
- The pursuit of compliance with standards also runs a risk of setting up cultural incompatibilities in the organization: a TQM open-ended approach is more likely to result in improvements that fit the organization's culture; adopting an externally devised model can lead to cultural mismatches. This topic is discussed in the section on organizational culture and change.
- Using the TQM open-ended approach may be difficult for companies who need that extra bit of guidance in terms of what to do. The standards and SEI CMM embody a lot of information on best practice, which can be useful in getting started.

To that end, I offer the following synopsis of my experience, as well as some theoretical frameworks I have found useful in understanding the dynamics of quality improvement programs as change programs.

Our Successes and Challenges in Quality Program Implementation

ACTC Experience with SEI as a Process Improvement Initiative

As the Director of Process Management, I was involved from the beginning in ACTC's efforts to select and manage a software process improvement initiative. ACTC is a medium-sized software development company of about 90 people. Our business centers on custom software development, which often involves adopting the client's models for software process and project management. This gives us good insight into good practice from a range of companies, but does not help us build an ACTC owned process (although we have adopted as company-wide practices some of the project management practices and templates offered by our clients). In 1989, our Senior Vice President of Operations attended a conference in which the original SEI model was presented. He challenged us to use this as a model to guide our process improvement initiatives. We set up a group of senior engineers and project administrators called the Software Engineering Review Board (SERB) to study the model, address the questions, and determine how we should proceed in process improvement. We did a great deal of thinking and talking, and managed to build a good base of understanding of the issues, but we had difficulty focusing on a plan of action to mobilize improvements.

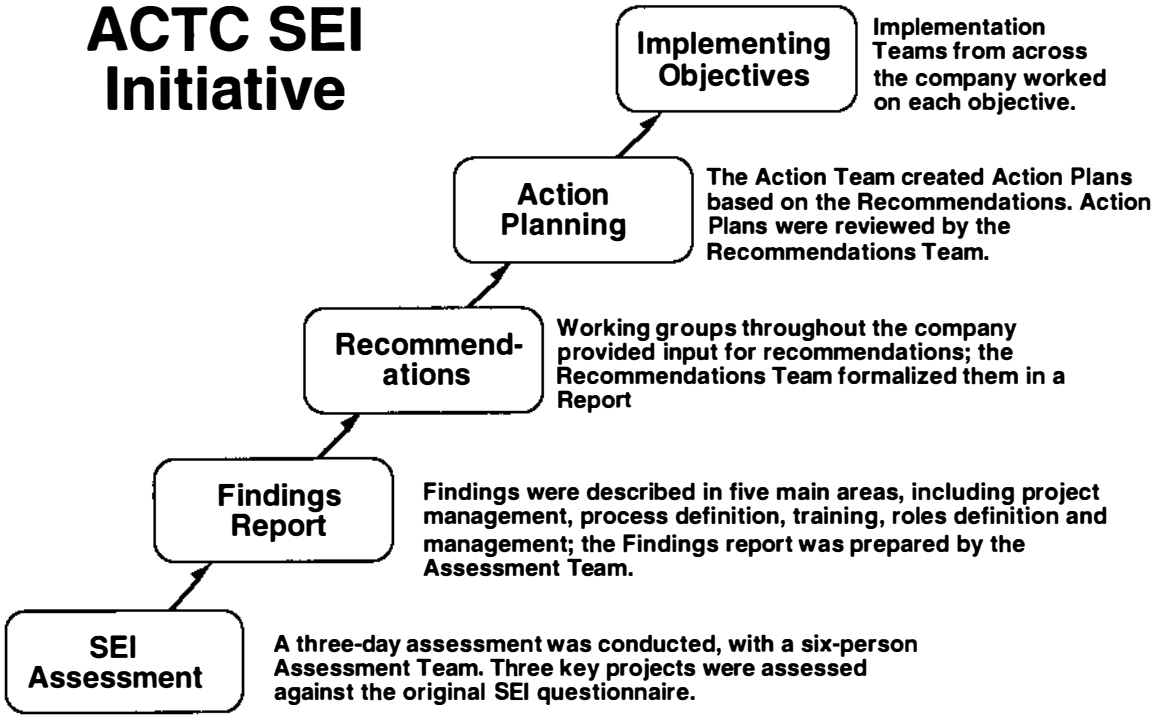
Why did we choose the SEI model as the basis of our program? Perhaps the reason is best stated the way the SEI itself explains the usefulness of the model:

> Although software engineers and managers often know their problems in great detail, they may disagree on which improvements are most important. Without an organized strategy for improvement, it is difficult to achieve consensus between management and the professional staff on what improvement activities to undertake first.[5]

In 1991 ACTC formally launched its SEI initiative with a mini-assessment conducted by qualified external assessors. The Assessment Team, made up of 4 external and 2 internal assessors, interviewed project leaders for three key projects, as well as functional area representatives (team members) throughout the development area. The Assessment Team prepared a Findings Report based on the model and the information provided in the interviews. This report was presented to the participants for initial feedback, and then presented to the company as a whole.

---

[5]Paulk, et. al, Capability Maturity Model for Software, Version 1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1993, p. 4.



The next steps were up to us. First we set up teams to work on recommendations. The whole company was asked to form working groups and make recommendations based on the information in the Findings Report. We formed a Recommendations Team (formed of senior engineers and project leaders/functional area leaders) which took all the input from the working groups and formalized recommendations in a Recommendations Report. We then formed an Action Team to turn the recommendations into actions. This team was smaller and consisted of the senior managers in each area affected by the recommendations. Seventeen objectives were developed based on the recommendations.

We then set up Implementation and Review teams to address each objective. These teams were made up of interested staff from all areas of the company. Each team was coordinated by a member of the Action Team, which oversaw the overall effort. The Implementation Teams were responsible for setting policy and defining processes for each objective area, subject to review by management and the review team. Thus the system was bottom-up, designed by the people working in the area. The output consisted of a document or series of documents setting in place policy, practice and templates or standards as appropriate. A system called the Software Process Documentation System (SPDS, called "spuds"), was set up to organize and maintain these documents, and provide a way to continuously improve them.

We made presentations to staff on a regular basis throughout this process. We ran introduction sessions for new processes and invited feedback. We initiated a Process Improvement Newsletter to capture and disseminate our learnings and inform people of improvement efforts and new process documents. PI News became a

monthly publication with many staff writing articles on process improvement in their areas.

We sustained an intensive improvement effort for about a year and a half, and managed to define a large number of processes that covered many of the SEI level 2 key process areas. We established coding standards, a change procedure mechanism, an estimation process as well as project initiation process, documented standards for software configuration management, comprehensive software quality assurance including reviews, code inspections, process and product audits and post mortem analysis. We also began collecting some metrics, mainly associated with reviews and inspections. Along the way we received extensive training in overall software quality, management of the software development process, software estimation and sizing, unit and system testing, reviews, metrics and continuous improvement methodologies. Although this training was offered initially on a one-shot basis, there are now plans to institutionalize the program.

The Results – Looking Back

Throughout the process we were concerned with the buy-in of everyone to the process. The formal assessment was designed to obtain buy-in from the project personnel by involving them in the generation of findings. Yet, even with this focus on building consensus, we had disagreement and serious questioning of the approach and the assumptions underlying it. The consensus built in the assessment was consensus on the findings – not consensus on the philosophy of the SEI or on the validity of the model. At that time we had not explicitly examined the philosophy behind the model, and consequently did not sell it sufficiently to build a common commitment.

Another weakness we encountered was the lack of management involvement in the initial assessment. The involvement of management came down to their role in visibly supporting the initiative. They were not interviewed as to their views on strengths or weaknesses/challenges facing the organization, presumably to avoid making the assessment appear overly influenced by management. This led to the comment of a senior manager in the final executive briefing: "I feel this has been thrown over the wall at me without any chance for me to have input into it – and yet I am expected to whole-heartedly support it."

We did have tremendous buy-in to the concept of things getting better – not necessarily agreement on what was required specifically for each key process area, but a great expectation that the process of improvement using the SEI model would solve our problems. This was both positive and negative – positive in that the willingness was there to change and invest in the effort, and negative in that the expectations for a silver bullet that would fix every problem were unrealistic and could not possibly be satisfied. This led to some disillusionment toward the whole process that was difficult to deal with.

There was excellent acceptance of the "megaprocesses" of continuous improvement, such as root cause analysis (fish-bone charts). These methods are now standard practice in the organization. In general, there was less acceptance of the specific processes we defined. There were lots of arguments as to why certain steps were included and not others, who should be involved and so on – an indication that we may have gone too far in defining some things or perhaps that we could have done better introducing the processes and rationale. In general, we may have put more emphasis on the defining of processes than was necessary, and too little emphasis on managing the organizational change we were undertaking.

SPDS proved to be too rigorous and time-consuming for us to handle in the context of ongoing project work. The system was basically sound, but we understaffed the documentation effort and things got bogged down in the area of capturing our process decisions and experience.

To summarize, some of the key things I learned include:

- People need to be involved, and that includes management. Buy-in is an emotional as well as intellectual commitment, and is a personal thing. People need individual time to discuss the ideas and make that commitment.
- You need to really understand the philosophy and assumptions behind the improvement program you select, and communicate that to everyone.
- The temptation to over-engineer things is great; the impact of over-prescribing things on people can be quite negative. It may be better to provide a general framework along with the tools for process analysis and improvement.
- There is great temptation to take on too much too fast – your learning and understanding will outpace your ability to put things into practice, and that leads to frustration. There may also be business pressures to try to move too quickly.
- Documenting processes in a way that captures experience and allows for continuous improvement is non-trivial. The system needs to be well designed and managed.
- It is hard to stay committed to your chosen path; the tendency is to give up at the first sign of trouble and try something else, yet results take time.
- The quality improvement program tends to become a lightning rod for every area of discontent in the organization. There is a need to constantly communicate the limits and scope of the program, to avoid unrealistic expectations.
- People's expectations can be well beyond your ability to deliver results, especially in the short term (e.g. that software quality and productivity will dramatically improve, that hassles will go away, that management will start making decisions the way developers see things, that

developers will start seeing things the way management does). Managing expectations is a major component of the process.

- There is significant danger that the change leaders (those driving the improvement efforts) will suffer burn-out.
- The frustration level can be acute, especially when you (perhaps subconsciously) buy into the expectation of a silver bullet.
- It is very easy to overlook your progress because you are focused way ahead on perfection.
- Things go well if you manage to find some leverage points – simple changes that have a big impact in your environment. In our case, the adoption of a peer review process gave everyone first-hand experience with a defined process for improving quality – a process that has positive measurable results.

In general, we had much less difficulty defining processes than we had institutionalizing them and getting people to embrace them. For me, the challenge became how to manage the buy-in – and if we could find a way to do that, the specifics of good practice would be relatively easy to set in place. Perhaps the following three questions generally capture the challenges and learnings cited above:

1) How do you design these programs to promote personal buy-in?
2) What happens when the thrust of the improvement initiative is out of sync with the organization's culture?
3) How do you manage the expectations, including your own?

You will notice that these questions do not have anything to do with the technical aspects of determining best practice in terms of specific methodologies. They speak to the heart of how people work, how they are motivated and rewarded in their work, and how the organization's culture changes (or resists change).

Theoretical Frameworks for Understanding Change

With this base of experience and with these questions, I searched for answers (or at least understanding) in my courses at the University of Calgary MBA program. I'd like to look at some of the academic theory I found useful in helping me understand my situation. In particular, I'll look at personal motivation, organizational culture, and cultural and personal change, in terms of the questions posed in the last section. I will also address how theory in these areas may affect the selection of an approach to quality improvement.

1) How do you design these programs to promote personal buy-in?

I begin this discussion of quality improvement as a change process at the level of personal motivation because any change is fundamentally a personal thing. "Quality has to come from within, pride of workmanship and a quest for continuous improvement regardless of the standards."[6] If the individuals involved in the change process are not motivated to change, the change is at best superficial and likely to be short-lived.

For me, perhaps the most powerful view on personal motivation is the concept of flow, as described by Csikszentmihalyi[7]. Flow is a state of total unselfconscious concentration, a state of involved enjoyment. This is the state we experience when we are so involved in something that time seems to go away. Csikszentmihalyi describes this as the state most rewarding to human endeavor – of optimal experience. The implication for work is this: the extent to which work promotes flow, the more rewarding and fruitful the work is.

Elements of Optimal Experience (FLOW) [8]

- done for intrinsic reasons
- goal directed
- receive feedback as to performance
- challenging activity well matched to skills
- focused concentration
- sense of control over actions
- altered sense of duration of time

The key conditions and characteristics of flow are listed in the table above. These characteristics suggest implications for the implementation of quality improvement.

---

[6]Aguayo, R. quoted in Howes, C. *Ahh, Perfect.* Calgary Herald, Oct. 17, 1992.

[7]Csikszentmihalyi, M. Flow: The Psychology of Optimal Experience. Toronto: Harper and Row, 1991.

[8]Summarized from points in Csikszentmihalyi, M. Flow: The Psychology of Optimal Experience. Toronto: Harper and Row, 1991.

In particular, there should be intrinsic reasons for the change, the change should be goal directed, with clear feedback. The change should be challenging but not be impossible to attain, and should make good use of people's skills. The results will support a sense of control and involvement.

This seems fairly intuitive, yet the way in which processes are defined and programs are implemented in the effort to achieve quality improvements and certification to standards can easily transgress these fundamental principles. In my own experience, one computer scientist spoke to me candidly after he had complied with a defined process that was supposed to represent an improvement. He said:

> I can do things according to the defined process, but it takes me longer and it takes absolutely all the fun out of it for me. When I can work the way I naturally approach problems, I get into a state of flow and become totally involved. Now that I have to follow the defined process, the whole thing loses all interest for me and becomes tedious.

If the improvement programs and documented procedures are not consistent with the fundamental ways in which human genius works, we will be locked into a cycle of ever-increasing mediocrity and bureaucracy, rather than responsive quality improvement.

Another view of motivation and commitment is presented by Peter Senge in The Fifth Discipline, The Art and Practice of the Learning Organization. He talks about personal mastery and genuine commitment as being fundamental to satisfaction in work life. Personal mastery involves defining what is really important, seeing current reality and then being true to that vision. It is often through work that the full range of human creativity and expression is allowed to flourish.

One of the ways this human potential is harnessed or focused in an organization is through a *shared vision.* The idea is that when an organization becomes excited by a commonly held vision of what it can be, great things can happen. Senge talks about the difference between commitment, which is the full dedication that people can bring to their work, versus various levels of compliance. Organizations need commitment because, for one thing, people *want* to be involved in their work at that level. That is what they are looking for - real meaningful, higher-purpose work, not just putting in time. Secondly, it is the engine by which innovation and high quality and excellence come into being. Starting with a powerful vision is the first step.

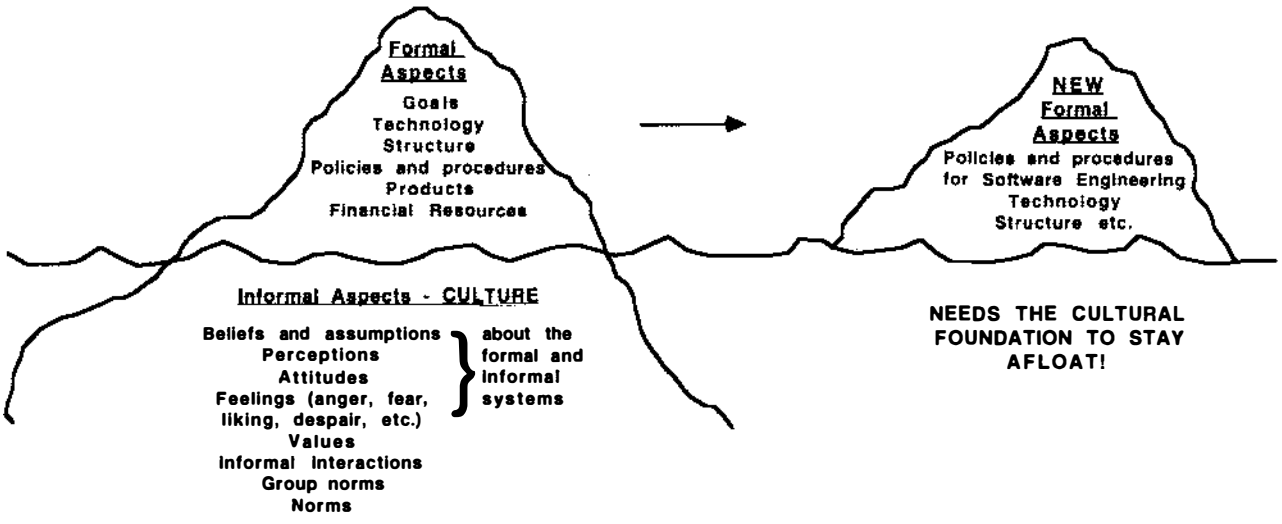
The implications of Flow and Senge's concepts of personal mastery, commitment and vision are the following: quality improvement programs that tap into the fundamentals of human motivation and commitment can be extremely powerful. Programs that do NOT tap into these fundamentals can become nothing more than surface level, short-lived facades, a waste of human and organizational potential. In my experience at ACTC there is among the employees a genuine longing for a

higher-level purpose – a desire to commit to something really important. This can become a real engine driving a quality program – yet it can quickly turn to distrust and resentment if the program starts to look like a game. Any program that looks like it is designed to satisfy standards for their own sake runs a risk of falling into this category.

2) What happens when the thrust of the improvement initiative is out of sync with the organization's culture?

Organizational Culture and Cultural Change

Another aspect that needs to be taken into account in any major change effort is that of organizational culture. The following diagram illustrates the concept of organizational culture as a foundation underlying all the explicit policies and practices of the organization[9]. Any attempts to change the policies and practices (such as through a quality improvement program) need to be supported by the underlying culture. If they are not, then the initiative will require a cultural change as well (as is often the case).



Organizational Iceberg

Source: French and Bell
Organization Development

Successful change depends on positive norms such as trust, understanding, and openness – the more these are the cultural norms the easier it will be. The term *culture gaps* refers to the gap between the norms that are actually in force and those that should be.

> The larger the gap, the greater the likelihood that the current norms are hindering both morale and performance. If the assessed culture

---

9 French, W. and Bell, C. Organization Development, Prentice Hall Inc., 1990.

gaps are allowed to continue, work groups are likely to resist any attempt at change and improvement. Specifically, *culture gaps materialize as an unwillingness to adopt new work methods and innovations, as a lack of support for programs to improve quality and productivity*, as lip service when changes in strategic directions are announced and, in the extreme, as efforts to maintain the status quo at all costs.[10](emphasis added)

Several models of organizational orientation also illustrate the problem of trying to change the overt practices of the organization if they are out of sync with the underlying culture. Waterhouse[11] discusses two types of organizations, sheltered and exposed, and distinguishes them on a number of characteristics (see diagram below). The sheltered organization tries to continually protect itself from the environment, whereas the exposed organization is open to challenges and change. The focus on certification and standards is characteristic of a sheltered industry orientation; firms within that industry are engaged in formulating formal policies, procedures, and controls. The opposite is true of exposed industries and organizations whose focus is on vision, future directions, strategic guidance, customer focus and empowerment. This seems to present opposing demands on a company that needs to operate in an open environment, yet become certified to defined standards. The challenge is to manage the need to meet standards in a way that enhances the open responsiveness of the firm.

GENERAL CHARACTERISTICS

ORGANIZATIONAL LEVEL	SHELTERED <——— TYPE OF ENVIRONMENT ———> EXPOSED	
SOCIETAL (GLOBAL, NATIONAL)	LAW & ORDER	ADAPTIVE, LEARNING
INDUSTRY (SECTOR)	STANDARDS QUALIFICATION, CERTIFICATION	VISION FUTURE DIRECTION
CORPORATION (FIRM, ORGANIZATION)	FORMAL POLICIES	STRATEGIC MGT. GUIDANCE
SUB-UNIT (DEPARTMENT)	FORMAL PROCEDURES CONTROLS, RULES	CUSTOMER FOCUS EMPOWERMENT

This conundrum is also illustrated in Towers Perrin's model of cultural stages and the need for consistency between all dimensions of culture for each stage (see diagram below)[12]. For companies who are used to working in the participative, self-directed style (of stage IV), who now need to define processes such as would be

[10]Kilman, R. *Corporate Culture,* The Corporate Culture Source Book by Bellingham, Cohen, Edwards and Allen, Human Resource Development Press, 1990, pp. 17-22.

[11]Waterhouse, M. *Developing and Managing Radical Change.* Faculty of Management, University of Calgary, February 1994.

[12]Model by David A. Lough, Principal, Towers Perrin.

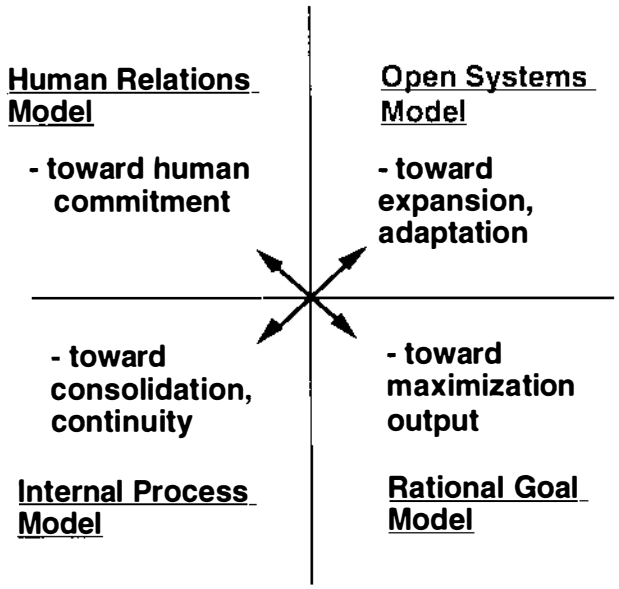
characterized in stages I and II, there may be some cultural resistance that needs to be managed. On the other hand, the model also implies the need for a foundation in basic quality processes before more advanced levels can be achieved.

Structure for Culture

stage	I	II	III	IV	V
leadership style	hierarchical	consultative	facilitative	participative	situational
formal organizational emphasis	individual roles	natural work teams	cross functional work teams	self-directed teams	clusters
what assets are valued in company		physical plant, inventory	+intellectual capital, diversity	ideas, information, creativity, vision	
employee line of sight	department	business inside a business	value centre	portfolio	enterprise
quality focus	inspection	process reliability	process efficiency	customer responsiveness	customer appreciation
quality goal	do things right	do things right the first time	minimize cycle time	do the right thing	acquire competitive advantage
values		profit, growth control	creating value, trust, learning	service responsibility personal fulfillment	

The Quinn model of Competing Values[13] may serve to further illustrate this situation (see diagram below). The upper right hand corner is the *open systems* model, which is in diametric opposition to the *internal process* model of the lower left. The key is to achieve balance – not to be so open as to be chaotic, yet to avoid over-defining things so as to become solidified and unresponsive. Finding the balance is the difficult part, and feeling comfortable with the balance may require a change in culture. Cultural changes may break implicit *psychological contracts* between employees and the organization, and thus may be resisted.

[13]Quinn, R. *Mastering Competing Values: An Integrated Approach to Management*, reprinted in The Organizational Behavior Reader, Prentice Hall, 1991, p 34.



Competing Values Framework: Effectiveness
Robert E. Quinn: Mastering Competing Values: An Integrated Approach to Management

In describing various cultural orientations in the workplace, a consultant for Wallace Bond and Partners observes the following, which applies to companies such as software development firms:

> The Task Culture concerns itself with solving problems... This leads to attracting people who are skilled, experienced and expert in their craft or technology... [these] people need new challenges to thrive. Therefore *this culture can come to an end abruptly as the need to experiment and achieve specific higher goals is overtaken by the requirement to stabilize and routinize,* i.e., become a Role Culture... Technical experts and scientists will ultimately function poorly in a work environment which requires them to toe the line to order and system.[14]

This is a caution against over-zealously defining processes to the point of driving out the very innovation that the firm needs in order to have a competitive advantage. The dangers of over routinizing processes is further discussed by McLagan, in her article *The Dark Side of Quality*

> It costs money to develop and follow strict procedures for work processes. When the benefits outweighs the costs, routinized processes may be desirable. Processes should not always be proceduralized, however. Why not? First, proceduralized processes are in great danger of becoming ends in themselves... The intended impact of waking

[14]Toombs, G. *Understanding the Organization - Influence of Culture.* Viewpoint, Toombs Wallace Bond Inc. (not dated).

> people up and stimulating innovation is lost. And without innovation, there is no such thing as continuous improvement. Second,...much work today requires complex thinking and involves many exceptions to the rules... A knowledgeable, goal-oriented person who believes in the key values of the organization can find many ways to work more efficiently... The crucial thing is to treat processes as means, not ends.[15]

Process as an end in itself is stifling bureaucracy, certainly not what software companies in today's fast-moving competitive environment need! In terms of the over-definition of process, the open-ended approach of TQM may be more appropriate, in that it recognizes the need to start where people are and make use of their intrinsic desire for quality.

> One of the human needs that TQM recognizes is the need to create. Quality is a form of perfection that has intrinsic value; a quality product is a work of art in the sense that it embodies the human quest for perfection.[16]

> ...the megaprocesses associated with quality [statistical process control, systematic problem analysis, group problem solving] have such *tremendous potential* for raising levels of thinking and performance[17](emphasis added)

Organizational Development Theory

Organizational Development (OD) theory classifies change programs aimed at quality improvement as *Structural Interventions*. These are programs that are aimed at

> improving organization effectiveness through changes in the task, structural, and technological subsystems...[and] includes changes in how the overall work of the organization is divided into units, who reports to whom, methods of control...work flow and procedures, and role definitions.[18]

Organizational development as a discipline involves a diagnosis / data gathering / action planning sequence, carried out by teams in the organization. The approach is bottom-up, in that problem definition and scope as set out by the individuals in the system is the starting point for activity. For this reason,

---

[15]McLagan, P. *The Dark Side of Quality*. Training, November 1991, p. 32.

[16]Grant, R. et.al. *TQM's Challenge to Management Theory and Practice*. Sloan Managment Review, Winter 1994, p. 31.

[17]McLagan, P. *The Dark Side of Quality*. Training, November 1991, p. 33.

[18]French, W. and Bell, C. Organization Development, Prentice Hall Inc., 1990, p. 180.

> most programs targeted from the outset at structural change are not OD... Interventions such as job enrichment, MBO, the formation of work teams congruent with a particular technology, and changes in work rules are often applied without much diagnosis and planning and are often "installed" without participation of the relevant work groups and/or job incumbents. [19]

The point here is that when programs are "installed" without workforce involvement in the change process, they may be less effective. The more the quality improvement program follows the principles of sound organizational development programs, the greater the chance for acceptance and buy-in.

What does all this mean in my own situation? I would characterize ACTC as in transition from a relatively sheltered environment to one which is very exposed. As to organizational orientation, I would say we are in the process of moving from the Open Systems quadrant in Quinn's model into the Internal Process quadrant. What this means is that we are undergoing significant cultural change – trying to institute definition into processes that used to be up to individual initiative and discretion. Within the organization I see different rates of change in different areas – so we are in a somewhat inconsistent state. What this adds up to is a situation in which the dynamics of the cultural change itself require significant attention.

How do you effect cultural change?
Cultural change is a difficult thing to accomplish. One of the concepts that is useful in planning for cultural change is the concept of *cognitive dissonance.* The concept is that the mental models we carry around prevent us from interpreting things in new ways, and tend, therefore, to preserve the status quo.

> Cognitive dissonance blocks change: Mental processes filter or reject information that does not match the paradigm... Repeated stimuli in several forms help swamp the paradigm.[20]

In order to overcome cognitive dissonance, a multitude of stimuli are needed. To cause change, messages from a number of sources and contexts, all in alignment with the new concept, are needed. If there are too few messages, or if they are not in alignment, then the existing environment will overcome the new initiative. The old mindset will prevail and people will reinterpret information based on old mental models. This was our idea when we instituted things like the Process Improvement Newsletter – to bombard people with the new way of thinking. We also held company-wide meetings and made presentations. The trouble with these approaches is that they are relatively poor ways of communicating with people.

---

[19]French, W. and Bell, C. Organization Development, Prentice Hall Inc., 1990, p. 180.

[20]Sharpe, K. *What is Corporate Culture and How do We Change It?* Presentation to University of Calgary Policy and Environment 789.10, February 23, 1994.

More direct one-on-one discussion is needed to overcome the cognitive dissonance and build commitment to change.

Another hindrance to cultural change can come when people protect their territory, sometimes to the detriment of the needed change. One idea that works is to get the group to make a mental leap into the future and build a shared vision of what they want to see. Then the current precious territory doesn't seem so valuable[21].

3) How do you manage the expectations, including your own?

This question was critical to my own feeling of satisfaction (or lack of it) in our progress. It was helpful for me to hear Rosabeth Moss Kanter, in a videotape interview on change in organizations, explaining that "everything looks like a failure in the middle." When you've put lots of investment in and have received no returns out yet, it can get very discouraging. Another gem from Kanter is that people won't say NO to an initiative in the beginning – they wait until the vulnerable period in the middle, when it looks like the change is going to become reality and affect them. It is in the middle when the most resistance and the greatest doubts occur. Knowing this is a common occurrence can help the change leaders stay the course.

Another concept I found useful is Senge's description of *the wall.*[22] This refers to the difference between *convergent* and *divergent* problems. There are some problems that are very complex, and they don't have a single "cause" – in fact they have a number of interrelated causes and effects that feed on each other. They do not converge, but rather *diverge* – the harder you look at them the more complex they become. Much of what we deal with in organizations does not have an easy answer; knowing this helps to set the expectation level, for yourself and for others (it also helps to know that management does not know the answer either!). Understanding this helps to maintain a positive creative tension between your current reality and your vision; if you believe that you should be able to bridge this gap easily, then it is easy to become discouraged and have the positive tension turn negative.

Finally, there is no substitute for communication in many ways and with many people to build a common understanding and realistic expectations for the quality program.

---

[21]Waterhouse, M. *Developing and Managing Radical Change.* Lecture, Faculty of Management, University of Calgary, February, 1994.

[22]Senge, P. The Fifth Discipline.

Conclusions and Recommendations

The words of a Fortune 500 CEO sum up my conclusions in six concise points:

Six hard lessons about Quality:	
1)	Quality is a matter of survival.
2)	Total quality requires a cultural change, and I believe that you cannot make a cultural change without an emotional experience.
3)	It's going to take time.
4)	You've really got to have top management pay attention.
5)	You have to have a scoreboard...
6)	Everybody *–everybody–* has to be involved.[23]

Although my analysis does not provide any conclusive results, I hope it gives some food for thought in terms of how quality improvement programs should be implemented, and perhaps more importantly, in terms of what we can expect in the way of difficulties. Quality improvement programs, (standards-driven or open-ended), are not trivial undertakings, and they run into all the barriers and challenges of personal and organizational change, yet they can succeed. Several themes emerge from this discussion.

First, management needs to understand the full impact of what they are taking on; this is not a quick 'fix' that will look good in the marketing literature. People will be very skeptical of anything that looks like gamesmanship; programs that are represented to be authentic quality improvement programs but really have political motives will be quickly dismissed. Remember the adage "they watch your feet, not your lips."

Second, the culture and personal motivation of the people working in it will need to be taken into account in designing the program. Blow-by-blow process definitions may be appropriate for McDonald's restaurants, but will likely be totally inappropriate for software R&D. This obvious fact can be easily overlooked when confronted with the exhaustive documentation requirements for ISO 9000 certification. Keep the documentation at as general a level as possible for the situation you are in. If your situation does not demand adherence to standards, the TQM approach to quality may be more appropriate.

Third, keep in mind that any cultural change is a lengthy process and, as Rosabeth Moss Kanter points out, they "all look like failures in the middle." Your key change

---

[23]Marous, J. Chairman of Westinghouse, quoted in Main, J. *How to Win the Baldridge Award.* Fortune, April 23, 1990. p. 101.

agents will likely need some outside support in order to keep on track and not become discouraged. Understanding of the change process can help keep things in perspective.

In spite of these cautions, instituting a quality program is a requirement these days, as well as the right thing to do for intrinsic reasons. The key is to lead and manage the program so that the results are realized:

> Companies that survive in the 21st century will be those that know how to use leadership skills to continually improve quality management systems, environmental quality systems, and overall organizational process so that they can truly be considered world-class companies.[24]

In summary, standards and models such as the SEI CMM are useful as a start to quality improvement, but they should not be pursued blindly or without a clear view of customer needs. Insofar as possible, open-ended methods should be used within the framework of the standard. Whether the quality improvement program is standards-driven or open-ended, it will likely involve significant cultural change. There is much to be considered in attempting any significant organizational change program.

---

[24]Marash, S. *The Key to TQM and World-Class Competitiveness Part II.* Quality, October 1993, p. 46.

A Process for Measuring Software Consulting Quality

Douglas Hoffman

Software Quality Methods
24646 Heather Heights Place
Saratoga, CA 95070
Phone/Fax 408/741-4830
hoffman@netcom.com

Abstract

This paper describes an innovative, copyrighted process implemented to improve quality in situations where software project work is done by consultants hired through an agency. The program was conceived by George Birdsong, III, founder of Systems Partners[1]. The process has been implemented over the last three years and is used by clients and consultants of Systems Partners. This paper provides a brief history of the quality program, the main elements and mechanisms in the process, and a description of some of the results obtained.

Keywords and Phrases: measurement of quality, critical success factors, software quality in consulting projects

Biographical: Douglas Hoffman is an independent consultant with Software Quality Methods and Adjunct Instructor with the University of San Francisco. He has been in the software engineering and quality assurance fields for over 21 years and now specializes in identifying the appropriate development processes and tools for software quality based upon specific organizational requirements. Currently, he is Chairman of the Santa Clara Valley Software Quality Association (SSQA), a Task Group of the American Society for Quality Control (ASQC), and Program Chairman for the First International Software Quality Congress scheduled for May 1995 in San Francisco. He was the Program Chairman for the Third International Conference for Software Quality in October 1993, and speaker on software quality at numerous conferences over the past twenty-two years. He is also active in the local section of the ASQC and the ISO 9000 Task Group. He received his MBA from Santa Clara University, and his MS in Electrical Engineering and BA in Computer Science from UC Santa Barbara.

---

[1] Systems Partners, a member of the National Association of Computer Consultant Businesses (NACCB), is headquartered at Two Orinda Theatre Square, Suite 315, Orinda, California, 94563. Founded in 1990, they have placed hundreds of data processing and software engineering consultants at an impressive list of Fortune 500 clients. Systems Partners developed this copyrighted program to measure and improve the quality of their contracting services

Background

Much has been done and written about measurement of software quality over the last twenty-five years. Recently there has been an emphasis on defining and managing the process of creating the software and the design, measurement, and improvement of the development process. These measures focus on the product and the methods for its creation, leaving out the human element that is fundamental to all projects. Interpersonal communication is paramount in a consulting relationship, and it is this human element that is the focal point for the measurement process for tracking the quality of work. This interesting departure from the historic approach has substantial advantages and is effective in maintaining excellence in client-consultant relationships and the end products.

The historic avoidance of measurement of the human factor may be due to the existence of separate personnel policies governing the measurement and feedback of employee performance in most companies. The implementation and effectiveness of personnel policies vary from company to company, and even department to department within most large companies. Consultant services are becoming widely used in the software arena, yet company policies and procedures are almost never applied to consultants. Although software development is really a human undertaking, it has been argued that assessment of personal performance is best left to human resources, and not included in the realm of quality assurance at all. Therefore, measuring performance has been avoided by quality groups, and the work and results are often not measured at all. This especially applies in instances where the person performing the work is a consultant or contractor.

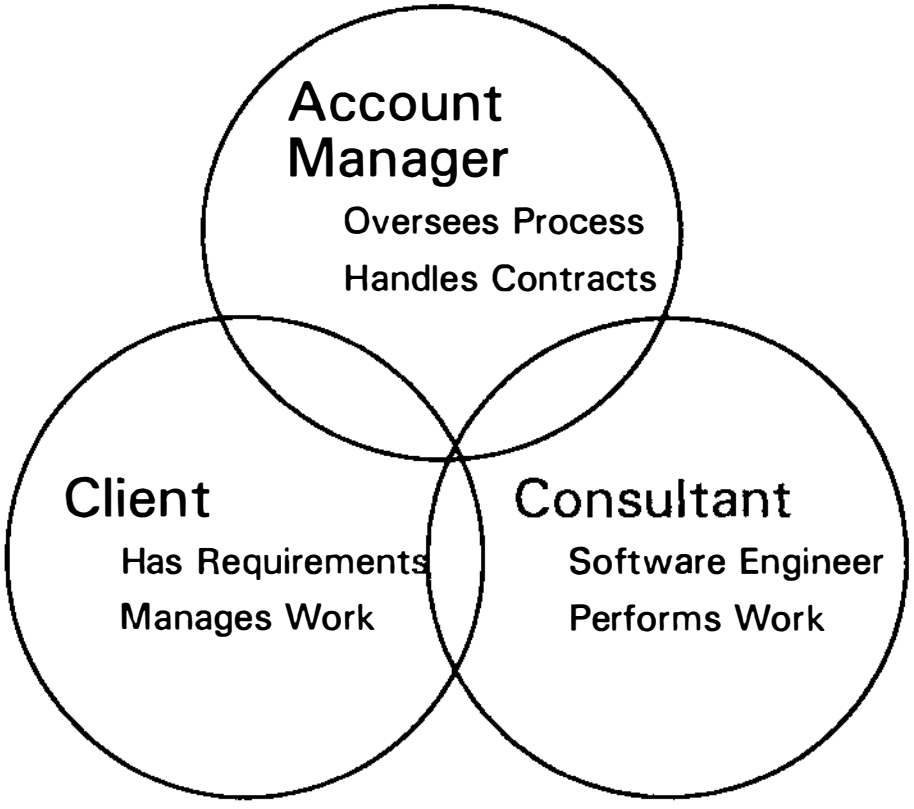
The desired benefits from the System Partner's quality program are: 1) increase the software consultant's effectiveness, 2) improve communication, 3) identify priorities, 4) reduce ambiguity, and 5) monitor consultant performance on project specific goals. The program also provides a framework for ongoing improvement in future contracts, both for the client and the consultant. The program has five main goals. First, project success is paramount to assure that the client ultimately gets what they need. Second, is to describe what good quality will look like to the client. Third, is to actually measure the quality provided. Fourth, the process records the history of the relationships with the client and the consultant. The fifth goal is to provide a mechanism to reward good work by the consultants, as measured by the program.

Initial planning for the quality program included brainstorming sessions with clients and consultants experienced in the staffing and managing of software quality assurance contracts. The challenge was to create a program that could measure quality results in consulting without stifling innovation. Some of the participants were skeptical and thought measurement of consultants was unnecessary, inconsistent, or even detrimental. Others thought it was an excellent idea, but that setting and measuring consultant performance was beyond the state of the art in software quality assurance.

Five major companies in the San Francisco Bay Area were selected for the initial development of the procedures, and a Beta test was run with a Fortune 10 company. Although the program was extremely successful during testing, Systems Partners felt that additional outside expertise would facilitate fine tuning and polishing the processes. They brought in a software quality

consultant to formalize and document the procedures. The consultant's job was to quantify the performance measures and define processes that would be accepted by the participants. The consultants who are central to the process and benefit from it are also very sensitive about measures and metrics. It was critical to create non-threatening methods and measures that would be effective. The end result was a program with simple procedures that quantitatively measure and monitor the quality of services provided. There are no hidden judgments or negative consequences.

The program itself focuses on two key individuals: the client and the consultant. The procedures are simple so they can be used by clients and consultants without training or further preparation. The focus is to identify priorities and expectations, then measure progress based on the identified criteria. Continuous communication and feedback to all parties closes the loops and monitors the quality of services provided. Although the program has been designed and implemented specifically for consultants working through third party managers, it could easily be adapted for use by consultants themselves (without the involvement of third party managers) or with in-house consultants. Some of the benefits of oversight by a third party would be lost, but the basic communication of requirements and feedback of performance would be valuable.



Relationship Between Participants

How The Quality Program Works

The quality program goes through three main phases for each consultant/client relationship: initial planning, project administration, and final reviews. The quality measures are incorporated into a customized time sheet for each consultant/client relationship. The Account Manager is responsible for the overall program, and initiates and oversees each phase.

During the planning phase, information is documented as part of the contract itself and the quality program is used as the mechanism for setting the project expectations. In this phase the focus is on identifying the deliverable work products and important factors that the client desires. The account manager and client work together to identify the major success factors for the job. The is held with the focus on the forms, so the clients uniformly specify the deliverables and communicate the success factors through the checklists. Although the process only takes 30 to 60 minutes to explain and work through, it provides the critical foundation for success of the project. Understanding and articulating the requirements, priorities, and expectations is key to success in any working relationship. The granularity of the tasks and outcomes is fine enough that interim milestones are often identified.

Results from the initial planning include a list of important elements to be monitored and measured, and methods for measuring and identifying good quality for each. The client is asked to think of three to five key items that are critical to the success of this particular project. The success factors range from "Completes Work on Schedule" to "Communicates Effectively" to "Presents an Appropriately Professional Image". A checklist is provided with two dozen suggested items in eight categories, but clients are encouraged to create any other items that are important to the specific project. Clients actually identify between three and a dozen success criteria. By focusing on only a few elements, the really critical success factors can be identified and articulated. Even subjective items are encouraged, as the client is asked to identify how each will be measured. This allows clear communication of expectations and priorities dealing with how the jobs are to be done.

Once the client and account manager have identified the deliverable work products and critical success factors, the consultant can be selected and informed. Because the deliverable work products are well defined, the right person can be more easily identified to do the job. The account manager reviews the information with the consultant and gains agreement on the details of the task and success items before project work begins. This clear communication of expectations and deliverables results in a very high degree of focus and success in the work with few surprises.

Once the work is underway, the program is administered by the account manager and monitored on a continual basis. The time sheet is customized for each consultant and project to list the success factors and provide space for the client to rate the performance during the accounting period (usually twice a month). When the client fills out the time sheet for the consultant each period, a rating is given on each of the identified measures. Both the consultant and client are presented with graphs of the ratings on the project so they can evaluate how well the project is progressing. Since the process is automated and built on top of already existing

contacts and procedures, very little additional effort is needed to gain the advantages from measuring the critical success factors.

The rating is done on a simple numeric scale of 1 to 4. A score of 4 is **Excellent**, 3 is **Above Average**, 2 is **Satisfactory**, and 1 means **Needs Improvement**. Since needs change throughout projects, **N/A** is used to indicate when the criteria is Not Applicable. This later rating is also one of the triggers for the account manager to contact the client to evaluate if the criteria should be removed or replaced.

The time sheet information is processed for both the billing and the quality ratings. This provides timely feedback on performance, and communicates any exceptions to the parties involved. When ratings drop too low, the account manager takes immediate action to ensure that all parties are aware of the situation. When corrective actions are required, very little time has been allowed to pass so the effort to get back on track is minimal. This also identifies problems early so contingencies can be put in place when required.

The final phase is triggered upon completion of the project. All three parties involved with the contract fill out evaluations of the project results, measuring the quality of the relationships as well as the delivered products. This information is combined with the quality measures taken throughout the project to produce a final summary "report card" that is presented to the client and consultant, and becomes a permanent record kept by Systems Partners to evaluate the quality of the work performed and to improve the chance of success for future contracts.

Benefits From The Quality Program

Since the program started in early 1993, hundreds of people have participated and benefited by using it. By using the program as a framework for open communication, the objectives and priorities for each project have been clarified. The account managers have been able to consistently identify the client needs and match consultants' abilities to best satisfy them. This has both simplified the job and increased the success in matching consultants to tasks. It has made clients think seriously about the work products they desire from the consultant and the critical success factors in accomplishing the tasks. It has also made consultants aware of what's important to the clients so they can concentrate on doing the right things in the expected manner. Timely feedback on performance has allowed early detection and correction of problems, so Systems Partners has effectively eliminated major difficulties in the consulting relationships.

Clients have benefited by getting well qualified and knowledgeable consultants for the projects. They are also assured of having consultants who understand the job requirements and are able to focus quickly on the creation of deliverable products. The consultants understand the milestones and are more effective, thus saving time and money for the client. The administration of the time sheets and quality elements is simple for the clients, and the addition of the quality program does not significantly increase administrative time requirements.

The consultants have also benefited. Their understanding of precise expectations and work products has made them more effective and efficient in their work. Consultants, like employees, like to know how they are doing on a task. The quantitative, timely, continuing feedback lets the consultants know how they are perceived by the client. They get special recognition for the

achievement of goals and participate in a bonus program as well. Rather than using purely financial incentives, Systems Partners' bonuses include specialized courses for consultants to improve their managerial and technical skills. These classes are in particular technical and managerial subjects of interest to the consultants.

A major benefit has come from the increased clarity of assignments. The clients and consultants have both benefited tremendously from the program in this way. This has cut rework and wasted efforts, and improved the quality of the end products. The amount of time spent in understanding the client requirements has not noticeably increased, but the quality and consistency of the communication has improved markedly.

Some Lessons From The Quality Program

To see how well the program was working and examine some of the assumptions, a sample of 26 consultant's ratings over a period of months was analyzed. The information was tabularized and some operational and statistical questions were answered. Some of the lessons learned and discoveries made are presented here.

Although the clients were asked to identify three to five criteria, many of them selected more than that. One thing the data showed was that the number of criteria selected and tracked makes a difference in how well they are used. The ratings were consistently recorded when 7 items or fewer were tracked. In each situation where more than 7 criteria were identified, not all of them were tracked. This seems to indicate that a maximum of seven criteria can be reasonably rated at one time.

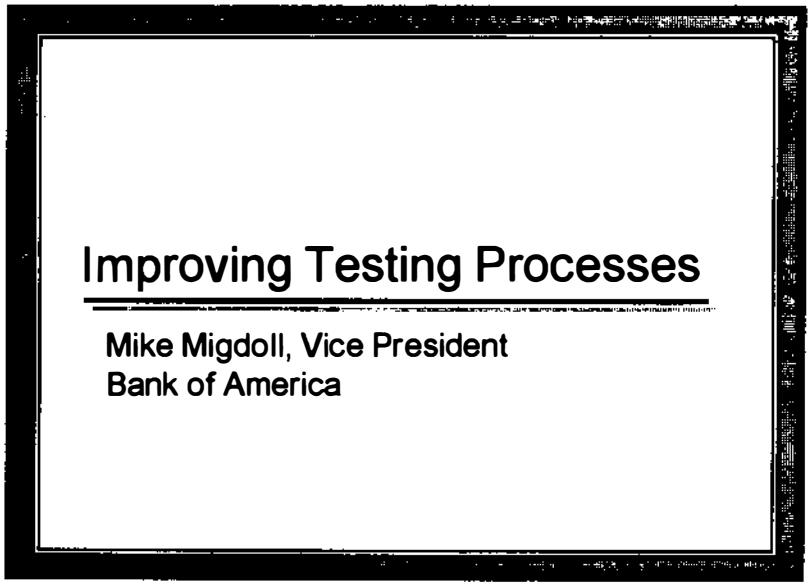
A close evaluation of the ratings showed one consultant with consistently declining scores. The management process was working, and corrective action was being taken as the sample period came to an end. Other declines in scores were from Excellent to Above Average — still an acceptable level of performance. The Needs Improvement rating was never selected to indicate performance, showing that all consultant performance was at least Satisfactory.

The two cases to date where there has been a significant decline in a consultant's scores have demonstrated the positive power of quick feedback. In one case the consultant was encountering personal problems outside of work that were worked out and performance returned to normal levels. In the other case, early detection of possible problems allowed easy rescheduling and reallocation of resources to overcome the difficulties. Other instances of **Satisfactory** scores were temporary situations, or cases where the manager used the rating to really mean that the performance was perfectly acceptable.

Further evaluation of the data rating the consultants shows that performance generally improved over the sample period. Of the 138 items rated, 33 showed improvement, 91 were unchanged, and 14 declined. The ratings were more than twice as likely to go up as down, and more than six times as likely to be unchanged. The data has a X^2 (Chi Squared) value of 70, with 2 degrees of freedom, leading to the conclusion that the improved ratings are not due to chance (with a 0.1% confidence).

We also found that 38% of the consultants showed improvement in their ratings during the sampled periods, 35% were unchanged, 19% showed declines in scores, and 8% showed mixed improvements and declines. Twice as many consultants showed improvements in scores over declines. The data has a X^2 value of 23.76, with 3 degrees of freedom, leading again to the conclusion that the improved ratings are not due to chance (with a 0.1% confidence).

In summary, clients are taking advantage of an innovative program to expose and measure success criteria in software quality consulting relationships. This program is straightforward automated, and integrated into normal business relationships and administrative work needed for a successful client/consultant relationship.



Overview

- Background
- Problem Analysis
- Strategy for Change
- Implementing Change
- Accomplishments
- Lessons Learned
- Recommendations

Background

Quality software, but . . .

- More complex systems
- Rising testing costs
- Testing on critical path
- Detection versus prevention

Slide 3

Problem Analysis

Testing Task Force

- Broad-based participation
- Time and cost analysis
- Structured interviews
- Recommendations
- Consensus building
- Management commitment

Slide 4

Strategy for Change

Testing Improvement Project (TIP)

- **Implement Testing Task Force recommendations:**
  - Develop testing methodology
  - Enhance testing training
  - Increase use of testing tools
  - Automate testware management
- **Build on existing culture**
  - Testing process
  - Testing deliverables

Slide 5

Implementing Change

- **Internal Steering Committee**
  - Draft policies and procedures
  - Distribute for review and approval
  - Present recommendations to management
- **External Consultants**
  - Conduct *Testing Practices Assessment*
  - Collaborate on training improvements
  - Recommend additional strategies
  - Provide external perspective
- **Pilot Projects**

Slide 6

Accomplishments

- **New Methodology**
  - Broadening acceptance
  - Emphasis on prevention
- **Enhanced Training**
  - Half-Day Seminar
  - System/Acceptancetesting
- **Increased use of testing tools**
  - On-linetool catalog
  - Automated facility
- **Automated Testware Management**
  - Repository development

Slide 7

Lessons Learned

- **Have patience**
  - Consensus building
  - Enculturation is a slow processs
  - Evolution versus Revolution
  - Tempered zeal
  - Education is crucial
- **Emphasize spirit over letter**
- **Build credibility**
  - Recruit a competent staff
  - Deliver tangible products
  - Keep selling
- **Recognize metrics as a challenge**

Slide 8

Recommendations

- Create consensus
- Set clear objectives
- Ensure management support
- Build on current culture
- Sell and educate at *all* levels
- Be patient and flexible
- Get external perspective

Slide 9

Dynamic Quality Management: Achieving and Maintaining the Software's Fitness-For-Use

Hubert F. Hofmann and Johannes Geiger
Department of Computer Science, University of Zurich
Winterthurerstr. 190, CH-8057 Zurich, Switzerland
phone: +41-1-257-4580
email: {hofmann, geiger}@ifi.unizh.ch

Abstract

Dynamic Quality Management (DQM) is proposed as the new way for achieving and maintaining the software's fitness-for-use in a continuously changing business environment. We discuss DQM's essential elements: action programs, quality principles and measurement schemata. We also outline their integration and we provide preliminary results of DQM in action.

Keywords and Phrases: software management, quality management, quality improvement, software maintenance.

Biographical Data: Hubert F. Hofmann is research assistant at the University of Zurich. He is a member of the Information Engineering Group of the Department of Computer Science and project leader at the Union Bank of Switzerland (Quality Assurance/Management in Great Maintenance Projects). His major research areas are requirements engineering, "situated design", quality management, software-development environments, and logic programming. He received a MS in business and computer science from the Johannes Kepler University Linz.

Johannes Geiger is a member of the Quality Assurance/Management Group of the Union Bank of Switzerland and a research assistant at the Department of Computer Science of the University of Zurich. He worked several years as an independent consultant in Austria. His research interests include quality management, software engineering, organizational development and learning. He received a MS in business and computer science from the Johannes Kepler University Linz.

1 Introduction

Quality management comprises principles, actions, and measures necessary to achieve and maintain a defined level of quality [19]. The crucial point in quality management of software systems is (1) to find an appropriate integration and balance among the different perspectives and requirements of the people involved in a quality project; and (2) to translate them into an organizational practice that will satisfy the desired level of quality [15]. When dealing with application programs, also called business software, we have to consider the technical implementation and its organizational environment. Both dimensions are continuously changing, largely unpredictable, and only partially knowable [38]. Therefore, quality management has to preserve the software's fit with the environment dynamically.

In the next chapter, we will outline our understanding of software's fitness-for-use. We will then propose Dynamic Quality Management (DQM) to achieve and maintain the software's fitness-for-use. In chapter three, we will outline a strategy for change based both on the organizational dimension and on the software-technical dimension. The fourth chapter will describe the constituting elements of DQM: quality principles, action programs, and measurement schemata. Their integration leads to the DQM framework, which we will outline in chapter five. The sixth chapter gives some preliminary results of DQM in action.

2 Fitness-for-Use: Considering the Quality Triangle

Quality is a multi-faceted concept. Depending on the experiences and expectations of the people involved, the quality of a software system is defined by various characteristics, e.g. efficiency, effectivity, security, usability, and portability. Although the balance between quality characteristics depends on the particular organizational setting, we can identify three elements generally underlying the quality of a software system. That is, the business process, application programs (software product) and the development and maintenance process constitute the quality triangle (Fig. 1).

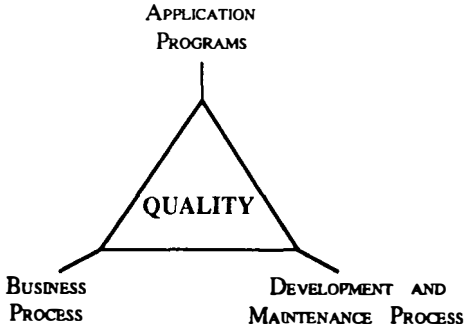


Fig. 1. The Quality Triangle

While existing approaches to quality management mainly focus on one element of the quality triangle, DQM focuses on all elements of the quality triangle. We propose this integrated view to ensure that the software meets the requirements of the various stakeholders (internal and external customers, users, software engineers, etc.), while keeping the balance between the software's organizational value and its costs. In other words, we define the quality of software by its *fitness-for-use.*

Of the many definitions of fitness-for-use, we follow Juran's and Crosby's definition that is widely used and closely reflects our understanding of the term [28, 12]: Fitness-for-use denotes the fulfillment of requirements and the conformity of the software product with its specification. Based on this general definition, we derive the constituting components of the software's fitness-for-use: the software's fitness-of-form, its fitness-for-purpose and its fitness-for-change.

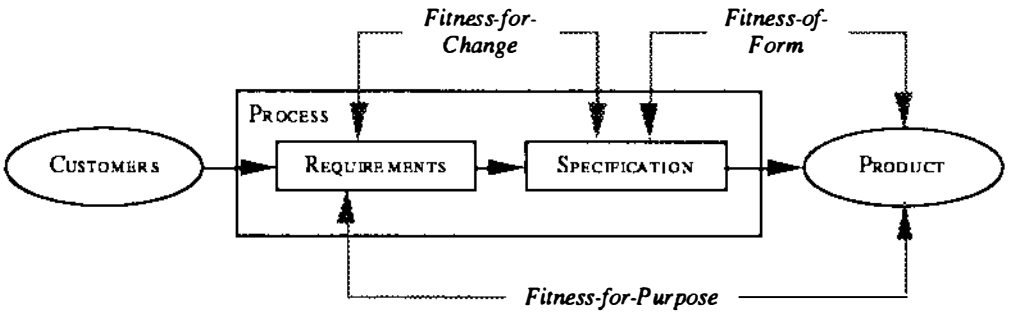


Fig. 2. Fitness-for-Use

Fitness-of-form deals with formal descriptions only, i.e. the quality of software in terms of technical measures. Thus, the fitness-of-form is solely concerned with the technical implementation. *Fitness-for-purpose* considers the environment, i.e. the business area where the software is in productive use, and deals with the question: How well does the application program fit its requirements? *Fitness-for-change* copes with the development and maintenance process of a software system.

2.1 Fitness-of-Form

In software engineering, quality management traditionally deals with the system's formal description only. For example, maintenance approaches in this vein focus on *corrective* and *perfective* maintenance. The term *verification* denotes those quality measures that are available at this level. Verification is concerned with the proof of consistency between two formal descriptions, typically the consistency of the programs with its formal specification, or the internal consistency of the specification against the demands of the specification language. Various standard verification techniques exist, e.g. static techniques and symbolic execution cf. [39]. The aim of these techniques is to define and apply quality standards that allow to measure the software's fitness-of-form automatically.

2.2 Fitness-for-Purpose

The formal domain (i.e. application programs), however, is only part of the story. To answer the question "how well does the software system fit the business process?", we have to deal with the informal domain as well. A business process combines cross-functional activities that take one or more kinds of input and creates an output that is of value to the customer [13, 21]. In addition, business processes shape the customer's perception of product quality.

To evaluate the software's fitness-for-purpose, quality measures are available that operate between an informal domain (e.g. business process) and a formal one (e.g. application program). *Validation* is the term that denotes these measures and can be performed by quality measures at the individual level (system-user interaction, work psychological aspects, etc.), at the group level (e.g. social systems and behavior), and at the organizational level (e.g. financial and non-financial measures). The levels are intertwined, and their interrelationships can be explained by the so-called "rippling effect": Someone throws a stone in the middle of a pond. This gives rise to a wave that moves outward until it reaches the lakeside. In other words, starting out with a software system influencing individual behavior, we can track the impact of a software system at the group level (communication structures, group performance, etc.) and at the organizational level (productivity, efficiency, etc.). Thus, to be valid software has to be "correct" concerning the requirements existing at the different levels. That is, they have to fit the purpose.

2.3 Fitness-for-Change

The organizational environment changes continuously, it is largely unpredictable, and it is only partially knowable [38]. Moreover, many studies have shown that the later wrong development and maintenance decisions are detected, the more expensive they are to repair cf. [9]. In other words, we have to consider the volatility of the environment. As the volatility is expressed through changing requirements, the specification's fit with the requirements has to be continuously evaluated. In many cases, the reason for inadequate software systems is that the initial specification is also the final one, thus hindering the communication and learning process of the people involved in the software project.

The fitness-for-change expresses the software's systematic adaptation and enhancement. In this process, the software's maintainability is a key factor to ensure its fitness-for-change. Maintainability can be defined as the necessary technical and organizational effort to find and correct existing errors and to avoid the reappearance of previous errors or similar ones [10]. Thereby, we are concerned with questions like: How reliable is the software when in productive use? What is the difference between specified requirements and the actual organizational and individual needs? Are the suggested enhancements and adaptations reasonable from various points of view, e.g. organizational efficiency, usability, security? Can the software adapt to additional requirements within a given time frame?

3 A Strategy to Achieve Quality: Transformation and Control

To achieve a software's fitness-for-use, we have to keep the balance between its constituting elements: fitness-of-form, fitness-for-purpose, and fitness-for-change. They are heavily intertwined. Thus, a strategy to achieve quality has to consider both the organizational dimension and the software-technical dimension:

- *Organizational Dimension.* This dimension comprises organizational change in a software's environment. Controlled organizational change is achieved through a step-wise *(total) quality transformation* of organizational processes. Let's assume, we introduce self-assessment groups for internal quality audits (code-reviews, inspections, and walk-throughs). This can require, for example, autonomous groups for software maintenance and the realization of participative leadership in this area; the installation of group-systems for documenting group and inter-group communication; and employee training to ensure efficient and effective self-assessment.
- *Software-technical Dimension.* This dimension is concerned with the assurance of the software's quality according to an improved or modified technical (software engineering) process. In achieving the product's quality goals, *quality assurance* is the major activity. Let's assume, we have decided to apply quality function deployment in a company [2, 40]. Therefore, requirements will be deployed in a way that shows their direct contribution to the software product's quality. The results from this process are permanently evaluated and thus support continuous improvement.

Although in most approaches to quality management only one of these dimensions is addressed, we can identify a common strategy to achieve quality in organizations: the "drop-the-bomb" strategy. This strategy relies on a single intervention, which is in most cases guided by external consultants. The drop-the-bomb strategy releases the "quality bomb" (additional control activities, continuous improvement teams, etc.) on the whole organization to achieve a "quantum change" in quality with a single effort (Fig. 3). While it seems plausible to concentrate ones efforts on a single intervention, this strategy does not take into account that quality cannot be bought as a general problem-solving package and installed at a site like a computer program. For example, organizational members are often overwhelmed by the mass of new ideas and activities they should incorporate in their daily work. Moreover, they have little experience in applying the introduced

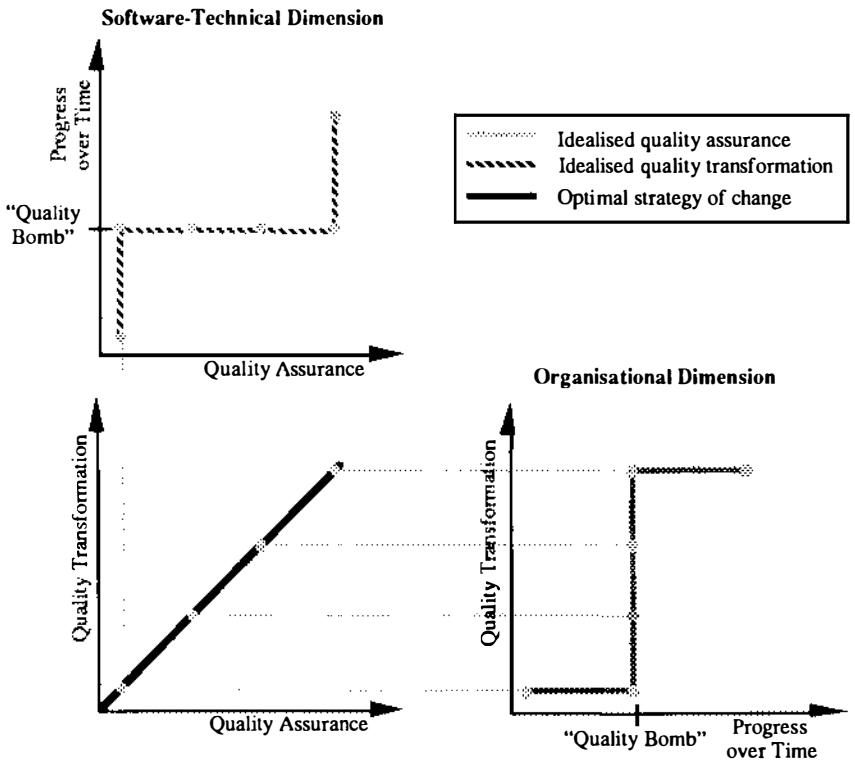


Fig. 3. DQM's gradual strategy for change

techniques and methods. Often they do not have examples of successful improvements in their organization.

Thus, we believe that quality has to "grow" in an organization. A gradual strategy to achieve quality seems appropriate. This maturity or evolution process is based on the organizational dimension and the software-technical dimension. Fig. 3 shows the idealized quality assurance and the idealized quality transformation over time as implied by the drop-the-bomb strategy. While the "quality bomb" leads to a *single* transformation in one dimension, it does not maintain the balance between them [31]. What help is it to train employees in regression tests (organizational dimension) if they do not have the necessary tools (software-technical dimension)? What help is it to restructure the employees' responsibilities (organizational dimension) if the programs in productive use are still inadequate (software-technical dimension)?

Therefore, we integrate the two dimensions to derive an "optimal" strategy of change. DQM's strategy of change equally considers the organizational dimension and the software-technical dimension. This strategy guides the controlled evolution of the current situation to reach the quality vision, i.e. to achieve and maintain the defined level of the software's fitness-for-use.

We distinguish three evolutionary steps for each quality dimension (Fig 4). The dimension *quality transformation* is composed of the steps: assessment, transition, and stabilization. Assessment can be understood as organizational diagnosis. It comprises the evaluation of the organizational effectivity in terms of quality management. The transition step denotes the gradual improvement of the current level of quality. Stabilization comprises that activities that consolidate the achieve the specified level of quality. In other words, this dimension denotes the process transforming the current state of the software product's quality to its desired level. The abscissa of Fig. 4 depicts the process of *quality assurance*. This process consists of the steps: quality awareness, quality improvement, and quality control. The awareness step sensitizes the employees for the quality vision to be achieved and for quality management methods and techniques. Quality

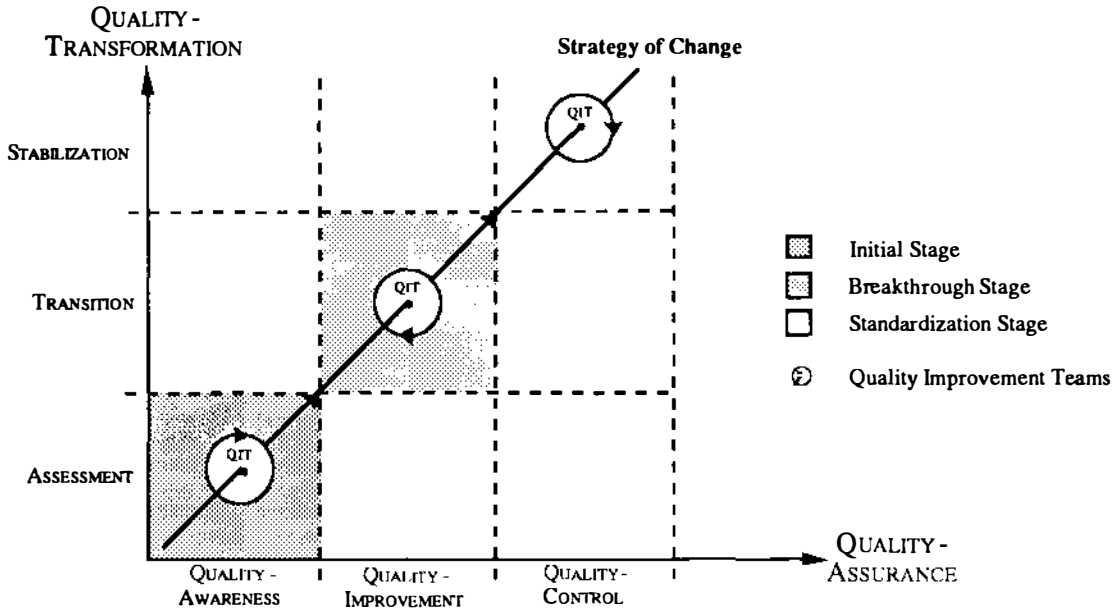


Fig. 4. The three stages of DQM's change strategy

improvement comprises the application of constructive methods of quality management to reach a higher level of software quality. Quality control is concerned with the standardization of the new processes and their continuous evaluation.

Fig. 4 provides a simplified version of the "optimal" strategy of change that consists of three "stages" and takes the possibility into account that these stages reoccur and that they are heavily intertwined. We propose three "stages" of quality management: the initial stage, the breakthrough stage, and the standardization stage. In the *initial stage*, one assesses the current process- and product-quality of the considered software systems and initiates quality awareness in the organizational unit under consideration. This should lead the employees to an ongoing activity of reflecting about what they are paying attention to, what is important to achieve, what they are trying to achieve in their daily work. When entering the *breakthrough stage*, an organization or organizational unit is ready for the transition in the pre-defined direction of change. In this stage, the application of quality control techniques is increased to permanently evaluate the application programs and the affected processes. The breakthrough stage results in a "satisficing" level of quality. In the *standardization stage*, the introduced methods and techniques for quality management are used to assure the achieved quality of the software system over extended periods of time. This stage leads to organizational guidelines and standards. In this way, we institutionalizes quality assurance by consolidating the learned methods and techniques.

4 The Ingredients of Dynamic Quality Management

To reach the quality vision, while following the "optimal" strategy of change, we based Dynamic Quality Management (DQM) on so-called *dimensional links*. These links connect the organizational dimension and the software-technical dimension (Fig. 5).

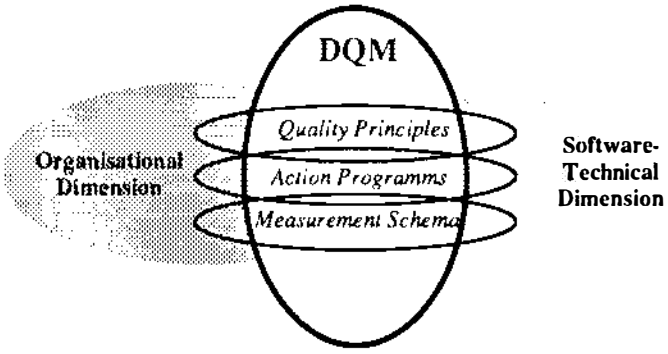


Fig. 5: Dimensional Links

We distinguish three dimensional links: quality principles, action programs, and measurement schemata. They constitute the situated instruments for planning (quality principles), performing (action programs), and controlling (measurement schemata) the adaptive actions necessary to plan and implement the software's fitness-for-use.

4.1 Quality Principles

Quality principles are selected according to key organizational and software-technical processes. In DQM, standards and guidelines for quality management systems (e.g. ISO 9000-3, Malcolm Baldrige Award) provide the characteristics defining these key processes. This leads to situated characteristics like success factors, degree of automation, process maturity, quality norms, and leadership styles. Applied to each process, they result in "process profiles" that are used by the top-management to determine so-called quality drivers. In DQM, quality drivers are processes that are essential to achieve and maintain the software's fitness-for-use.

Quality drivers are the input for DQM's planning process. They are the basis for selecting and defining quality principles. Quality principles are different combinations of methods, techniques, and leadership styles that are necessary for transforming and assuring quality in an organization. These principles are usually well-known and widely accepted [17, 28, 25, 12, 14], for example, zero-defect programs, conformance to customer requirements, the degree of user satisfaction, exceeding the customer's expectations, cost of rework, and value for money.

The selected bundle of quality principles establishes the organization-specific quality vision, guides the formalization of the organization quality policy and provides the basis for its "quality-culture". The quality principles are the "backbone" of adaptive actions. However, quality is not just an inherent property of quality principles, but emerges from the proper application of adaptive actions.

4.2 Action-Programs

Action programs have their theoretical basis in action research [32, 18]. They can be understood as projects consuming little resources (time, money, etc.) in order to control and reduce the risk of adaptive actions in every day work. They are "action patterns" based on the assessment of the current situation. Action programs can be initiated, evaluated or stopped when ever necessary. In DQM, action programs are autonomously carried out by a small group of people leading to the advantage of organizational flexibility. These so-called QITs (quality improvement teams) are distinguished by their "freedom" of action and by the self-motivation of their members.

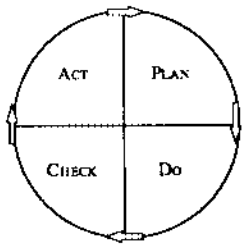


Fig. 6: Plan-Do-Check-Act Cycle

Action programs perform the personal, organizational, and technical change that has been defined in the quality principles. Thus, the results of successful action programs have to be in accordance with the planned direction of change. That is, the performed quality transformation and quality assurance matches the software's evolution strategy.

Action programs can be characterized by the Plan-Do-Check-Act (PDCA) cycle (Fig. 6). This so-called "Deming Cycle" guides the continuous improvement to reach the quality vision [14]. Its starting point is to plan for change. In a next step, the plan is implemented. However, its implementation is locally restricted to a small organizational areas in order to delimit the effects of the applied actions and to reduce the organizational and technical risks. Then the results have to be checked and analyzed (What have we learned from this experiment?). Finally, one has to decide if the plan should be put into action and implemented in organizational practice.

4.3 Measurement Schemata

To control DQM-activities, we analyze the contribution of quality principles and action programs to the planned software evolution. As the action programs are the realization of quality principles, they are DQM's point of measurement. To evaluate the improved software quality in terms of its fitness-for-use, we apply various quality control techniques that are bundled to so-called measurement schemata. We distinguish two types of measurement schemata. On the one side, they evaluate the effective and efficient realization of quality principles, for example, the frequency of software failures in terms of zero-defect principles. On the other side, measurement schemata provide a means to show an action programs' effect on software quality.

Each measurement scheme focuses on one aspect of the quality triangle:

Product Quality. Measurement schemata concerning application programs focus on quality factors like correctness, maintainability, reliability, which are derived from customer requirements [30, 23]. Following the software quality metric methodology, a quality metrics council (managers, internal and external customers) is responsible for this derivation [24]. Although the use of appropriate quality models like the goal-question-metric approach [6] or a qualitative evaluation (e.g. code-inspections) has to be considered.

Process Quality. High-quality processes are well-defined and measurable. To have such processes under control, they have to be stable over a period of time. That is, the application programs match their requirements. The process control techniques are based on quantitative evaluation (measures like cost, time, and risk). They are used to periodically evaluate the business and software-technical processes. For the software-technical process, we can use, for example, cause and effect analyses, independent validation techniques, software audits, and capability evaluation. In the business process, we can apply techniques like the requirements tractability matrix, customer complaint and feedback systems, cause and effect analysis, and customer surveys. To evaluate both dimensions we can use, for instance, ISO9000-3 auditing.

5 The DQM Framework

The DQM framework aims at the gradual quality transformation and the continuous assurance of quality. On its way to reach the quality vision, the action programs are the seeds of quality in an organization. They are initiated at carefully selected places in the organization to give rise to company-wide quality management. Thus, DQM follows a *middle-out approach.* That is, the action programs are at the "heart" of DQM. On the one side, each action program is linked to measurement schemata. On the other side, action programs can either be derived from established quality principles or initiate their consideration. For example, if a company runs an action program using attribute control charts to analyze software failures, this can initiate the application of principles based on software process control from which, in turn, additional action programs can be derived.

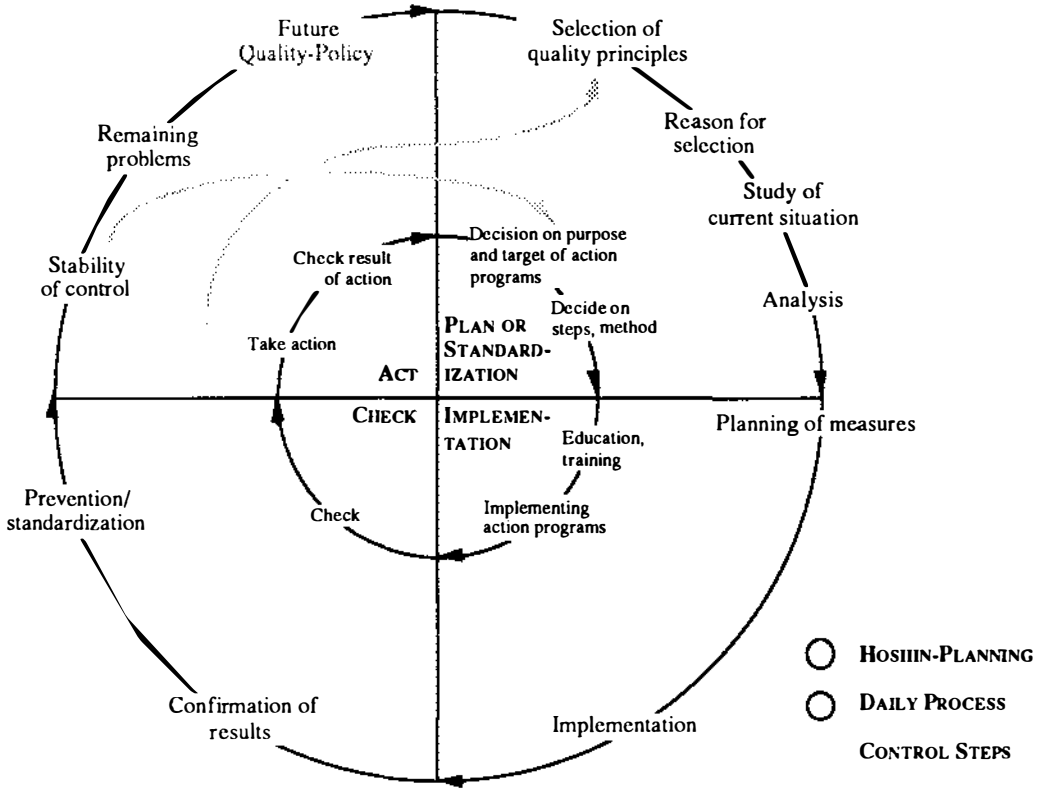


Fig. 7: The DQM Framework

The middle-out approach of DQM is based on five components that are well-accepted in the quality management community [26, 37]. The planning process, the daily process, and the improvement cycle constitute the core components of the framework. They are streaked with customer-orientation and employee-participation, which are cross-functional components. The realization of the five components is driven by (1) the current evolution phase (initial stage, breakthrough stage, or standardization stage); (2) the actual level of software-quality; and (3) the organizational, technical and individual constraints (responsibilities, capabilities, resources, etc.).

We use the "Hoshin-Kanri" method in DQM [2]. The Hoshin *planning process* (outer circle in Fig. 7). In DQM, it is applied in planning the "fit". The goal of DQM's planning process is to manage the gradual improvement of the software's quality. The planning process helps to break away from the status quo and to make a great leap up by analyzing current problems and deploying in response to environmental conditions. This process further guides the definition of quality principles (statistical process control, quality function deployment, etc.) and their dissemination by defining so-called breakthroughs, i.e. incisive improvements of the current software quality level.

The *daily process* (inner circle in Fig. 7) implements the "fit". The maintenance and development process performs the daily activities in DQM. The daily process results in a standardization of activities to assure the actual quality level. An effective and efficient process is supported by action programs, which combine necessary activities. The action programs include analytical as well as constructive quality assurance activities. The quality policy bounds the applicable actions to achieve the breakthroughs defined for a certain period of time. In addition, control items are established to check the results, the causes and the applied actions (*control steps*).

The *improvement cycle* (PDCA-cycle) is an inherent characteristic of "Hoshin Kanri". It guides the planning process and the implementation of quality goals. In DQM, assessment (qualitative) and measurement (quantitative) are applied to control the improvement of the "fit" over extended periods of time.

6 Dynamic Quality Management in Practice

We are applying DQM in one of the biggest banks in Switzerland. We are concerned with a large data processing system (DPS). This system is an important part of the bank's information infrastructure. After more than twenty years in productive use, this DPS is composed of many COBOL-programs with interwoven data structures and complex transactions.

The application of DQM's middle-out approach is guided by the actual fitness indicated by the current software-quality level and the current situation given by organizational, technical and individual constraints (responsibilities, capabilities, resources, etc.). To improve and assure the fitness-for-use of the considered DPS, its maintenance process was identified as the main quality driver. Quality drivers are thought as accelerators gaining optimal customer (i.e. bank) satisfaction and improving the software's fitness-for-use. In other words, a quality driver refers to crucial processes or specific process characteristics (e.g. costs and time). DQM based on specified quality drivers supports effective action programs and leads efficiently to the planned level of software quality.

In our project, we selected Juran's method to reach the quality vision: achieve and maintain the application program's fitness-for-use. Currently, the project is in DQM's initial stage of the planned evolution. Based on the Juran trilogy, the planning process should lead in a first step to the reduction of the percentage of software defects. The following issues provide more information on how we are planning to reach the quality vision.

Engineering Quality. The DQM project should improve the quality of the considered application programs in the whole company. However, DQM starts out with small QITs (quality improvement teams) carrying out local action programs. QITs are self-controlled work groups. They perform and control the realization of the defined quality principles. QITs are responsible for the action programs, which are based on "management-by-fact" [33]. In other words, DQM closely couples planning process and daily process. While all stakeholders work closely together to improve the software's quality, the maintenance process is used to engineer the planned level of quality in the software product.

On the one hand, the QITs periodically evaluate their actions according to their contribution to the quality goals, as formally expressed by the top management. On the other hand, an independent committee (top-management task) performs cross-sectional evaluations of all quality projects in terms of organizational cost and risk. By combining the results of both evaluations, we avoid the bias of individual judgement and get a complete "picture" of the current situation. The combined evaluation shows the contribution of each quality project to the quality vision and keeps the quality management on track.

Organizational learning and quality culture. DQM's application should start a learning process improving quality awareness in the organization [16]. Action programs are used to test competing interpretations and successful ones are incorporated in the organization quality policy. Successful

action programs are the models of change. They are the means to support the interpretation of the maintenance process in terms of the quality vision and the quality improvement teams are the "champions" of this interpretation. Important issues in the interpretation process are: (1) thinking in alternatives; (2) tracking down contradictions; and (3) change by experiments [36]. While the QITs guide the daily process, the successful change requires in addition that the new interpretations are learned and used by all individuals affected by a particular change [5, 3, 4].

Adherence to the Quality Track. The so-called quality improvement track bounds the bandwidth of the applicable actions to reach the quality vision. We use cross-functional evaluations to keep the action programs on track. The cross-functional evaluation determines the position of the action programs concerned with the planning process, the daily process and their control. When the results of an action program indicate that it is leaving the quality track, the quality management either stops the program or orders correcting actions. However, the identification of "action patterns" is at the center of cross-functional evaluation [34, 22]. That is, action programs are grouped and classified according to their goals, results, resources and history. These "action patterns" are observed over extended periods of time. Depending on their degree of organizational and software-technical success they are included in the so-called "quality patterns". That is the formal description of successful experiments in a defined organizational setting, resulting from the proper application of the DQM framework. These patterns are available in a company-wide repository (i.e. a collection of situated artifacts belonging to organizational software) to guide future actions and support the planning process. Given an organizational situation, we can use existing "patterns" to estimate, for example, the duration of a future project and its costs; or one could use the pattern as a benchmark to compare various projects.

Action programs. The action programs at the initial stage of DQM are applied to standardize the existing maintenance activities. In this process, DQM's cross-functional management ensures the participation of all departments affected by the maintenance process of data processing systems. For example, the handling of change requests should be standardized and supported by computer tools. The communication between maintenance departments should be supported by groupware systems. To improve the software's fitness-for-use, of course, additional action program are necessary, for example, (1) Fitness-for-Change: the experimental application of the so-called quality function deployment method to improve the fit between the specification and the customer requirements [2]; (2) Fitness-of-Form: an action program for the formal control of the software products. This program links a set of metrics to participatory selected quality factors; (3) Fitness-for-Purpose: Application programs requiring high maintenance efforts are often due to reengineering activities to reduce their organizational costs (time, money, etc.). This action program should define the strategic position of selected software systems to determine adequate maintenance efforts.

7 Summary

We defined the quality of software system by its fitness-for-use. Its fitness-of-form, its fitness-for-purpose, and its fitness-for-change are the "cornerstones" of underlying DQM's quality triangle. While the fitness-of-form is solely concerned with the software product, the fitness-for-purpose and the fitness-for-change deal with the business process and the software process, respectively. With DQM we proposed an integrated view on these quality elements. Thus, our strategy to achieve quality considers the software-technical dimension and the organizational dimension. The strategy comprises quality transformation and quality assurance, which are performed at the three stages: initial stage, breakthrough stage, and standardization stage. To follow this strategy and to achieve the quality vision, DQM proposed the so-called dimensional links: quality principles, the action programs, and the measurement schemata. These links provide

the basis for the DQM framework. The framework combines the planning process and the daily process by situated control items.

8 References

[1] Committee to Study the Impact of Information Technology on The Performance of Service Activities, *Information Technology in the Service Society*. Washington, National Academy Press, 1994.

[2] Y. Akao (eds.), *Hoshin Kanri, policy deployment for successful TQM*. Cambridge, Productivity Press, 1991.

[3] C. Argyris, *Reasoning, Learning and Action*. San Francisco, Jossey-Bass, 1982.

[4] C. Argyris, The Executive Mind and Double-Loop Learning. In: Th.M.A. Bemelmans (eds.), *Beyond Productivity: Information Systems Development for Organizational Effectiviness*. New York, Elsevier Science Publishers, 1984, pp. 255-275.

[5] C. Argyris, D.A. Schön, *Organizational Learning: A Theory of Action Perspective*. Reading, Addison-Wesley, 1978.

[6] V. Basili, H. Rombach, A Quantitative Approach to Quality Assurance. *Informatik Spectrum*, no. 6, 1987, pp. 145-158.

[7] K.H. Bennet, The Software Maintenance of Large Software Systems: Management, Methods and Tools. *Reliability Engineering and System Safety*, vol. 32, 1991, pp. 135-154.

[8] B.I. Blum, *Software Engineering: A Holistic View*. New York, Oxford University Press, 1992.

[9] B.W. Boehm, *Software Risk Management*. Washington, IEEE Computer Society Press, 1989.

[10] T.P. Bowen, J.T. Tsai, G.B. Wigle, *Specification of Software Quality Attributes - Software Quality Evaluation Guidebook*. AD-A153-990, Vol.I, II, III, Rome Air Development Center, Air Force Systems Command, 1985.

[11] S.D. Conte, H.E. Dunsmore, V.Y. Shen, *Software Engineering Metrics and Models*. Menlo Park, Benjamin/Cummings, 1986.

[12] P.B. Crosby, *Quality is free*. New York, New American Library, 1979.

[13] Th. H. Davenport, *Process Innovation - Reengineering Work through Information Technology*. Boston, Harvard Business School Press, 1993.

[14] W.E. Deming, *Out of the crisis*. Cambridge, MIT-Press, 1988.

[15] R.G. Dromey, A.D. McGettrick, On Specifying Software Quality. *Software Quality Journal*, vol. no. 2, 1992, pp. 45-74.

[16] R.B. Duncan, A. Weiss, Organizational Learning: Implications for organizational design. *Research in Organizational Behavior*, vol. 1, no. 2, 1979, pp. 75-123.

[17] A.V. Feigenbaum, *Total Quality Control*. New York, McGraw-Hill, 1961.

[18] W.L. French, C.H. Bell, *Organization Development*. Englewood Cliffs, Prentice Hall, 1984.

[19] J. Geiger, H.F. Hofmann, *Software Maintenance and Quality Management*. Department of Computer Science, University of Zurich, 1994.

[20] C. Gillies, *Software Quality: Theory and Management*. London, Chapman&Hall Computing, 1992.

[21] M. Hammer, J. Champy, *Reengineering the Corporation*. New York, Harper Business, 1993.

[22] C.R. Hinings, R. Greenwood, *The dynamics of strategic change*. Oxford, Blackwell Scientific Publications, 1988.

[23] H.F. Hofmann, R. Holbein, Reaching out for Quality: Considering Security Requirements in the Design of Information Systems. *6th Conference on Advanced Information Systems Engineering*, Utrecht, Netherlands, Springer, 1994.

[24] IEEE/P1061, *Standard for a Software Quality Metrics Methodology*. The Institute of Electrical and Electronics Engineers, 1988.

[25] K. Ishikawa, *Guide to Quality Control*. Tokyo, Asian Productivity Organization, 1976.

[26] K. Ishikawa, *What is Quality Control*. New York, Prentice-Hall, 1985.

[27] ISO/9000-3, *Quality management and quality assurance standards - Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software*. ISO 9000-3: 1991 (E), International Organization for Standardization, 1993.

[28] J.M. Juran, *Quality Control Handbook*. New York, McGraw-Hill, 1974.

[29] A. Kanno, Japanese Software Quality and the Deming Award. *Software Quality: The Challange in th 90s*, Madrid, 1992.

[30] N. Kano, N. Seraku, S. Takahashi et al., Attractive Quality and must be Quality. *Quality*, vol. 14, no. 2, 1984, pp. 39-48.

[31] W. Kirsch, W.M. Esser, E. Gabele, *Reorganisations-theoretische Perspektive des geplanten organisatorischen Wandels*. München, DeGruyter, 1978.

[32] K. Lewin, *Field theory and social science*. New York, 1951.

[33] C. Manz, H.P. Sims, Self-Management as a Substitute for Leadership: A Social Learning Theory Perspective. *Academy of Management Review*, vol. 5, no. 1980, pp. 361-367.

[34] D. Miller, P.H. Friesen, *Organizations: A Quantum View*. Englewood Cliffs, McGraw Hill, 1984.

[35] M. Munro, Software maintenance, reuse, and reverse engineering. *Proceedings Reuse, Maintenance and Reverse Engineering of Software: current practice and new directions*, 1989, pp.

[36] P.C. Nystrom, W.H. Starbuck, To avoid organizational crisis, unlearn. *Organizational Dynamics*, no. 2, 1984, pp. 53-65.

[37] S.S. Soin, *Total quality control essentials: key elements, methodologies and managing sucess*. New York, McGraw-Hill, 1992.

[38] T. Winograd, F. Flores, *Understanding Computers and Cognition*. New York, Ablex, 1986.

[39] John A. Wise, V. David Hopkin, Paul Stager (eds.), *Verification and Validation of Complex Systems: Human Factors Issues*. Berlin, Springer, 1993.

[40] R.E. Zultner, TQM for Technical Teams. *Communications of the ACM*, vol. 36, no. 10, 1993, pp. 79-91.

[41] H. Zuse, *Software Complexity: Measures and Methodes*. Hawthorne, deGruyter, 1990.

Verifying User Interface Designs with Usability Testing

By
Roger G. Harrison
Novell
280 West 10200 South
Sandy, UT 84070
(801) 568-8527 Office
(801) 568-8512 Fax
Roger_Harrison@Novell.COM

Abstract

One aspect of usability engineering is usability testing—the process of testing the quality of user interfaces. While usability testing is often considered a costly process, a worthwhile usability testing program can be implemented at a very modest cost. This paper presents the usability testing methodology recently implemented at 3M Health Information Systems along with a case study of a usability test and materials to aid the reader in implementing a similar program.

Keywords: usability testing, user interface design, GUI

Roger Harrison received his B.S. degree in Electrical Engineering from Brigham Young University. He has spent the last four years doing human factors research and design of user interfaces for advanced clinical systems and continues to consult in this area. He is currently employed as an Advanced Software Engineer in Novell's UnixWare/NetWare integration group. His interests include human factors, user interface design, and graphic design technologies.

Introduction

As the software industry matures, the competition among vendors for software sales is increasingly fierce. Adding features and functions to new releases of software products has been a traditional approach to competing in the marketplace, but as software grows ever more complex in its abilities and is marketed to an ever-widening audience, it seems clear that attention to software usability issues will become a critical component of the software design process for those companies that are to have a place in the market of the future. Many software companies are already differentiating their products on their ease of use. [Taylor94]

Usability engineering encompasses all product development tasks that impact the usability of a product including product concept, analysis of users needs, product design, and product development. One component of usability engineering is usability testing: the process of verifying the usability of a product's design by evaluating it under real or simulated field conditions. While the benefits of usability testing seem clear, it is often assumed that state-of-the-art usability labs with full-time usability specialists are required to take advantage of the benefits of usability testing.

Many corporations—especially those with few resources—feel the cost of beginning a usability testing program for their products is prohibitively expensive. The experience of this author is that a worthwhile usability testing program can be initiated at a cost well within the constraints of virtually any budget. This paper describes that experience of developing a budget usability testing program at 3M Health Information Systems during the past ten months. It is hoped that this information will guide the reader in developing a usability testing program.

Overview of the Usability Testing Process

Software usability tests are generally performed either to improve the user interface of a product being designed (*formative evaluation*) or to evaluate the overall usability of an existing user interface (*summative evaluation*). [Nielsen93]

In keeping with the user-centered design philosophy followed at 3M Health Information Systems [Harrison93a], formative evaluations are used during the design and prototyping stages of the user interface development. As design work begins on a product, usability goals are based on the anticipated users of the system. An iterative rapid user interface prototyping methodology is then employed to create prototypes of the user interface for the product. Each iteration of prototyping is concluded with usability testing on "typical" users of the system to determine if these usability goals have been met. The results of each test are used to make improvements in the next iteration of the user interface prototype until the usability goals are met.

Creating a Test Plan

In preparation for a usability test, a test plan is created to organize the testing process. The test plan includes the following:

- A statement of the test objectives.
- The usability criteria that will be measured.
- A profile of the desired test participants.
- A general plan for finding and screening suitable test participants.
- A description of the environment, equipment, and personnel needed for the test.
- A time line of tasks to be completed in preparation for the test.
- A list of people who are willing and available to be involved in planning and/or administering the test.
- The projected use and distribution of test results.

Test Objectives

Key areas of usability should be identified during user task analysis and specification of the system functionality, and usability benchmarks should be set for these areas. Generally, the objectives of the usability test will revolve around testing whether one or more of these usability benchmarks has been met. Deficiencies discovered by the testing are corrected in later iterations of the prototyping/testing process.

Establishing Usability Criteria

Quantifiable criteria are important in objectively assessing a system's usability. Ideally, usability criteria should directly support meeting key usability targets; minimally, they should be oriented toward improving usability in these key areas. There are five measures traditionally associated with usability [Nielsen93] [Bailey82]: learnability, memorability, efficiency, errors, and subjective satisfaction.

- *Learnability* is the type and amount of training required to bring users to a desired level of performance.
- *Memorability* addresses the ability to retain skills in using a product once it is learned, especially when a long period of time elapses between uses of the product.
- *Efficiency* measures the speed with which tasks can be performed or the quantity of tasks that be performed in a given time.
- *Errors* measures the number of incorrect actions a user makes in trying to accomplish a task.
- *Subjective satisfaction* is the user's overall feeling about using the software; if using the software gave them satisfaction—in short, if they liked it.

Usability criteria should generally be concrete and measurable in order to draw meaningful conclusions from the test results. For instance, rather than stating a usability goal as "the product should be easy to learn," say instead, "the user should be able to successfully create a memo with indented paragraphs, bold and italic text, page headers (including page numbers), and a simple 4x4 cell table in less than 45 minutes."

Make sure that the usability criteria will answer specific questions about the usability of the product or will give you information to support or reject a specific user interface design decision. Even subjective satisfaction can be measured in relative terms by asking users to rate aspects of satisfaction regarding the system.

Profiling and Finding Test Participants

To make the test results meaningful, it is important to test the prototype on typical users. In some cases, this population may be very homogeneous; in other cases it may be necessary to identify several distinct or overlapping groups of users in order to accurately model a diverse user population. The latter situation is especially true of products, such as word processors, that are used by a wide audience and for varying tasks.

Generally the product's user population is profiled during the task analysis and specification of the system functionality. For the purposes of usability testing, one or more subsets of this population are selected to match the objectives of the usability test. Among the user characteristics to consider are the following:

- General computer literacy level and comfort level.
- Familiarity with mouse, windows, and other specific user interface elements.
- Work or educational experience with tasks accomplished by the software.
- Work, educational, or recreational experience with software similar to that being tested. This may include entire software applications or just salient tasks within software applications.
- Familiarity with other types of software that may influence the results of the usability test.

Although creating an accurate user profile for a product can be difficult, an even more daunting task may be finding people who actually fit this profile! This is especially true when one is beginning a usability testing program because building a ready pool of potential test participants takes time. Therefore, it is imperative that due consideration be given to methods for finding suitable test participants in the planning stages of the test to allow plenty of time to find them.

Some good sources of potential test participants are university or college students, temporary employment agencies, current users, and professional societies whose members are likely to fit the desired user profile. For initial or very informal tests, one might even use work colleagues or walk-ins to a computer store. [Tognazzini92]

Care should be taken not to rely too heavily on any source of test participants because most sources have potential drawbacks. [Rosenbaum92] Students may not fit the desired profile. In addition, students tend to be young and thus may not be representative demographically. Agencies may send people who just want jobs. Current users may be too experienced or have acquired habits that are not typical of the target population. Internal candidates are rarely naive enough, and their perceptions may be biased by corporate culture in subtle and often-invisible ways. Well-designed profiles accompanied by thorough screening of potential test participants can overcome most of these limitations, but it is wise to be aware of them.

The Usability Testing Environment and Personnel

Usability testing is often equated with elaborate laboratories full of electronic recording equipment and observation booths behind one-way mirrors, but the expense of equipping, staffing, and maintaining such facilities is often prohibitive, especially for smaller organizations or those just beginning a usability engineering program.

While there is little doubt that state-of-the-art equipment can be beneficial to the usability testing process, it is possible to get valuable and satisfactory usability results with a very modest investment. A usability lab may consist of nothing more than a room temporarily set aside for that purpose containing a table, an appropriate computer workstation, and several chairs. During the test, the test participant uses the software while under the observation of one or more people who record their comments on paper. Audio or video tape might be used as a secondary record of the event to fill in gaps or provide other feedback not easily captured on paper.

One observer also serves as the test administrator and familiarizes the test participant with the testing equipment, the room, the other observers, and the purpose of the test. Of all the observers, only the test administrator interacts with the test participant during the test. Because of the temptation to discount the existence of usability problems or to be too helpful if a test participant encounters difficulties during the test, it is wise for the test administrator to be an impartial observer. This generally means that the test administrator is not a designer of the product, although designers are welcome to participate as silent observers.

Use and Distribution of Test Results

The intended use and audience for test results will largely determine the way that they are collated. If the audience is composed of designers who are also involved in the test as observers, many of the test results will be self-evident and a summary of key findings may be sufficient. If the audience is managerial or sales staff, it may be helpful to edit a full length video tape of one or more tests into a short highlights video to demonstrate key successes of the testing program.

Since the whole focus of the iterative prototyping/testing process is to reduce the costs of finding and correcting user interface deficiencies, it seems wise to use the simplest and least costly method of collating and disseminating feedback from the usability test.

Creating Test Documents

Once a general test plan is in place, several documents are created to facilitate the testing process. These are a screening questionnaire, a test administrator's script, a test script, a post-test questionnaire, and optionally, a legal release form. Developing initial drafts of these documents can be fairly time consuming, and while some of these documents are essentially new for each test, some can be re-used partially or entirely from test to test. For this reason, examples of some of these documents are available in appendices to this paper to aid the reader's efforts in creating them.

The Screening Questionnaire

Finding representative users to participate in a usability test is often a time-consuming task. To speed the process of interviewing prospective test participants and evaluating how closely they match the desired user characteristics, we use a fill-in-the-blanks screening questionnaire.

The questionnaire is designed to anticipate potential responses from participants based on desired user profiles in such a way that most responses can be recorded by writing a single check mark, letter, or number. Figure 1 demonstrates this approach with a table that allows the interviewer to quickly record responses regarding a participant's experience with several common types of applications and the frequency with which they are used.

Software	System	Length of Time	Self Rating (N/I/E)	Amount of Training Received	Frequency				
					Daily	3-4 times per week	Once per week	Once per month	Never

Figure 1: Sample screening questionnaire.

Include questions specific to the type of product and the type of test. It may be helpful to include some "backwards" questions—where the desired answer is negative rather than positive—since some potential participants will try to qualify for the test by answering every question affirmatively. [Harrison93b] It may also be possible to quickly screen out unsuitable participants with one or more key questions regarding core user traits at the beginning of the questionnaire. More is said on the screening process in the section titled Screening Test Participants.

After the screening interview, the tabular format for the data makes it easy to compare prospective test participants and quickly locate the most likely candidates for the test. The tabular format can also be readily adapted to allow scoring of the participant's match to the desired profile.

The Test Administrator's Script

The test administrator's script is designed to act as a standard way of providing the participant with pertinent information prior to the commencement of the test. Points made in the script should put the test participant at ease, acquaint him with the equipment and people in the room, and review what is expected of him during the test.

Because the administrator's script doesn't change significantly from test to test, it is memorized and recited in the administrator's own words. This helps the script to seem more natural and gives the administrator the flexibility to increase or decrease the time spent on various points (such as taking extra time to put an unusually nervous participant at ease). Care is taken to be certain that the concepts embodied in the script are followed closely to reduce the variability in training between different administrators and testing sessions.

More is said regarding the administrator's script in the section titled "Conducting the Usability Test." A complete sample of a test administrator's script is contained in Appendix 1.

The Test Script

The test script is the written information given to the test participant. The script contains a list of tasks that the user is to perform during the usability test. The tasks should be described in general non-leading terms to ensure that the test results are not skewed by telling the user how to accomplish the tasks.

While not necessary, it can be helpful to put the tasks into context by creating a scenario in which the user is doing them. For instance, a test of word processing software might contain a scenario such as:

> You are a temporary employee on your first day as an administrative assistant at XYZ Corporation. Your manager hands you the following outline for a memo including figures on last month's sales...

A copy of the outline would appear in the script, then the scenario would continue:

> *You have a copy of a recent memo which you can use as a model for the one you have just been asked to write.*

The sample memo is in included in the script. Next comes a series of tasks.

> ***Task 1*** *Open a new document and create the heading of your new memo as in the sample.*

Task 2 *Type in the text of the memo following the style of the sample memo.*

Task 3 *Create a table and enter last month's sales figures into it.*

The scenario should be representative of a typical experience with the product and should match the profile of the test participant. Also, the tasks should make it possible to measure how well the software is performing against the usability criteria established in the test plan.

Other written information regarding the software being tested may also be included in the test script if it meets the objectives of the test. For instance, a test designed to determine if a user can use a short written tutorial to accomplish a task would include that tutorial as part of the test script. If this type of information is included, care should be taken to ensure that it is both appropriate and necessary so as not to spoil the test results.

The Post-Test Questionnaire

After completing the tasks in the test script, the participant is given a post-test questionnaire. The format of the questionnaire varies based upon the objectives of the test, but the questionnaire is generally composed of some combination of multiple choice, short answer, and essay questions. Usually the questions posed in the post-test questionnaire are designed to assess the user's satisfaction with the software, to test conceptual understanding of the software's user interface, and to allow the user to give general feedback based on the usability testing experience.

Multiple choice and short answer questions are naturally suited for written response. It can be intimidating for someone to work on a such a questionnaire while under observation, so the test administrator and any observers should leave the participant to work in private for a few minutes.

In practice, an oral format for essay questions works well because it is generally quicker to speak responses. When the oral format is used it is important to have some recording device recording the answers verbatim to ensure accuracy in transcription. It may also be helpful for the test administrator to get clarification on written and spoken answers in the questionnaire.

A number of mistakes are commonly made in developing written evaluations. The following guidelines [Prekeges93] may prove useful in avoiding several of them:

- When using a numeric scale for a question, avoid changing the meaning of the values in the scale from question to question. People tend to start answering one way and stop looking at the words next to the numbers.

Example 1: "Best" answer sometimes at one end, sometimes in the middle.

Bad				Good
1	2	3	4	5

Too Much		Just Right		Too Little
1	2	3	4	5

Example 2: Reversing the order of the values from question to question. This is sometimes done in the erroneous belief that by varying the order the reader will have to consider the answer more carefully and thus the results will be more valid.

Bad				Good
1	2	3	4	5

Good				Bad
1	2	3	4	5

Bad				Good
1	2	3	4	5

Good				Bad
1	2	3	4	5

Bad				Good
1	2	3	4	5

- When asking a participant to rate something on a numeric scale, pick a range with an odd number of choices (for example a scale from 1 to 7) to allow a neutral choice by picking in the middle. An even number of choices (such as 1 to 6) forces the participant to choose a side.

- Avoid words that can be interpreted in more that one way. In the example below, is "OK" seen as "good" or as "neutral?" Depending on how a subject interprets it, neutral could be either a 2 or a 3 depending on personal interpretation.

Very Bad		OK		Very Good
1	2	3	4	5

- Avoid questions that have similar response structures but different response methods. In the example below, question 1 allows both X and Y to be rated as 1 if they have equal importance. In question 2, the choices are mutually exclusive, but a person working quickly might misinterpret one or both questions because of their similarity.

1. Assign a priority (1 = low, 3 = high) to each of the following:

   W: ___ X: ___ Y: ___ Z: ___

2. Assign the order in which the following were used:

   W: ___ X: ___ Y: ___ Z: ___

The Legal Release Form

Usability testing represents one of the first points where the public might gain information regarding the existence of a software product, and thus it may be necessary to protect that information. A legal agreement between the test participant and the legal entity that the software designer represents may be prudent to avoid potential disclosure of proprietary information or misunderstanding between the parties on other matters.

Among the points to be considered for inclusion in a legal release are:

- A non-disclosure clause restricting the dissemination of information about the software being tested.
- An indication that the software being tested does not represent a commitment by the company to meet any particular design or to even release software in the future.
- A release for use of ideas and information resulting from the test.
- The terms of remuneration for the test participant.
- A release for the company to use any audio or video recordings made during the test. This is particularly true if there is any intention of using the recordings in a public setting. Often participants are pleased when these rights are fairly restrictive because they do not wish to suddenly see themselves in some usability commercial in the future.

Screening Test Participants

Screening test participants can be done by phone or in person. Begin by explaining that general purpose of the interview is to find people who can help evaluate how easy it is to use some software. Explain that it is important to ensure that test participants have a specific type of experience, then ask if it is okay to go through a few questions to determine if the person has the type of experience desired.

Record the potential participant's responses on the screening questionnaire. If a candidate seems suitable for the test, confirm his ability to participate in the test on the date when it will be held. Give the candidate information on remuneration, any paperwork that may need to be completed, proper dress for the test, directions to the test site and information on reimbursement for travel or mileage expenses. [Harrison93b]

If a candidate is unsuitable for the test, thank him for his time and explain that you are looking for people with different experience. Ask if it would be okay to contact the candidate in the future when you need someone with his experience, and ask for suggestions of other people who might be interviewed for the test. [Rosenbaum92]

Testing the Usability Test

Before the actual test, it is a good idea to do a dry run of the usability test to work out any unforeseen problems with the test or test equipment. Conduct the dry run in an environment as much like the real test as possible. Use all documents exactly as prepared for the real test. Be sure to note any missing equipment or other problems so they can be solved before the real test. If substantial changes are made to the test as a result of the dry run, a second dry run might be warranted. This simple step will avoid unnecessary frustration and embarrassment during the actual test.

Generally, it is not necessary to use a "real" test participant for the dry run, especially since good participants are often a scarce resource. Do pick someone who is a relatively similar to the "real" users, if possible, since someone who knows too much about what you're doing may not notice problems with instructions or test materials. It is worth noting that some preliminary results regarding the usability of a product can often be noted from observing a dry run of the usability test. While these results may not be as reliable as the results from tests with participants matching the user profile, the information can still be useful.

Preparing for the Usability Test

At least one day before the usability test, reconfirm the date, time, and place of the test with each test participant.

Before the test begins, review the test plan to ensure that all equipment and test materials are ready and in good working order. Double check problem areas found during the dry run of the usability test.

Conducting the Usability Test

As the test begins, the test administrator follows the test administrator's script. The following dialogue [Gomoll90] demonstrates how the administrator's script might be used:

- Describe the purpose of the usability test. For example, the administrator might say:

We are testing the usefulness of a new reporting system called "Reporter."

*We would like you to use this computer terminal and the Reporter software as you perform the test. Keep in mind that we are testing the software—**not how well you perform the test.** We are interested in discovering where you have difficulties so that we can make improvements to the system.*

Participating in this test is completely voluntary. I don't know of any reason for this to happen, but if you become uncomfortable or find the test objectionable in any way, it's OK to quit at any time.

- Acknowledge the equipment in the room.

  As you can see, we're using a camera to record this session. The camera is on now and will be until we are done. The camera is a much better note taker than I am, and helps us later on to remember all your comments and questions.

- Demonstrate the think aloud protocol.

  As you do the test, I also want you to think out loud. Read any instruction out loud, and then talk yourself through the procedure so that the video camera and I know what you and thinking and what you are trying to do. For example, if you need to press the ENTER *key, say out loud, "I'm looking for the* ENTER *key."*

- Give any necessary preliminary training. For example, if the user profile assumed some basic familiarity with mouse and windows interaction, a short tutorial might be given to be certain that each participant has at least a basic understanding of mouse and windows concepts.

- Introduce the test script and explain the observers' role in the test.

  Here is the test. Let me warn you that the test does not give you step-by-step instructions or a lot of detail. This is because it is important for us to find out how you do with the software on your own.

  These people (acknowledge any other observers by name) will be sitting here behind you so that they can see how you use the software. They are just observers and won't be participating in any other way.

- Explain that you will not provide help.

  As you're working through the exercises, I won't be able to provide help or answer questions. This is because I want to create the most realistic situation possible.

Even though I won't be able to answer most of your questions, please ask them anyway. I'll note your questions, and when you've finished the exercises, I'll answer any questions you still have.

- Ask if there are any questions before you start; then begin the test.

During the test, the test administrator and observers take notes while the participant works through each task. It may be necessary to occasionally remind the participant to use the think-aloud protocol, especially because some people tend to become very quiet when they are confused or uncertain. This is precisely when the protocol is most helpful, and a gentle reminder or a question such as "What are you thinking now?" is generally sufficient to get the participant to resume following it.

The administrator should not express any personal opinions or indicate whether the user is doing well or poorly. The administrator may make non-committal sounds like "uh-huh" to acknowledge comments, but care should be taken not to express any opinion through the tone of voice used in such comments. [Nielsen93]

It can be very tempting to help a participant through the tough spots of the user interface, but this is counterproductive to the test's objectives. If the participant is clearly stuck and getting unhappy with the situation, the administrator may choose to provide limited assistance to get the participant moving along. Most often, however, administrators tend to err by helping too early and too often. Another situation where assistance is warranted is when a bug in the prototype has caused the user's distress. In this case, the software should be returned to a usable state as near to the point where the bug occurred as possible to let the test continue.

After the test, the user is debriefed and asked to fill in the post-test questionnaire. Any discussion of the system or the test between the administrator and the participant should be postponed until *after* the written portion of the questionnaire is completed to avoid biasing the participant's answers.

The administrator can use this time to answer any questions the participant may still have and to get clarification on any parts of the test with which the participant had difficulties.

The administrator should end the debriefing by thanking the participant and explicitly stating that the test helped to identify areas of possible improvement in the product. It may also be necessary to mention that the development team may not be able to correct every usability problem discovered by the test in order to leave the participant with reasonable expectations regarding the outcome of the test. [Nielsen93]

Evaluating the Usability Test

After the usability test, results should be compiled and a report prepared. Depending on the audience the report may be an informal presentation to development staff, a formal written document, or a video presentation containing highlights of the testing. In any case, feedback

should be given on how well the software met the usability criteria established in the test plan. Areas needing improvement should be identified along with suggestions for making needed changes. This information should be incorporated into the next iteration of the software prototype.

A Case Study

A recent software development project made use of the usability testing process described in this paper to pilot a usability testing program at 3M Health Information Systems. The project began as a small group of employees with interest and experience in the field of human factors who wanted to improve the usability of new software products in the early stages of development. The reporting package for a clinical information system was picked as the first product to be iteratively designed and tested.

Because this was the first attempt at such a program, a fair amount of time was spent in developing a prototyping/testing philosophy and preparing initial drafts of test documents. Upon completion of the test documents, usability tests were performed and the results were used to improve the software.

Test Objectives

The software being tested was to replace an existing product, therefore several of the usability goals were related to improving areas of concern with the existing reporting software. A reduction in training time and associated training staff responsibilities was primary among these areas. Also, the design team wanted to gather information regarding the new software's conceptual model. Finally, because this was a pilot project, the team wanted to demonstrate the benefits of the usability testing process to management in order to gain support for future efforts.

The test goals included the following:

- Determine if novice users can successfully complete the most common tasks required for reporting such as printing pre-defined reports and creating typical ad-hoc reports without any training.
- Measure the time required to perform these basic tasks against the training time currently required for users of the existing reporting software to begin performing these tasks.
- Gather evaluative information regarding key components of the software's conceptual model such as whether users could see how a report was based upon a format (which determined how the report's printed appearance) and a population (which determined the records to be included in the report).

Test Participant Profile

It was determined that test participants needed to have the following user traits:

- Background in medical records or health care related Quality Assurance (QA) or Utilization Review (UR) experience.
- Some experience with the reporting needs of a hospital medical records department.
- Some experience creating and generating reports, either by computer or by hand.
- Minimal experience using computers, especially in any function outside of hospital medical records.

Because this was the company's first experience with usability testing, there was some concern about disclosing the existence of the software development project to potential clients so early in the development cycle. As a result, several suitable test participants were located from within the organization. Each participant had the proper experience from prior employment but had not been directly involved in using the existing reporting software or developing the new reporting software. It was assumed by the team that approval would be given to continue usability testing on more realistic users as the value of the project was demonstrated.

Test Tasks

A set of basic tasks was assembled. They were chosen to reflect the basic tasks necessary to successfully use the reporting software and were modeled on the types of reports needed most often by new users of the software. The test included the following tasks:

- Find and print an existing report.
- Create a new report.
- Add fields to the report.
- Rearrange the fields to appear in a new order.
- Add two levels of sorting to the fields in the report.
- Replace an incorrect field in the report.
- Define a filter to select the records of specified patients for inclusion in the report.
- Change the filter to correct a "mistake."
- Print the new report.
- Save the new report.

The test tasks were incorporated into a scenario in which the participant was the director of medical records in a hospital (a very typical user) and needed to use the "new" reporter software that had just been installed to prepare some reports while everyone else was away.

Resources

The resources available to the usability testing group were limited. All of the people involved had other primary job responsibilities in addition to their involvement in the group. Members of the group included four technical writers and one engineer. Short meetings were held weekly to ascertain progress on the test plan and other test documents. The testing facilities

included a small conference room with a table, several chairs, a computer workstation, and a VHS video camera. The only costs to the project were employee time, printing and copying of test documents, and the purchase of blank video cassettes.

The Dry Run

Several dry runs of the test were necessary to satisfactorily work out the problems. Among the issues encountered during the test were fatal bugs in the software prototype, unclear information in the test documents, and missing information in the test documents.

The dry runs also highlighted several large usability problems with the prototype which prompted some changes in the user interface before the actual testing began. For instance, at a point where the user was to schedule the printing of a report for some future time, users generally scheduled the print job correctly, but when they left the scheduled printing dialog, they weren't sure if the report would actually print at the time they had chosen. A single line was added to the print dialog near the scheduled printing button that stated the time the report was scheduled for printing. This effectively solved the problem encountered during the dry runs, and the testing team noted (with some satisfaction) that users used this feedback during the actual test to confirm that they had completed the scheduled printing task correctly.

The Usability Test

The test was conducted with three participants. One member of the usability testing team acted as test administrator and the software engineer who created the prototype observed the test. Each test took approximately two hours to complete including the time used for orientation and the post-test questionnaire.

Test data was gathered primarily through the use of written notes; a video recording of the test was kept as a complete and permanent record.

Test results

The usability test effectively demonstrated that training time for the reporting software could be drastically reduced or eliminated altogether. Every user was able to successfully accomplish all of the test tasks in 1 to 2 hours of self-guided exploration.

A simplified approach to data selection that was tested as part of the prototype was shown to be quite understandable by non-programmers.
he usability test also highlighted some areas of concern including the need for feedback in several key areas, places where the user interface gave users the wrong cues, and cumbersome user interface mechanisms. Some of these areas were identified during dry runs of the test, and the actual test validated several changes made to correct these deficiencies.

While the usability testing effort was obviously an initial and imperfect effort, it demonstrated that the program was valuable. Management and technical staff support have both increased in the months following the initial test. Also, much of the initial effort in developing the

program did not have to be duplicated for future tests. By reusing the test administrator's script and parts of the test plan and screening questionnaire, the time required to prepare for recent tests of other products has been significantly less.

Unfortunately, the iterative development cycle of the reporting software was cut short when development on the project was halted due to a realignment of business priorities. Thus while the results of the initial tests were promising, they are also incomplete.

Overcoming Management and Cultural Obstacles

The usability testing program at 3M Health Information Systems began as a grass-roots desire in several groups to create better, more usable software for clients. As work on the program has progressed, several key elements seem to have contributed to its success and growth.

Adapt and mold the program to existing corporate culture.

Rather than forcing a new methodology into our current work patterns, we have tried to adapt the best parts of those established patterns along with sound usability engineering principles. As a result, engineers and other technical staff are much more willing to adopt usability engineering techniques because they feel that they own the process.

Collaborate informally with others who want to improve usability.

Rather than trying to organize a formal department with its accompanying need for managerial approval for dedicated resources, we constantly searched for individuals within the corporation who wanted to improve usability and were willing to lend a hand to our efforts. As we found these people, we invited them to participate in various aspects of the usability testing effort. This provided us with a broad base of support from throughout the organization at very low cost and provided the means to quickly begin a usability testing program without getting bogged down by corporate bureaucracy

Start small and build on successes.

The usability testing program at 3M Health Information Systems has grown by demonstrating its value with small but important initial successes and then building upon those successes. As word of the program has spread, those interested in participating have been included in both the development and implementation of the program until virtually every development group within the organization is currently represented in some way.

Management has given increasing support to the effort as the benefits of the program have been demonstrated. For instance, in initial usability tests management was reluctant to disclose new product development plans to people outside the organization even though these were the very people needed to get the most benefit from the usability testing process. Due to this circumstance, initial usability tests were conducted using the few employees who had the proper educational and professional backgrounds of the user profile. The results of those tests

were convincing enough that management has allowed the use of "real" users in subsequent tests. As the value of the program continues to be proven, it is anticipated that more resources and financial support will be forthcoming.

Show how usability can improve the bottom line.

Not long after the initial usability testing program began at 3M Health Information Systems, a business analysis showed that training and support were large expenses for certain types of products. As a result, the usability testing team has focused its efforts on helping the company reduce training and support costs for these products. Our work to date has shown that training costs can be reduced dramatically when proper usability engineering takes place which has encouraged management to increase support for our effort.

Conclusions

Attention to usability issues is increasingly key to the success of software in the marketplace. Usability testing, employed as part of a user-centered design process, can provide valuable feedback to software designers to aid them in producing the best software possible

While 3M Health Information Systems is still relatively inexperienced with regard to usability testing, the recent development of its usability testing program has shown that an effective usability testing program can be developed at a modest cost to any organization, even one with limited resources.

Appendix 1: Administrator's Script

[Start this script after greeting the subject and inviting him/her to be seated next to you in front of the terminal. Introduce yourself and other observers, if necessary.]

We are testing the usefulness of a new reporting system called "Reporter."

We would like you to use this computer terminal, the Reporter software, and this document called "About the software" as you perform the test. Keep in mind that we are testing the software—*not how well you perform the test.* We are interested in discovering where you have difficulties so that we can make improvements to the system.

[Turn and acknowledge the camera equipment.]
As you can see, we're using a video camera to record this session. The camera is on now and will be until we are done. The camera is a much better note taker than I am, and it helps us later on to remember all your comments and questions.

Think-aloud protocol

As you do the test, I want you to think out loud. Read any instruction out loud, and then talk yourself through the procedure, so that the video camera and I know what you are thinking and what you are trying to do. For example, if you need to press the ENTER key, say out loud, "I'm looking for the ENTER key."

Is this clear?

Let's take a minute and try it out. I brought along a travel alarm clock, and I'm going to set the time to 6:15 and then set the alarm to go off at 7:00. I'll think out loud as I do these things to show you how this works.

[Proceed to demonstrate the think-aloud protocol.]
"I'm looking at the back of the clock to see how to set it."

"There are two dials. One has what appear to be little sound waves by it, so that must be the alarm. The other dial shows hands on a clock, so that must be the time setting."

"I'm using the time setting first. It moves both hands together in either direction, so I'm moving them to 6:15."

"Now I'm using the other dial to set the alarm. It seems to move the red alarm hand in both directions, too. So I'm moving that to the seven."

"Now I need to turn the alarm on. This button on the top looks like where I would turn it off, so I'll just push it up to the "on" position."

That's what I mean by thinking aloud. You don't need to read all of the documentation out loud, but it is a good idea to read the test task aloud so that we know where you are in the test. Do you have any questions about this think aloud protocol?

Okay. I think we're about ready. Here is your documentation and test. If something is confusing in either of these documents, mark the spot with an X or a question mark in the margin. Then we can go back and see what areas need revising. Let me also warn you up front that the test and documentation do not give you step-by-step instructions or a lot of detail—this is intentional. It is important for us to find out how easy or hard this software is to use. If we give you too many instructions then we'll be testing the instructions instead of the software. Don't forget to think out loud!!

It's really important that you do the tasks in order on the test-—don't skip ahead. If you really get stuck and can't complete a part, ask me for help.

Roger and I will be sitting here behind you so that we can see how you use the software and documentation. Roger is strictly an observer, and I'll try not to get involved or interrupt you unless you ask me to. Remember—it's very important to be totally honest in your evaluation, even if it's brutal! We want your true opinion about this package, so please don't hold back any comment or response, positive or negative.

Bibliography

[Bailey82] Bailey, Robert W. *Human Performance Engineering: A Guide for System Designers.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1982.

[Gomoll90] Gomoll, Kathleen. "Some Techniques for Observing Users," *The Art of Human-Computer Interface Design.* Reading Massachusetts: Addison-Wesley Publishing Company, Inc., 1990.

[Harrison93a] Harrison, Roger and Larry Cousin. "Improving Application Software Quality Using Rapid GUI Prototyping," in *Proc. of Pacific Northwest Software Quality Conference*, Portland, Oregon, October 18-20, 1993.

[Harrison93b] Harrison, Roger. Personal notes from Northern Utah Computer-Human Interaction meeting based on paper titled "Stalking the Wily Test Subject." 1993.

[Nielsen93] Nielsen, Jakob. *Usability Engineering.* San Diego, California: Academic Press, Inc., 1993.

[Prekeges93] Prekeges, James G. "Usability Testing for Non-Testers." Annual Conference of the Society for Technical Communication, 1993.

[Ramey92] Ramey, Judith, PhD. Usability Testing of the Documentation and User Interface of Computer Systems, Course. Department of Technical Communication, College of Engineering, University of Washington. September 14-16, 1992.

[Rosenbaum92] Rosenbaum, Stephanie. "Selecting Appropriate Subjects for Usability Testing." Ann Arbor, Michigan: Tec-Ed, Inc.

[Taylor94] Taylor, Suzanne. "Intuit Focuses on Creating Wow!", *QuickNews.* Intuit Corporation. Spring, 1994.

[Tognazzini92] Tognazzini, Bruce. *TOG on Interface.* Reading Massachusetts: Addison-Wesley Publishing Company, Inc., 1992.

A Feature-Oriented Software Life-Cycle

Herwig Egghart Edgar Knapp

Purdue University
Department of Computer Sciences
West Lafayette, IN 47907-1398
{egghart,knapp}@cs.purdue.edu

July 28, 1994

Abstract

The complexity introduced into modern applications by scores of "features" is a threat to software quality. Our methodology helps mitigate this threat by keeping track of features throughout the life-cycle. Among the corner stones of our methodology are the reliance on incremental enhancements, the explicit specification of relationships between features, and the mapping of software features to the code that implements them. The feature-oriented approach has the potential to lead to increased software quality, better system understanding, and reduced time-to-market.

Keywords

Configuration Management, Design Methodologies, Features, Feature Dependency, Feature Interaction, Life Cycle, Software Quality, Software Understandability, System Analysis and Design, Version Control.

Biographical Sketches

Herwig Egghart is enrolled in the Ph.D. program at Purdue University. He has worked as a senior software engineer for Tele Control, Vienna from 1989 to 1993 and is interested in interrelating his practical experience with results from the scientific community. His main concern is how rigorous techniques can help us manage complexity and overcome mediocrity in software development. He graduated as a Diplom-Ingenieur from the Technical University of Vienna in 1991; his Master's Thesis is entitled *Specification of a Universal Ticket Pool for Betting Systems.* He received a Fulbright Scholarship in 1993.

Edgar Knapp is an Assistant Professor of Computer Science at Purdue University. His main research area is Programming Languages and Systems, specifically he is interested in increasing the quality of sequential and concurrent software through application of mathematical techniques to software engineering. He received his Ph.D. from the University of Texas in Austin in 1992. His dissertation is entitled *Refinement as a Basis for Concurrent Program Design.* He is affiliated with the Software Engineering Research Center (SERC) at Purdue, a consortium of Industry, Government, and University researchers and software developers. Within SERC, he is working on integrating formal techniques of software construction with more traditional software engineering approaches. He is also actively involved in research and teaching of Object Oriented Technology, both in academia and industry.

1 Introduction

A pressing concern of the software industry is decreasing time to market without sacrificing software quality. Yet, the inherent complexity of modern software systems makes the goals of rapid delivery of applications and the attainment of high quality standards appear mutually exclusive. Much of the complexity of current software is caused by the abundance of features they support.

Many UNIX tools have upwards of 30 options, off-the-shelf commercial software has manuals totaling thousands of pages. To combat the inherent complexity of large software systems, it is necessary to break their structure down into manageable pieces. We see three dimensions in which this can be done:

Modular Programming, the oldest approach to hierarchical decomposition, attempts to organize system components around collections of related tasks.

Object-Oriented Programming, a more recent paradigm, organizes the data structures of a software system around collections of objects.

Feature-Oriented Programming, our contribution, aims at breaking down system functionality into collections of features.

While current modularization and object-oriented techniques seem sufficient to create an understandable first version of a software system, the complexity introduced by subsequent functional enhancements tends to get out of control. One reason for this is that functionality is not explicitly captured by programming language constructs, and the code implementing a single feature is often hidden at many different locations. This problem has gained special importance through the increasing popularity of evolutionary life-cycle models, which promise rapid delivery and close control over the development process by the gradual addition of features [3, 4]. Whereas in the classic waterfall model, stages such as specification, design, and coding happen one after the other for the total system, in the evolutionary model, developers first specify, design, and code a reduced kernel, which is gradually incremented toward the full functionality. Hence, every increment can be considered as an instance of software maintenance, turning most of the software development problem into a maintenance problem.

1.1 Vertical versus Horizontal Abstraction

The best weapon to combat the inherent complexity of large software systems is abstraction. Traditionally, higher-level notations have been used to abstract from implementation details. However, there are two problems with this vertical approach: First, it is difficult to verify the consistency between different levels of abstraction. Second, even if we abstract from all implementation issues, the external behavior of systems is becoming increasingly more complex so that it can no longer be easily understood.

The feature-oriented approach presents a horizontal alternative to the vertical abstraction paradigm. Rather than omitting implementation details, we investigate the omission of behavioral details, i.e. well-defined pieces of system functionality. Subsequent decomposition of these pieces leads to atomic features.

Horizontal abstraction induces an algebraic structure which gives a precise mathematical meaning to features. As a result, we can explicitly associate atomic features with software fragments and thus disable and enable features at will. This makes them traceable and manageable throughout the software life-cycle:

- During requirements analysis, our approach allows structuring and decomposing functionalities into features. All known feature interactions are also made explicit at that time.
- During design, internal system functionality is identified and managed in exactly the same way that external functionality was before.
- By mapping the functional structure into the actual source code, programmers are confined to implementing features in a clean and traceable way, which leads to understandable and maintainable code.

- For testing purposes, parts of the system functionality can be disabled, which allows testers to focus either on specific features in isolation or on feature interactions.
- Automatic evaluation of regression tests is usually difficult because the behavior of the system is not exactly the same as previously. By disabling functionality added in the meantime, the old behavior can be obtained at any time, although the internal structure of the system may have changed significantly.
- Marketing considerations sometimes require feature bundling in a way that was not anticipated during design. Instead of increasing the system complexity further to achieve the desired bundling, unforeseen functional subsets can be created easily with our approach.
- In order to train people in the use of a new product, it is often useful to start with a simpler, more understandable version of the system in which certain advanced features are disabled.
- Maintenance does not always mean *adding* functionality. Sometimes features have to be retired because they have turned out to be useless, poorly implemented, or even harmful. Our approach guarantees full reversibility, i.e. easy removal of obsolete or unwanted features.
- By using run-time feature control, a new approach to fault-tolerance becomes possible. If an error occurs within code belonging to a certain feature, only this feature needs to be turned off. The overall system may be able to continue operation at reduced functionality.

We will illustrate most of these stages through the example of an SQL database management system. We will analyze its set of features, and how this analysis has guided the design and implementation of the system. In addition, we will show how readily available tools such as cpp (the C preprocessor), sed, and grep can be used to support selective enabling and disabling of features.

1.2 Previous Work

We know of two previous attempts for obtaining better feature traceability:

Norman Wilde (University of West Florida and, like us, a member of the Software Engineering Research Center) has created the notion of "software reconnaissance" [6]. He runs test cases over an instrumented program to find out where in the program a certain feature is implemented. His reconnaissance tool tries to identify "unique code" with respect to the feature in question, i.e., program statements that are only executed when the feature is invoked.

Hart and Shilling (Georgia Tech) try to solve the problem earlier, viz. at the time a feature is being implemented [2]. They designed a syntax-directed editor that allows programmers to declare features and to link them explicitly with nodes in the program tree. Advanced operations like feature extraction (selected disabling of features) and feature instantiation (creating additional instances of generic features) are also possible.

What has not been attempted, though, is to define the *concept* of a feature. According to [2], *"it is precisely this lack of rigor which gives features much of their power. The designer is free to define as a feature any useful, easily described, slice of program functionality."*

While we agree that features may resist a *complete* formalization, we have found that a high degree of precision is desirable and achievable. Our feature-oriented methodology, which spans the entire software life-cycle, would not be possible otherwise. The formal basis of our feature theory is elaborated in [1].

1.3 Organization of this Paper

After a brief introduction to feature-oriented terminology (Section 2), we present our view of the evolutionary life-cycle in the context of feature awareness in Section 3. The following sections (4–8) give details pertaining to a number of selected life-cycle phases including requirements specification, design, implementation, testing, and maintenance of feature oriented software systems. We conclude with a discussion of our methodology and suggestions for further research. Throughout the later sections, we will illustrate the approach by applying it to our SQL example.

2 Feature-Oriented Terminology

We will not go into the mathematical details of feature theory here. However, a working knowledge of feature-oriented terminology is a prerequisite for everything that follows.

2.1 Features

Features of a system are found by imagining simplified system versions, which we call *reductions*. One can think of a reduction as a historic predecessor (which it might in fact be), or just as a preliminary version to be shown to the customer. The difference between a reduction and the full version is a *feature*.

Similarly, to determine whether some capability of our system is a feature, we try to imagine a reduction that would result from its removal. If such a reduction is possible and easier to understand than the original system, we have identified a feature.

The converse is not always true, though. One feature might logically depend on another feature, so the latter cannot be removed alone. But if the reduction becomes possible after removing some other features, we also know that we have a feature. (The full version inherits the features of all its reductions.)

For some reductions, it is not immediately obvious whether they are simpler or more obscure than the full version. This typically happens when the granularity gets too fine, and then it is better not to reduce. For coarse-grain reductions, on the other hand, the following heuristics have proven useful:

- Reductions must not obscure the main purpose of the system.
- Reductions must not obscure the main purpose of a data item.
- Irregularities, exceptions to the rule, and unmotivated restrictions increase obscurity.

Generating a certain data item x and displaying it on the user interface is a typical example of a feature. We can well imagine an identical system which does not support data item x and is therefore obviously simpler.

Displaying x, on the other hand, is *not* a feature. Although we can imagine a version where x is generated without being displayed, such a version would obscure the purpose of x, which is to appear in the user interface.

Generating x is not a feature, either. It is logically impossible to remove this functionality as long as the functionality that displays x is present. But as we saw before, the display functionality may not be removed alone. Because of this catch-22 situation, generating x and displaying x can only be removed together. This means that they form one atomic feature.

Although we are mainly interested in software here, features can also be found in other systems such as organizations or mechanical devices. A car designed to go straight all the time is obviously simpler than one that can be steered. Therefore steering is a feature. Now imagine a car where only "turning left" is removed, i.e., one can turn right, but the steering wheel blocks on the left. Such an asymmetric car is far more obscure than one that lets you turn both ways, therefore "turning left" is *not* a feature.

2.2 Functionality

The overall system functionality is composed of three parts. They are core functionality, library functionality, and feature functionality, as discussed below.

Core Functionality

The purpose of an information processing system is never that it consume input, but that it yield output. Some of the output may just be nice bells and whistles, but at least one kind of main output is the purpose of the whole system.

True system understanding implies understanding the system purpose, which implies understanding the main output and how it depends on inputs and system state. Again, input and internal processing may include bells and whistles, but at least one kind of main input and main processing is involved.

Main input, main processing, and main output together form the *core functionality*, which would typically be the kind of view given in the first chapter of a user's guide. The core functionality must be preserved by every reduction, which means (by definition) that it is free of any features.

Library Functionality

Library functionality is functionality that is not application-specific and is provided by the programming language, built-in procedures, run-time libraries, and other general-purpose tools.

Sometimes a new utility module has to be written during a project because the standard library did not provide what was needed. The functionality of such a module is considered library functionality, provided that the module is logically separated from the project and meets the quality standards necessary for later incorporation into a central library. Moreover, all maintenance on the module must happen in an upward-compatible way.

Like core functionality, library functionality must be preserved by every reduction. Therefore, library functionality is also considered completely free of features. This makes sense, after all we want to model the features of our application, not the features of our run-time library.

Feature Functionality

Since core and library functionality are completely free of features, all features are captured by the remaining feature functionality. In its entirety, feature functionality can be seen as one big composite feature, although in practice it is more useful to decompose it into individual features of a finer granularity.

2.3 Relationships between Features

For any two features, exactly one of the following relationships holds:

- They are parallel.
- They are orthogonal.
- One depends on the other.

Two features are *parallel* if there is no sequence of reductions in which both features get eventually removed, typically because they functionally overlap.

An important task in feature-oriented software engineering is to decompose the feature functionality into mutually non-parallel features. Fortunately, as demonstrated in [1], it is always possible to factor out the common functionality of two features into a new feature, which splits the original, overlapping features into a group of non-overlapping features.

Note that two features are not only parallel if they overlap, but also if each needs some functionality from the other. However, such circular dependencies are not a problem in practice because they disappear naturally as feature decomposition proceeds.

Two non-parallel features are *orthogonal* if they can be removed in any order. This means that there is some reduction sequence in which the first feature gets removed ***before*** the second feature, but there is also a reduction sequence in which the first feature gets reduced ***after*** the second feature. Intuitively, orthogonality implies that the two features can be freely combined.

The last possible case is that one feature ***depends*** on the other. We say that the feature f depends on the feature g if there is no reduction sequence in which f gets removed before g. There are two possible reasons for a dependency:

Logical dependency: The semantics of f is based on g.

Derived dependency: Only the implementation of f relies on g.

3 Feature-Oriented System Evolution

The feature-oriented life-cycle is evolutionary in nature. At the beginning, a core version is developed, which is subsequently incremented with feature functionality. Every increment is an atomic feature.

The feature-oriented life-cycle allows for maximum customer interaction. Every new feature the customer requests is inserted into a "wish list", which is transparent to the customer. At any time, the customer may put a new feature on the wish list, reorder priorities, or cancel a feature. Of course, it is in the interest of both the customer and the developer to keep the wish list as stable as possible. If the customer suddenly requests a feature that cannot be implemented with the present design, then the price for that feature will have to cover all the redesign and recoding effort.

Features from the wish list are processed according to their priority. First, the requirements engineer clarifies the semantics with the customer. If the feature is still on top of the wish list after that, the effort will be estimated and a price calculated. If the customer accepts the price and the feature is still on top of the list, the designer will devise an implementation strategy. Subsequently, it will be implemented and removed from the wish list. All these activities happen asynchronously, with the wish list acting as a pipeline.

Should the customer change his mind later, he can still cancel a feature after it has been implemented. In a feature-oriented implementation, features can be disabled and enabled at will. So if the customer changes his mind again, he can have the feature back.

Another advantage of automatic disabling is that a new version can be given to the customer every time a new feature has been implemented. Traditionally, an expensive synchronization is needed in order to obtain a stable and consistent version. All implementation and testing activity in progress must be finished, and nothing new may be started until the new version has been frozen. In a feature-oriented implementation, by contrast, features that are not ready for release can be disabled, and the customer gets one new feature and nothing else.

4 Feature-Oriented Specification

It is a highlight of the feature-oriented life-cycle that there is no major gap between specification and design. Both center around system functionality and deal with it on the level of individual features.

There is an interesting difference, though, which has to do with library functionality. By definition, library functionality is application-independent, and its only purpose is to facilitate the implementation of the required application functionality.

In the specification phase, we are only concerned with application functionality, not with how it will be provided. The identification of desirable library functionality or useful existing building blocks is a design issue. Therefore, we can restrict ourselves to core and feature functionality during specification.

4.1 Core Specification

The first step to a feature-oriented specification is to specify the core functionality, which is equivalent to specifying a *core version*. The core version should be a useful approximation of the full version. It should capture the basic idea behind the system, but leave out any additional bells and whistles.

Everything else is based on the core specification, so it should be made as small, clean, and precise as possible. Since the core version is not "messed up" with features, it is far more susceptible to a formal specification than the full version. But even a semi-formal core specification will profit greatly from the simplification achieved by feature omission.

4.2 Feature Statements

The next step in the requirements specification phase is to decompose the feature functionality, i.e. the difference between the core and the full version, into features. This happens in two directions: Starting from the full version, we can remove features to get a sequence of reductions. Starting from the core version, we can add features to get a sequence of enhancements. It is a good idea to mentally construct each of the versions and to verify that they are logically possible. Every feature must also be checked

to verify that its removal has a simplifying rather than an obscuring impact. When the two sequences meet at some version, we have completely decomposed the feature functionality into a set of mutually non-parallel features.

Whenever a feature has been identified (i.e. either something that has to be added to the core version to increase its complexity toward the full version, or something that can be removed from the full version to simplify it toward the core version), the question of atomicity arises. If the feature is quite complex but can be decomposed into simpler features by constructing intermediate reductions, then it should be further decomposed. Otherwise, decomposition halts and the feature is declared atomic.

Once the atomic features have been identified and named, their precise meaning must be defined. For each feature f the following question must be answered: "How does adding (or removing) f change the behavior of a version?" It should be easy to give a precise, informal *feature statement*. Note, however, that the feature statement must take into account the presence or absence of other features. A special case is that f cannot be added at all unless feature g is already present. This would mean that f depends on g.

4.3 Interaction Statements

Dependency is not the only reason why it may become necessary to mention other features in a feature statement. Suppose f and g are orthogonal, i.e. there are versions with f but without g and vice versa. If the functionality of both features is to apply some transformation to the same data, then the order in which these two transformations are performed may be significant. Therefore, the feature statement of f would be incomplete without specifying that f must happen, say, *before* g. Similarly, the statement of g would have to require that g happen *after* f.

The above relationship between orthogonal features is called *feature interaction*. Generally speaking, interaction between f and g means that it is impossible to deduce the exact behavior of a version in which f and g occur together from the behavior of versions in which only one of them occurs. Interactions between three or more orthogonal features are also possible.

Note that every feature interaction must clarify an ambiguity introduced by the simultaneous presence of orthogonal features. If a straight-forward combination of the features is possible, then there exists a version in which the features are all present without any interaction. Adding unsolicited interaction between the features would increase complexity, so the "interaction" is really a new feature.

Clearly, specifying an interaction in all the corresponding feature statements would introduce an awkward redundancy into the specification. Therefore, feature interactions should be expressed in *interaction statements*, separately from the feature statements. Through this convention, feature statements may only contain references to prerequisite features, not to orthogonal ones.

4.4 Example

Consider an SQL database management system that supports the commands *create table, insert tuple, select tuples, update tuple, delete tuple, drop table*. The purpose of a database is to respond to queries, so *select tuples* belongs to the core functionality. But before we can *select*, we must create tables and fill them with data, so *create table* and *insert tuple* are also part of the core functionality.

The remaining three commands, however, can be omitted without sacrificing the main system purpose. Moreover, their omission makes the system simpler. Therefore, we treat them as features: **`upd`**, **`del`**, **`drop`**.

Another way to simplify the system is to abolish *where* clauses for the commands *select, update, delete*. **`where`** is only *one* feature, because if it worked for some commands but not for others, we would have obscured this system with anomalies. (Note the feature interactions **`upd/where`** and **`del/where`**.)

The *select* command allows joins over multiple tables. What if it were limited to only one table? Again, we would get a simpler system, so **`mult`**, the ability to utilize multiple tables, is a feature.

As a final feature, we may require that all the meta-information about tables and attributes be stored in a special *schema table* so that it can be easily retrieved (**`schtab`**).

At this stage of the example, there are no logical dependencies, so the six features induce $2^6 = 64$ different versions.

5 Feature-Oriented Design

5.1 Core Design

Similarly to the specification phase, in feature-oriented design we deal first with the core functionality. As feature-orientation is orthogonal to existing paradigms, any appropriate design method can be used to design the core version.

It is important to review the core design with respect to features that are already known from the specification. While some features might fit in smoothly with the anticipated system structure, others might be impossible to implement without major restructuring. The core design should be adjusted accordingly, until every known feature is expected to be implementable in a clean way.

5.2 Library Design

Another issue to be addressed during core design is to identify all the library functionality needed to implement the core functionality. Some of it will be covered by existing libraries, the rest has to be designed as new libraries and logically separated from the application design.

The decision what to delegate to a library and what to leave inside the application is often based on a tradeoff. This is true for existing libraries as well as for new ones. On the one hand, we want to move out of the application as much functionality as possible to keep it simple. On the other hand, we lose the flexibility to make arbitrary, application-specific changes once we incorporate functionality into a library.

While the distinction between genuine application modules and new libraries written during the project seems not too important in conventional software engineering, it becomes vital in feature-oriented development. The reason is that functional enhancements to application modules are always explicitly registered as features, whereas, e.g., adding an entry point to a library module is not considered a new feature.

5.3 Feature Design

The core design is *not* updated when a new feature is designed. Like the core specification, we want to keep the core design simple, so flooding it with features is not a good idea. Moreover, changing an existing document is a tedious endeavor, and it's hard to keep track of what has been changed at what time and for what purpose.

As in feature-oriented specification, we keep feature designs textually isolated from the design of the core version and from each other. Designing a feature means developing an implementation strategy that satisfies the semantics of the feature statement. It is often possible to encapsulate most of a feature in a new implementation module, which would then be designed with some existing design notation. How the new module must be "hooked" into the rest of the system, however, is explained in plain prose (*feature attachment*).

Derived Features

The designer of a feature-oriented system must always be aware that a unique correspondence between application code and features is required, i.e., features may never share code. If it turns out during design that there would be shared code, then the common functionality must be factored out into a new feature and removed from the original features. Since the new feature was not visible at specification time, we call it a *derived feature*.

The rule that features never share code may seem a little arbitrary. But one of the problems with conventional software is that code is often shared by many features, and the way in which a code fragment contributes to each of them is never made explicit. With our "pure" fragments, on the other hand, there is no such problem because each fragment contributes to one feature and nothing else.

Let us look at an extreme example. With code sharing, it would be perfectly legal to say "all the features are implemented by all the code". Clearly, this reveals nothing about the code/feature relationship.

Derived Dependencies

Every derived feature causes *derived dependencies*, because the features for which the shared code was originally needed now depend on the new feature.

But this is not the only way a derived dependency can emerge. In many cases, the easiest way to implement a feature is to reuse the functionality of some other feature, even if there is no logical dependency between them. The features may be orthogonal in the specification, but because of the design decision to base the implementation of f on g, a reduction with f but without g becomes impossible. Hence, f depends on g.

5.4 Interaction Design

Apart from core functionality and features, we also need to design every feature interaction, i.e., to plan the incorporation of code that ensures that the interaction will happen exactly as specified in the interaction statement.

Like in the specification, interactions are designed in textual isolation from the rest of the system. If a new module is introduced for the sake of a particular interaction, then the module design becomes part of the interaction design. If existing modules have to be modified (*interaction attachment*), then this is described verbally.

It sometimes happens that an interaction statement requires no code at all to be implemented. One example are interactions like "f has to happen before g". In sequential programming, putting the code for f and g in the right order will do the job. (In concurrent programming, additional code for synchronization may be needed.) Another example are interactions that fall in place automatically as a consequence of the system structure. Because there is no explicit program logic that implements such interactions, we call them *implicit interactions*.

Whether an interaction is implicit or not is unknown at specification time, because it depends on the implementation strategy. Therefore, implicit interactions must be explained explicitly in the design.

Derived Interactions

In analogy to derived features and derived dependencies, there are also *derived interactions*. If the implementation strategy of two features causes an undesired *feature interference* as soon as both come together, then additional code is needed to correct this problem. Since the implementation strategy, and therefore the interference, cannot be known at specification time, this code is a derived interaction identified during design.

Failure to detect all the feature interferences is a major source of software errors. Although we cannot offer a recipe that guarantees complete detection, we believe that an interference is more likely to be noticed in a feature-oriented design than in a traditional design. After all, we can easily draw a matrix whose rows and columns are labeled with features. If we spend some time thinking about every field in this matrix, we should be able to explain how the two features interfere, or give an argument why they don't.

5.5 Example

Continuing our SQL example, we find that there is shared functionality between **upd** and **where**, viz. dealing with expressions over attributes. Although **where** expressions are boolean and **upd** expressions are not, we can centralize them into a single module. This means that we have a derived feature **exp**, which parses expressions and evaluates them for individual tuples. Both **upd** and **where** depend on **exp**.

Deletion of tuples is done by marking them as deleted. Since this is not so different from updating a tuple, we can reuse parts of the **upd** feature. Thus we introduce the derived dependency of **del** on **upd**.

Dropping tables is also achieved by reusing existing features. To delete the meta-information of table **X**, we simply simulate the command: **delete from schema where tab = X**. Clearly, we need three features to do so: **del, schtab, where**. This causes three more derived dependencies.

Figure 1 shows the dependency graph with the features as vertices and the dependencies as edges, the feature higher up being dependent upon the one below. The set of all dependent features is given by the transitive closure of the pair-wise dependencies.

Figure 2 shows the corresponding version graph. It is a lattice in which the vertices denote versions of the system. Versions connected by an edge differ in exactly one feature. Moving up or straight to the right in the diagram corresponds to adding features, going down or straight to the left in the graph means removing features. By combining all features, the full version of the DBMS is obtained (indicated by the top $\top$ of the lattice). By combining only some features, reduced versions can be generated, at the extreme of which we obtain the core version without any features (represented by the bottom $\perp$ of the lattice). Feature dependencies restrict the 128 possible feature subsets to 30 feasible versions. Notice that `mult` is independent from all other features.

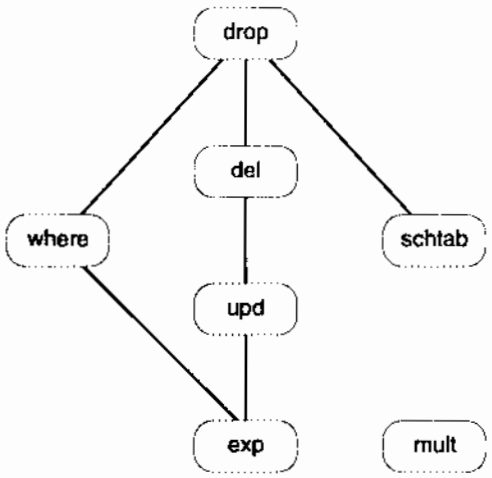


Figure 1: Dependency Graph

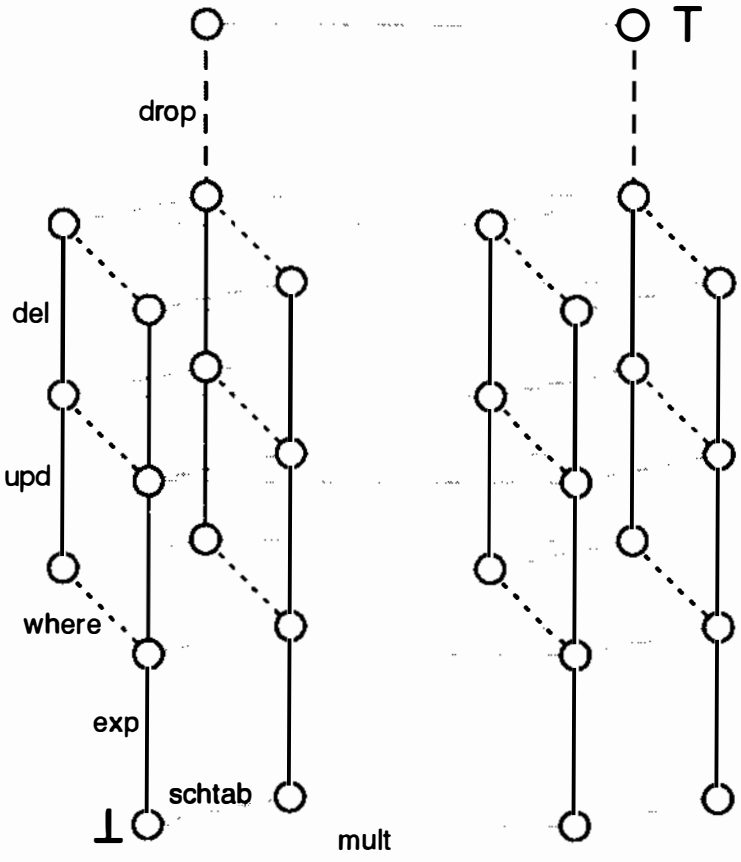


Figure 2: Version Graph

6 Feature-Oriented Coding

6.1 The Problem of Feature Isolation

The feature-oriented life-cycle postpones a complete formalization of the system as long as possible. Specification and design are largely informal, because features and feature interactions are described informally. To implement the system on a computer, however, we finally need to describe it in a formal programming language.

A key difference between informal and formal descriptions is the extent to which we can isolate features. Adding a new feature to an informal description means simply writing a statement that explains the impact of the feature on the system. In case the new feature is supposed to interact with existing features in a certain way, corresponding interaction statements must also be written.

Both the feature statement and the new interaction statements are textually isolated from the rest of the description. By simply ignoring them, we would immediately re-obtain a description of the old system, thereby disabling the new feature. And this is not just true for the new feature: Every old feature can also be disabled by ignoring the statements related to it (and to dependent features).

Unfortunately, this isolation can generally not be maintained in formal languages. Of course, we can try to encapsulate the feature's crucial logic, e.g. in a subroutine. But as long as nobody invokes this subroutine, the system behavior will stay the same. In formal descriptions, we must explicitly ***attach*** the feature to the description, e.g. by inserting invocations of the new subroutine at every appropriate location.

An important task of the design phase is to bridge the conceptual gap between isolation and attachment. While the design of every feature stands textually by itself, it explains exactly how the feature will be attached to the implementation. Hence, features are still isolated with respect to the ***structure*** of the design document, but they are already attached with respect to its ***contents***.

6.2 Marked Fragments

The basic idea behind feature-oriented coding is to mark all the code fragments that are responsible for a certain feature. This may seem an expensive overhead to the coding task, but improvements in software quality do not come for free.

The quality of the code is what ultimately determines the quality of the product. Rather than emitting as many lines of code per day as possible, programmers should be encouraged to think carefully about the precise correlation between the functionality they want and the code they write. The obligation to mark code fragments enforces this kind of awareness, which in turn raises the pride of the programmer and the quality of the code.

A sophisticated environment for code marking with a syntax-directed editor can be found in [2]. In order to demonstrate that the benefits of feature-oriented coding can be achieved with much simpler tools, we use the C preprocessor **cpp** available on every UNIX system. The following **yacc** example shows a fragment that belongs to the feature **where**:

```
select:
  SELECT { select_clear(); }
  sel_attributes FROM sel_tables

  #ifdef Fwhere
  where_clause
  #endif Fwhere

  '\n' { select_go(); }
;
```

Nesting of marked fragments is also possible, but only if a relationship between the features has been identified during design. One case is feature interaction, because we must make sure that the interaction code disappears as soon as one of the associated features disappears.

The second case is feature dependency, although strictly speaking nesting could be avoided there because the dependent feature may never be enabled without the prerequisite feature. However, it is often more natural to nest the code of the dependent feature inside the prerequisite feature. Nesting a prerequisite feature inside a dependent feature is, of course, forbidden.

6.3 Code Suppression

Although code insertion is the cleanest way to attach a feature, it is sometimes necessary to modify existing code. This can always be achieved by suppressing old code and inserting new code. Using **cpp** directives, we can write:

```
#ifdef Fxxx
<...code with xxx...>
#else  Fxxx
<...code without xxx...>
#endif Fxxx
```

However, this option should be used with extreme caution and only if there is no similarity between the two fragments. Otherwise, each new feature would cause code duplication, which would blow up the code and introduce a lot of redundancy. Therefore, suppressed code fragments must be made as small as possible and should always be annotated with a comment that explains why the suppression was unavoidable.

Most of the time, there is a way to do without suppression. Sometimes the existing code has to be slightly rearranged, e.g., by introducing a new auxiliary variable, but this is only good for the clarity of the code. In our personal experience, so far we have always been able to do without code suppression.

6.4 Version Generation

The benefits of marked fragments for code documentation and programming discipline have been mentioned above. Another advantage is the ability to generate reduced versions with a simple preprocessing step. Actually, this is the main advantage on which all the other advantages rely. The programmer is not only responsible for the correctness of the full version, but also for the correctness of every reduction. This makes coding much more challenging, and it fosters exactly the kind of alertness on which software quality depends.

For our environment, we wrote the feature preprocessor **Fpp**. It is a simple shell script that shifts **cpp** commands to the left, invokes **cpp**, and removes **cpp** directives from the output.

```
#! /bin/csh
sed 's/^[ ]*#/#/' | /usr/lib/cpp -CP `grep - Features` | grep -v '^#'
```

The macros passed to **cpp** are read from the file **Features**. This file contains one line for every feature, which is simply the **cpp** option needed to enable it. To disable a feature, we edit the **Features** file and replace the hyphen with a blank. In the following example, **drop** and **mult** are disabled.

```
-DFdel
 DFdrop
-DFexp
 DFmult
-DFschtab
-DFupd
-DFwhere
```

To incorporate feature preprocessing into the **Makefile**, we use the following rules for C, **yacc**, and **lex** files, respectively. Note how an underscore is added for the filenames of the generated version.

```
%.o: %.c ; <$*.c Fpp >$*_.c; cc -c $*_.c -o $*.o
%.c: %.y ; <$*.y Fpp >$*_.y; yacc $*_.y; mv y.tab.c $*.c
%.c: %.l ; <$*.l Fpp >$*_.l; lex $*_.l; mv lex.yy.c $*.c
```

6.5 Example

The following excerpt from the SQL system is the code executed for every tuple in a table. In the core version, its purpose is merely to support the *select* command and print the selected attributes on the screen. In addition, the features **del**, **mult**, **where**, and **upd** have been attached.

del: Skip tuples marked as deleted. Delete rather than print during *delete* commands.

mult: Only print if this is the last table specified; otherwise continue recursively with remaining tables.

where: Check predicate and skip if **FALSE**.

upd: Update rather than print during *update* commands.

```
#ifdef Fdel
if( glob_tab->tupbuf [0] != SQL_DELETED )
{
  #endif Fdel

#ifdef Fmult
if( glob_tab->link != NULL )
  select_from_tab( glob_tab->link );
else
{
  #endif Fmult

#ifdef Fwhere
if( exp_eval() == TRUE )
{
  #endif Fwhere

#ifdef Fupd
if( glob_upd_flag )
{
  #ifdef Fdel
  if( glob_del_flag )
  glob_tab->tupbuf [0] = SQL_DELETED;
  else
  #endif Fdel
  <...update tuple...>
}
else
{
  #endif Fupd
<...print selected attributes...>
  #ifdef Fupd
}
#endif Fupd
  #ifdef Fwhere
}
#endif Fwhere
  #ifdef Fmult
}
#endif Fmult
  #ifdef Fdel
}
#endif Fdel
```

7 Feature-Oriented Testing

A feature-oriented implementation contains much more information than a conventional implementation. In addition to the full version, every possible reduction is implied by the implementation and can be built automatically by feature preprocessing. Feature-oriented testing uses this additional information in order to obtain more significant test results.

7.1 Feature-Oriented Correctness

For a feature-oriented implementation to be correct, not only the full version must be correct, but also every reduction. The correctness of a reduction can be tested by comparing its behavior with the corresponding reduced specification. If the full version seems correct but some reduction behaves incorrectly, then this indicates that the relationship between code and functionality has not been completely understood by the programmer.

This means not only that the functionality is incorrectly documented in the source files, but also indicates an increased probability that there are other problems with the code. It may even be that there is an error in the full version that would have never been found by testing the full version alone.

Another advantage of feature-oriented testing is that the testing team can become familiar with the system functionality gradually. They can start testing the core version, and if they don't find any errors, they can add a feature and test it against the specification. To test a feature interaction, they would activate all the features that participate in the interaction. Of course, the correctness of extremely reduced versions is not a *sufficient* condition for overall correctness. But it is a *necessary* condition, and it allows the testers to concentrate on one feature at a time.

7.2 Extent of Testing

For very big systems, it may be expensive to test every single version. On the other hand, the more resources have been allotted to testing, the more versions can be tested. The most profitable versions are probably the ones close to the full version and close to the core version.

What should be done for every version, though, is to see if it compiles. It is straightforward to write a tool that reads all the features and dependencies, enumerates the feasible feature subsets, and builds all versions. Even though this is only a *necessary* condition for correctness, problems such as misuses of constants, record fields, variables, or routines outside the intended feature are all detected at compile time. Similarly, if the compiler reports unused variables or record fields, then the programmer might have forgotten to put them in a properly marked fragment.

If automated testing is desired, we could even go a step further and write "feature-oriented" test cases with marked fragments. An acceptance procedure would also be written with marked fragments, so that a complete automatic test could be built and run for every version.

8 Feature-Oriented Maintenance

The goal of feature-oriented maintenance is to transform one consistent feature-oriented implementation into another consistent feature-oriented implementation. It cannot be overemphasized that a *transformation* and a *feature addition* are different activities. Adding a feature always means a transformation, but not every transformation is due to a feature. Marked fragments are only used for features, not for other transformations.

There are four typical kinds of transformations:

- functional enhancement
- functional change
- error correction
- internal restructuring

8.1 Functional Enhancement

Functional enhancement is the most natural kind of transformation in a feature-oriented implementation. However, it is important to break down enhancements into features of reasonable granularity, and to mark new code fragments according to the feature to which they belong.

Ideally, a feature can be implemented by mere insertion of new code (and maybe code suppression). In this case, the feature-oriented correctness of the reduced version comes for free, because as soon as we disable the new code fragments, we have exactly the system as it was before.

If slight rearrangements have to be made in order to implement the feature in a clean way, then some regression testing is required. For regression testing, the new feature is disabled, and the system is expected to behave exactly like it did before the transformation. Automatic testing can be of great help here.

In the worst case, the enhancement is impossible without major modifications to the internal system structure. In this case, a separate restructuring transformation must be applied first, as described below.

8.2 Functional Change

A change to the system functionality does not always increase complexity. Requirements include many choices the customer might as well have made differently, and if such a choice turns out to be detrimental, it will be replaced by a different one, typically without a change in complexity.

A functional change might even reduce complexity, typically if the customer finds out that a certain anomaly or restriction that was put in the initial specification was not such a good idea after all. A special case occurs when an entire feature is not wanted any more. Then there is no need to touch the code, because the feature can be disabled. Notice, however, that every dependent feature will be disabled, too.

In general, changes to existing parts of the specification, the design, and the code will have to be made in order to realize a functional change. A good rule of thumb is that everything should look afterwards as though no change had ever been made (except for a change log in the source). We don't want to bother future maintenance engineers with facts that are no longer current. Their "historical" understanding of the system should be restricted to reduced versions of the current implementation, even if this is not the true history. (The idea to "fake" history stems from [5].)

8.3 Error Correction

Error corrections are handled much like functional changes. Depending on the phase in which the error occurred, we must update the specification, the design, or just the code. Under no circumstances should a correction be treated as a feature, e.g. by marking the fragments that correct the error. A proper entry into the change log is, of course, indicated, but we will never "reduce" to the erroneous version unless it is a simplification.

If the erroneous version *is* a simplification, however, then "correcting" the error is really adding a new feature. Therefore it should be treated as a feature to begin with.

8.4 Internal Restructuring

The last kind of transformation, internal restructuring, is supposed to have no impact on the system behavior. (The only exception is that error corrections are allowed if the errors disappear as a side-effect of the restructuring.)

The reason for restructuring is to facilitate some functional change or enhancement that would otherwise be awkward to implement. Ideally, the initial design anticipates many future changes, but features that are requested many years later are often impossible to predict.

In traditional maintenance, restructuring usually happens "on the fly", together with major modifications of the system behavior. With this approach, it is anything but easy to verify whether both the restructuring and the functional changes have been done right. Automatic regression testing against old test data tends to be difficult because of the changed system behavior.

A cleaner approach is to separate structural and functional changes, starting with the necessary restructuring. Since the claim is that the system behavior has not changed as a result of restructuring, regression tests with old test data are possible in a trivial way. A correctly restructured version with the old behavior can then be used as the basis for further functional enhancements.

The separation of structural and functional changes is not limited to feature-oriented systems but also works for conventional implementations. However, feature orientation greatly amplifies its benefits in two ways: First, a feature-oriented regression test over all the reduced versions is much more significant than a regression test of the full version alone. Second, the regression test is not confined to the moment immediately after restructuring, but is still possible after many new features have been added. By disabling all the new features, we can easily test whether the major restructuring plus all the subsequent minor rearrangements have preserved the original behavior.

9 Discussion

9.1 Summary

We have outlined a feature-oriented software life-cycle and illustrated it with an SQL database management system. The system functionality must be partitioned into library functionality, core functionality, and feature functionality, and the feature functionality decomposed into atomic features. Every code fragment that does not implement core or library functionality must be explicitly associated with a feature. Thus, software quality is increased since features become traceable throughout the software life-cycle.

9.2 Evaluation

We have extended the feature-oriented approach of [2] from a documentation and maintenance aid to a life-cycle paradigm, from a syntax-directed editor to language-independent text files, and from vague ideas of what features are to precise and workable definitions.

The experience with the SQL implementation was very encouraging. Features were identified without difficulties and implemented one at a time, which facilitated coding as well as code marking. The sample excerpt with the four nested `if` statements shows how marked fragments keep the code organized even after heavy feature attachment.

9.3 Outlook

It seems that the full potential of feature-orientation has yet to be tapped. The gains in software understandability, testability, and maintainability look very promising, even if only simple tools are available and features are only controlled at compile time.

More experience with larger systems will lead the way to more powerful tools, which could, for instance, keep track of feature dependencies and interactions in order to perform consistency checks on the implementation. To cope with large numbers of atomic features, support of hierarchical structuring will also become necessary.

Although marked fragments are meant to make the code more transparent, excessive marking can easily reduce readability. Of course, there is no royal road to understanding complex software, but we do believe that there are better ways of presenting marked fragments to the human eye. For instance, feature-oriented editors could enclose code fragments in boxes, or distinguish features by colors. It would also be nice if the user could control the visibility of features with simple editor commands, thus walking through the version lattice and seeing only what he wants to see at any moment.

As we gain more experience with feature-orientation, we expect it to feed back into the design of programming languages. Current restrictions occasionally cause difficulties in code marking, e.g., because no comma may follow the last item of a comma-separated list.

The life-cycle we have sketched out is biased toward informal specification and design because it seemed most natural to describe the impact of every feature in prose. However, it would be interesting to use feature orientation in a more rigorous environment as well. One possibility is to mark fragments of

a formal specification according to the features they stand for. This would make the formal specification more transparent, and the semantics of every feature would be defined formally.

Another interesting question is whether the notion of "simplification", on which all our definitions rest, can be made more precise. Unfortunately, it seems that the ability to simplify in a meaningful way is inseparably linked to human reasoning, which is hard to capture precisely.

Acknowledgements

Valuable feedback and suggestions came from anonymous reviewers and Margie Davis, Shannon Nelson, and Barbara Siefken.

The first author was supported by a *Fulbright* scholarship, a scholarship from the *Austrian Ministry of Science and Research*, and a scholarship from the *Austrian Chamber of Commerce*.

References

[1] H. Egghart and E. Knapp. A feature-oriented approach to high-quality software. In *Proceedings of the Fourth International Conference on Software Quality*, McLean, VA, 1994. To appear.

[2] C. F. Hart and J. J. Shilling. An environment for documenting software features. *ACM Sigsoft*, 15(6):120–132, Dec. 1990.

[3] D. Hough. Rapid delivery: An evolutionary approach for application development. *IBM Systems Journal*, 32(3):397–419, 1993.

[4] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–137, Mar. 1979.

[5] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, Feb. 1986.

[6] N. Wilde, J. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proceedings IEEE Conference on Software Maintenance*, pages 200–205, Orlando, Nov. 1992.

Using Usability Inspections To Find Usability Problems Early in the Lifecycle

Rosemary Marchetti
Hewlett-Packard Company
1501 Page Mill Road, MS: 5MS
Palo Alto, CA 94304

Phone: 415/857-5688
e-mail: rosemary@ce.hp.com

Abstract

Usability Inspections give development teams another option in creating usable software and in finding usability problems early enough to fix them. This paper focuses on enabling the reader to conduct usability inspections on their software.

Keywords: Usability, Inspections, Software Evaluation, Review

Audience: This paper will be aimed at a technical audience. Specifically, people who are responsible for determining what evaluation techniques to use to evaluate their product and people responsible for creating usable products.

Biographical Sketch: Following graduation from New Mexico State University with a Bachelor of Science in Computer Science, Rosemary Marchetti began her career at IBM's Santa Teresa Lab. After joining Hewlett-Packard she worked with teams company-wide to improve their software inspection techniques. She is currently working with Hewlett-Packard teams to identify and build the competencies they need to meet their business goals. Rosemary worked with human factors experts from Hewlett-Packard to create and document this usability inspections methodology.

Contents

1. What Are Usability Inspections & What Can They Accomplish?

1.1 Situation at Hewlett-Packard

Hewlett-Packard development teams develop software for computers, printers, medical equipment, chemical analysis equipment, and electronic test and measurement equipment. We found that these development teams needed a way to evaluate their products for usability early in the lifecycle. Some development teams were successfully using user-centered design techniques to develop and evaluate products for usability. The usability testing these teams did found usability defects. Frequently these defects were found too late to fix. Many teams were also doing Fagan/Gilb style inspections. These were finding defects early but were not finding usability defects. Additionally, customer surveys showed that for a product to be successful, it had to be easy to use.

Requirements: We wanted techniques that could help us get usable products to customers in a timely fashion. Our requirements were to have a method that:

- Could find usability problems early enough to fix and prevent systematic problems. These problems include things that are built into the system that influence product interface and design decisions that are too costly to fix once they are implemented.
- The development teams could use to evaluate the product based on the users and the tasks the product was created for.
- Would help the developers to look at the product from a user perspective so they would inject less usability problems.
- Would minimize the expense associated with bringing in actual users for usability tests.

Our solution: To create a method that would meet our requirements, the inspection experts, the usability experts, and the developers combined Fagan/Gilb style inspections with user-centered design and usability testing techniques. The resulting technique helps product developers find usability defects early and hence create a more usable product. The technique also teaches developers to create the product from a users perspective.

1.2 Results of Using Usability Inspections

We have used usability inspections in many divisions inside Hewlett-Packard. Conservatively calculated, we get a *12 to 1 return on the investment* in inspections. The return on investment is calculated comparing the cost of defects that go out the door compared to the cost to find and fix the defects using usability inspections.

1.3 Usability Inspections: An Example

Imagine this situation: We are creating an operating system. Specifically we are working on the system administration software. We have a specification on paper with a representation of the user interface. We want to know if this specification will meet user needs and if the users will be able to use the interface as we've defined it. We decide we need a user evaluation of the specification. We have identified who the users are and what tasks they want to accomplish using the product. We then determine it is too expensive to bring in external users to evaluate the usability of the product. We also determine that if we distribute the specification for review among the engineers like we usually do, we won't know if the product will help the users do the tasks they want to do. So we decide to do a usability inspection.

We want the inspection to include people from inside the company. When we come together for the inspection, we want those people to be able to say "I am a novice user. As a novice user, I have experience or education using the system but I haven't administered it. As a novice user, I am going to try to get the verify the system is up and running. Now, let me try to do that and see if I can do it." We identify people in the organization who can help us walk through the tasks and answer the question "Can the users we are aiming this product at perform the tasks they want to perform using our product?" We put together a packet of information including the specification, a description of the users, and a description of the tasks the users will perform.

The inspectors take this information and first individually, then collectively in the inspection meeting, put on their "user hat" and try to perform the task we have identified. As they perform the task, they ask themselves questions like: "How would the user know to do that? For example, would the user know that when we say 'verify' that we mean 'check if it worked'? Does the user know that 'point' means 'click with the left mouse button'? As they ask these questions and identify places where the user wouldn't be able to perform the task easily, they keep a log of the things the user might not be able to do. This list is the usability concern log. If ideas come up on ways to address the issues, they might jot those down next to the concern. They also keep a detailed log of the sub-tasks the user would go through to perform the task. As they complete the task, they ask "Did we get the task complete correctly?" Finally, they complete the task satisfactorily and end the inspection meeting.

Appropriate members of the product development team take the usability concerns, decide which are truly problems, decide how to fix the problems and update the product. The people writing the documentation for the product take the description of sub-tasks and use it as they create the user documentation.

After the product is updated, members of the team ask: Did we address the issues without adding new ones? Are we ready to develop the product from this specification or do we need to re-inspect it? Depending on the situation, they move forward with development or reinspect the updated specification.

This is one view of a usability inspection. Now, let's look at this scenario in detail.

2. Usability Inspections Overview

2.1 What is a Usability Inspection?

At Hewlett-Packard, we've defined **usability** to be the degree to which intended users are:

- able to perform the tasks the product is intended to support in the intended environment;
- satisfied by the procedures they must follow and the resultant output; and
- protected from the consequences of their actions.

A usability inspection is a process we use to evaluate usability early in the lifecycle. It is a structured review done by the product designer(s) and peers of the product designer. Generally real users don't participate. The inspectors evaluate the users' ability to perform a task using a new or modified product. They look for usability defects: characteristics of a product that makes it difficult or unpleasant for users to accomplish tasks supported by the product. Once these defects are identified, the product is updated.

2.2 When Do Usability Inspections?

We conduct usability inspections as early in the lifecycle as possible. How early can we do the inspection? In a usability inspection we are answering the question "Can the target users conduct the tasks this product was intended to help them accomplish?" To answer this question, we need to have a description of the user, the tasks, and the product available. Generally, as soon as a specification-type document is available, we conduct the inspection.

Traditional Fagan/Gilb style inspections are also done the design phase. We've found these inspections help us find where the product description or design is technically incomplete, incorrect, unclear, or where it is inconsistent with related documents. They haven't helped us determine if the users will be able to use the product as they expect.

Usability testing is done during build and test phases. Usability testing is performed on a running version of the product and has actual users trying to perform the tasks. We are able to observe where the user has difficulty performing the tasks. Because usability testing is so expensive, we recommend a usability inspection before conducting a usability test.

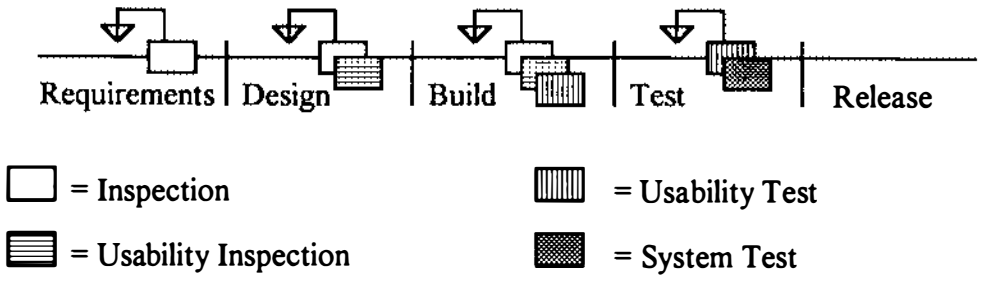


Figure 1: *Usability inspections compared to other evaluation techniques*

2.3 Who Inspects the Product?

The inspection team is a team of people from within the organization. An ideal team for a product interface includes:

Participant	*Find usability concerns based on:*
Developer(s)	Familiarity with the product and users
Marketing Representative	In depth knowledge of who the users are and what they want from the product
Support	Experience from supporting similar products
Documentation	Experience representing this product and similar products in a task-oriented manner in the documentation
Human Factors	Experience with user-centered design and usability testing.

When available, an actual user from inside or outside the company can also participate. The user is nice to have but not required.

2.4 What Steps Does the Team Go Through?

The process steps in a usability inspection are similar to Fagan/Gilb inspections. The team goes through planning, kickoff, preparation, meeting, rework, and follow-up. For detail on the steps, see section 4.1 - "What Steps Does the Team Go Through?"

3. Finding Usability Defects

3.1 How Do Inspectors Find Usability Defects?

In a traditional Fagan/Gilb-style inspection, we give inspectors a document to inspect, such as a specification. We may also include supporting or related documents, standards, and checklists. The inspectors (a group of people on or close to the development team) compare the specification to the information in the supporting documents (such as an Investigation Report) and answer the question: "Will this product meet the user needs?" If appropriate, they check for compliance with the standard or look for defects using a checklist.

We have found that for product engineers to identify usability defects in the product, it is helpful to look at the document (or product) in a different way. The engineers, acting as inspectors, visualize themselves as the user of the product. They then use the product (or a description of a product) to try to perform the tasks a user would perform. If the inspector can't perform the task, they have found a usability defect.

3.2 What Information Is Given to Inspectors?

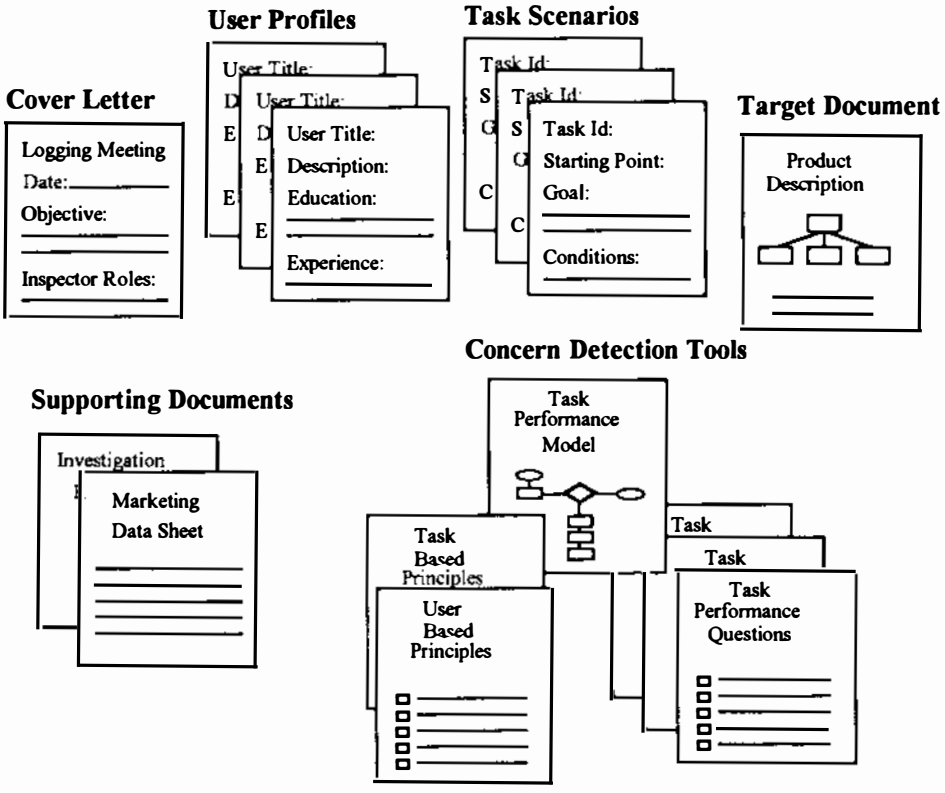


Figure 2: *The Usability Inspection Packet*

The Packet: In order for inspectors to find usability defects, we give them a packet of information. This packet contains information that helps them look at the product from the users' perspective. This packet includes user information, product information, and usability information. The user information describes the a profile of the users and the tasks those users will perform.

The user information given to inspectors comes in the form of user profiles and task scenarios. The user profile describes the users who will be using the product and the task scenarios describe the tasks those users will perform. The user information helps the inspectors visualize the user who will be using their product and helps them understand what kinds of tasks the users will perform with the product.

3.2.1 User Profiles

User profiles contain a description of the target user. Generally the user profiles we create include a description of background, experience level, and skill set of the user. A product will have one or more user profiles. An example of a user profile is:

Profile Label:	High End System Administrator
Description:	Responsible for all installation and access to the network.
Education:	Computer Science Degree or similar background
Experience:	Minimum of 2 years as a System Administrator with HP-UX. Includes experience with networking and with windows. Uses HP-UX every day.

3.2.2 Task Scenarios

Task scenarios contain a description of the tasks the users expect to perform with the product written from the user's perspective. The task scenarios include the user's end goal, the starting point, special conditions that may occur, and the ending point of the task. An example of a task scenario is:

Goal:	Verify that the computer is set up correctly and is ready for general use.
Starting Point:	User has taken the Model 807 out of the box, cabled all hardware components and performed software configuration.
Condition:	The LAN card is not working.
End Point:	Computer is ready to use.

Note that the task scenario does not include a sequence of sub tasks that the user will perform in getting from the starting point to the ending point because the user would not know this information.

3.2.3 Product Description

The packet also includes some type of product description. The product description describes or shows what the product will look like to the users and how it will behave. We have used paper prototypes, storyboards, example screens with descriptions of how to use them, and online prototypes to represent the product. Which of these is used depends on how the development team documented the product.

3.2.4 Usability Information and Tools

Because the inspectors are not human factors experts and they are not always real users of the product, we also give them some basic usability information and tools. These tools help them identify usability concerns. The tools we use most often are *usability principles* and a *task performance model*. These tools enhance the inspectors knowledge about usability and help them be more objective as they look for usability concerns.

Usability principles are principles we have learned we must follow so users will be able to use our product. There are many different usability principles. We have selected the following set specifically for usability inspections on software products. When we follow these principles we have a better chance of users being able to use our product.

User-Based Principles

1. Provide a mental model of operation that capitalizes on pre-established mental models.
2. Speak the users' language.
3. Use a simple, natural dialog.
4. Be consistent:
5. Provide an intuitive visual layout.
6. Make functionality obvious and accessible.
7. Provide good help.
8. Provide for multiple skill and use levels.
9. Allow for user customization.
10. Put the user in control of the task and interface.
11. Provide feedback -- minimize uncertainty.
12. Minimize memory load and mental processing.
13. Design for user error.
14. Protect the user from the details of the implementation.

Task-Based Principles

1. Minimize actions required to accomplish tasks.
2. Support the natural task flow.
3. Don't ask the users to do what a machine can do better.
4. Provide defaults.
5. Provide shortcuts.
6. Provide undo, redo, pause, and resume.
7. Make it very difficult to destroy work.
8. Make sure errors can be corrected with very little rework.
9. Don't waste the users' time.
10. Provide clearly marked exits.

The ***task performance model*** explains what users will look for and think of as they choose what actions to take to accomplish their goals. The task performance model we are using shows a simplified process that users go through as they decide what action to take to do a task. The model enables the inspectors to be even more objective about what the user will see, think and do. Based on the model and the questions the model incites s you to ask, the inspectors have more tools to answer: "I know how to do that, but would the user know how?"

After observing human factors experts as they perform usability evaluations, it is this type of model and questions that enables them to do an excellent job evaluating a product for usability. The model we are currently using is a four step model shown in the flowchart below.

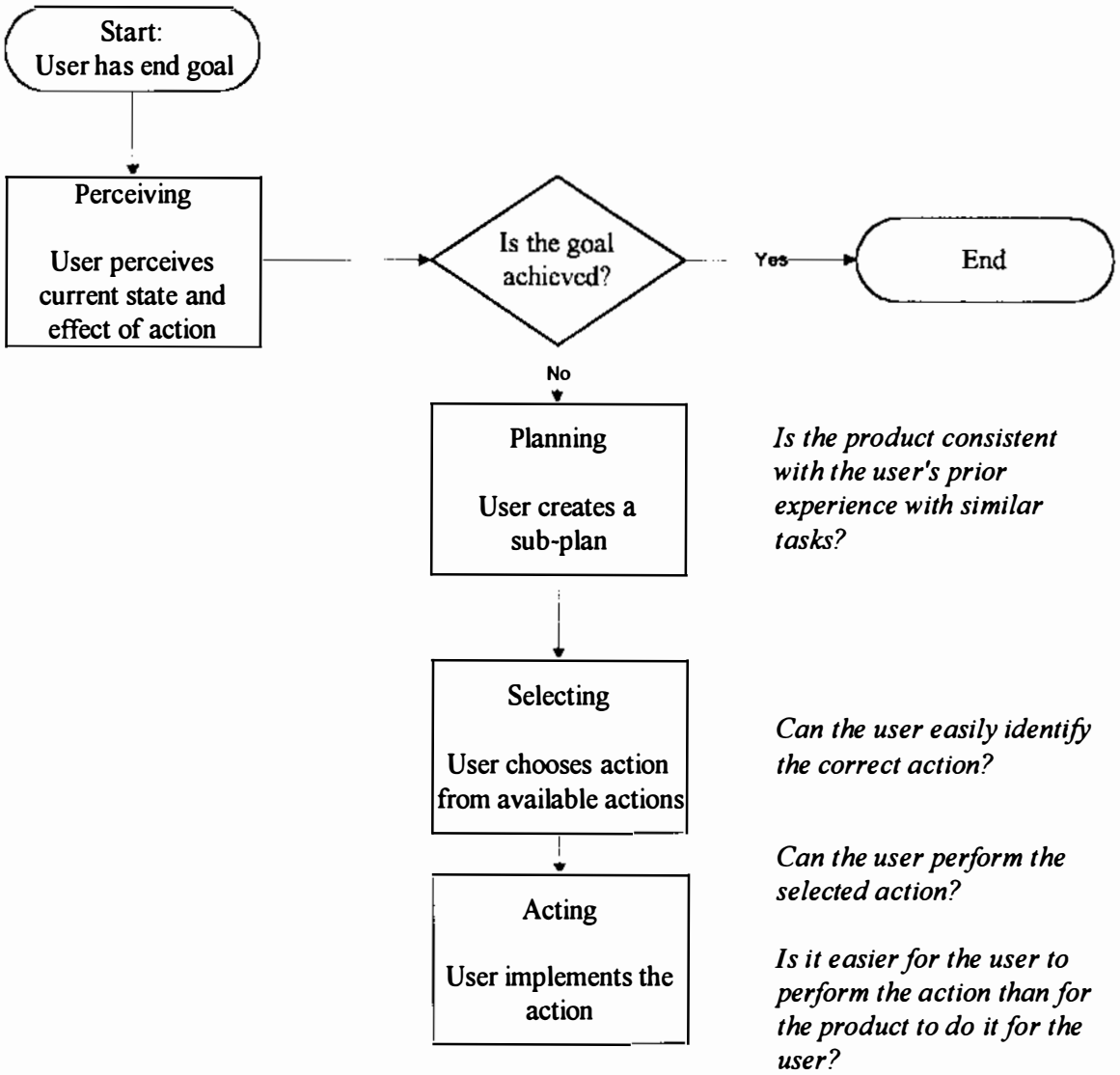


Figure 3: *Task Performance Model*

Step 1: Perceiving
The first step in performing a task is when the user defines their end goal. The user then looks at the product and **perceives** a current state. The information the user gets tells the user where he is and he gets an idea what he can do next. The inspector can ask questions about what the user will perceive in this product. For example, the inspector can ask:

- Can the user find the necessary information?
- Can the user determine if he or she has reached the end goal?
- Can the user easily identify the following:
  Current product state?
  Available actions?
  If the action chosen was correct?
  If the action chosen was performed?
  An error state?
  Appropriate error recovery action?

Step 2: Planning
Second, based on what the user perceives, he determines whether he has achieved his goal. If he has, he ends the task. If he hasn't, he creates a plan or direction defining what he could do next. An example of this would be "I am going to create a new document". Again, knowing the user might have usability problems with this step, the inspector might ask:

- Can the users apply plans or models based on prior experience with similar tasks?
- Can the users phrase the goal correctly in terms of the product?
- Do the users have sufficient support for selecting or creating the right plan?
- Is it easy for the user to map their goal into a product goal?

Step 3: Selecting
Once the user has made a plan of which direction to go, he chooses an action from the available actions. This step represents the mental action of selection. An example of this is "I am going to look inside the File menu item and see if I can create a new document from there." Questions an inspector can ask at this point include:

- Are the available actions easily identified?
- Are there competing actions that the user could select in error?
- Are there sufficient cues provided to help the user select the right action?

Step 4: Acting
Once the user has chosen an action, he will physically perform the action. The user might say "I am going to press the right mouse button while holding it over the File menu item and see what happens." Questions that help inspectors find potential usability problems with taking action include:

- Can the user physically perform the action they selected?
- Is the user protected from accidental actions?
- Is this an action that the user should do or should the product do it?
- Is the time (including waiting time) required to perform the action reasonable?

3.3 How Do Inspectors Use the Information to Find Usability Defects?

This process an inspector goes through to find a defect is represented in the following flowchart.

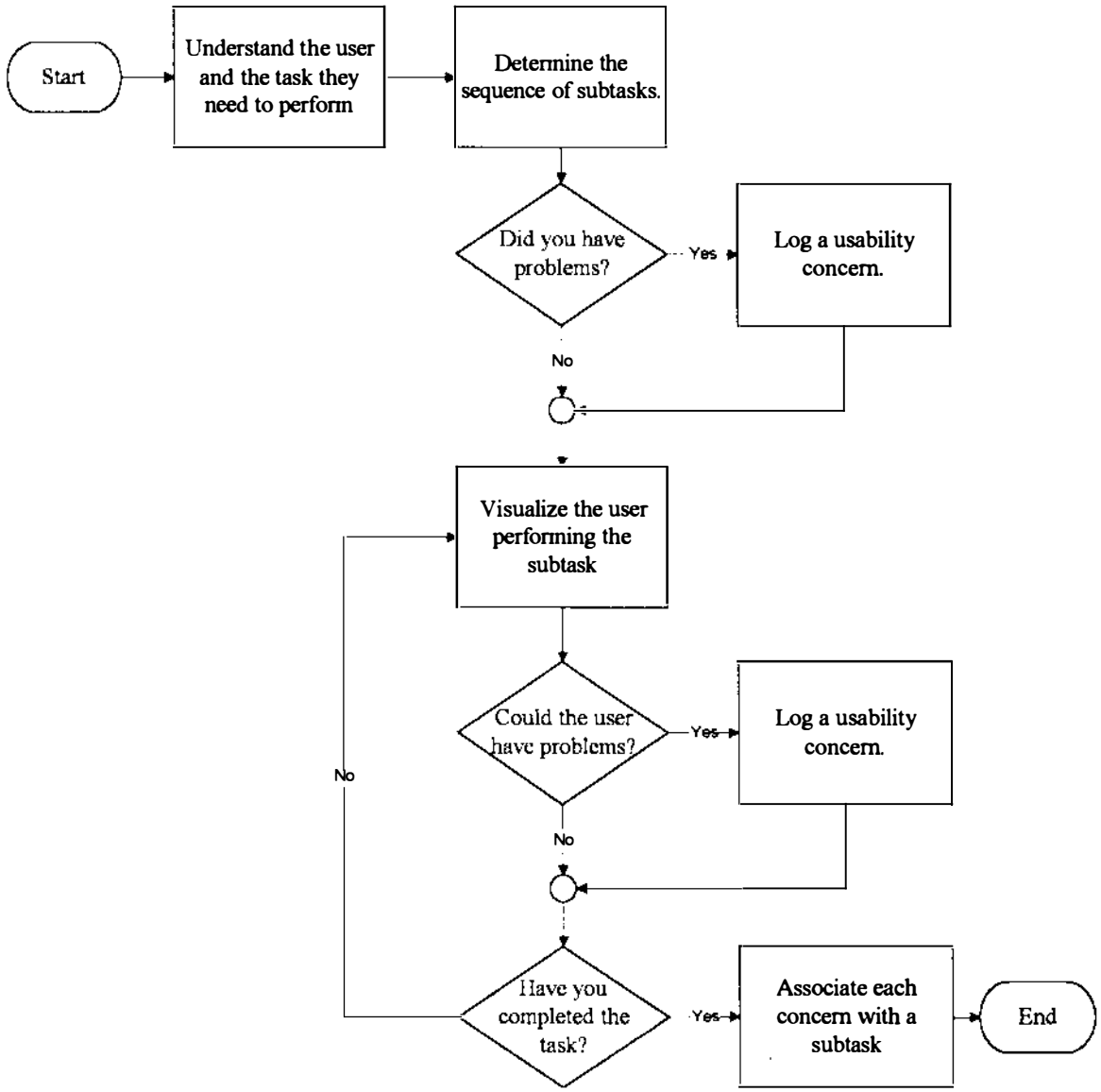


Figure 4: *Process for Finding Usability Defects*

In order to find usability defects, the inspectors first understand the user and the tasks the users need to perform. They use the user profiles and task scenarios to visualize the user performing the task. They then determine a sequence of subtasks the user could perform to accomplish the task. The inspector uses the "product description" to identify this sequence of subtasks. If the inspector has problems developing the sequence, they log a usability concern.

The inspector then tries to perform the subtasks (again from the perspective of the user represented in the user profile). When they have problems executing the subtasks, they log a usability concern.

Finally, considering the usability principles and task performance model they ask "Could the user have other problems?" We recommend that the inspectors refresh their memory on usability principles as they inspect. If they see an area where the product doesn't follow usability principles they can log a defect. Also, if they see something that looks suspicious, they can see if it violates a usability principle.

Another way the inspector can determine if the user could have problems is to ask questions about how the user would think about the problem. A common question we ask is "How would the user know to do that?" Questions like this are based on the task performance model. As the inspectors perform the task, they ask these questions to themselves. Additionally, during the inspection meeting, the Human Factors Engineer and the facilitator ask the questions as part of their facilitation of the meeting.

4. Team Aspects of Usability Inspections

4.1 What steps does the team go through to do a usability inspection?

The process steps in a usability inspection are similar to Fagan/Gilb inspections. The team goes through planning, kickoff, preparation, meeting, rework, and follow-up.

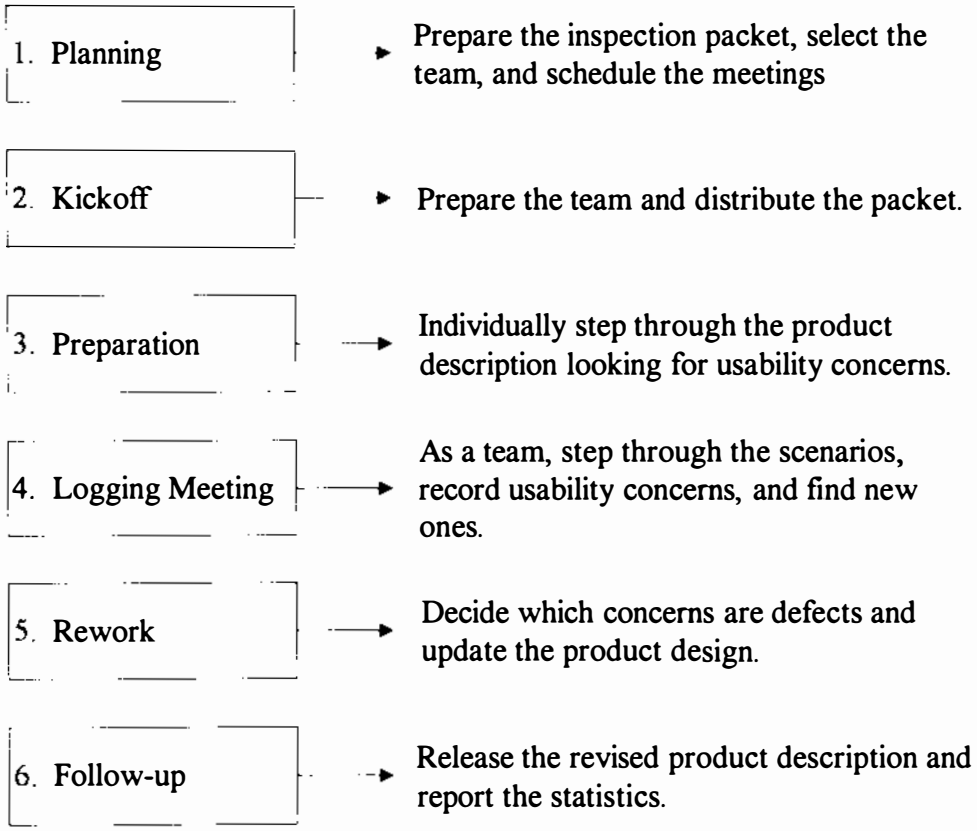


Figure 5: *The Usability Inspection Process*

One way of looking at the steps of an inspection is that these are the steps needed to have a good meeting. In the following explanation, the *italicized portion of the explanation is what the facilitator would do for any meeting*. The normal (non-italicized) portion contains the specifics for an inspection.

4.1.1 Planning

To have a good meeting, you plan what you want out of the meeting, who needs to be there so you can get what you want, what they need to do to prepare for the meeting, and what you need to give them so they can prepare for the meeting.

For an inspection, the planning step includes:

- identifying the objectives of the inspection,
- creating the packet (if everything needed for the packet is not complete, the product designers, working with a usability expert or a human factors expert, create the necessary information),
- defining which tasks you will try to do,
- selecting the team members (we usually have about 8 people on a team),
- scheduling the meetings (our meetings are between 2 and 4 hours),
- deciding what "done with the inspection" means.

4.1.2 Kickoff

Once you are done with planning, you tell the team members what you want from them and you give them what they need prepare for the meeting.

In an inspection, this is the kickoff step. This can be done via a meeting or via a memo. If the team is new to usability inspections, we usually hold a half hour to an hour meeting. In the meeting we set expectations for how this kind of inspection will work and define what we need from each inspector.

4.1.3 Preparation

Once you have given the participants the information they need to prepare for the meeting, the participants prepare for the meeting.

Inspectors prepare as per your instructions in the kickoff. These preparation instructions are usually one of two flavors. Inspectors can either become familiar with the product, the tasks, and the user profiles and come to the meeting to walk through the tasks *or* they can actually try to do the tasks, find usability concerns, and bring these concerns into the meeting. Which option you choose depends on the ability of the team to walk through the tasks offline and the time constraints of the team members. Frequently it is difficult for the team members to get together for long meetings so the preparation is done prior to the meeting.

If you haven't done it yet, while the participants are preparing, you decide how you should run the meeting so you can meet the team's expectations for the meeting.

For a usability inspection, you decide or recommend:

What roles should the team members play so the meeting goes well?
How will you lead the team through the tasks and how will you log the concerns?
How much discussion you will have on solutions versus on identifying concerns?
How will you know if the tasks are taken to a satisfactory completion?

4.1.4 Logging Meeting

After the team members have prepared, you then hold the meeting facilitating it for effectiveness & efficiency, using other team members as appropriate to help you achieve the goals, and keeping the notes you need to do the work after the meeting.

In the inspection meeting you have a facilitator. It's easier if the facilitator is not the human factors engineer or the designer of the product. They need to focus on the content of the meeting while someone else watches meeting process.

We usually have two flip charts or printing whiteboards. We use one to log the usability concerns and one to document the task steps. The facilitator prompts the inspectors with questions like: "First we are going to see if user x can do task y. What would the user do first?" The moderator writes the subtask (steps) on one flip chart and uses this log as a tool to keep the meeting focused. The scribe logs the usability defects on the other flip chart.

We usually focus on logging concerns during the meeting. If a solution comes up, we log it and move on. We focus on solutions during the rework step. As we walk through the task, the facilitator or designer is responsible for determining if we are successfully completing the task.

4.1.5 Rework

After the meeting, the appropriate team members do action items as identified in the meeting.

After the Logging Meeting, the appropriate people (generally the designers, the human factors engineer) decide which concerns are really defects and decide how to address the usability concerns. The appropriate people update the product description to fix the usability defects. The person writing documentation for the product uses the task steps as they write their task-oriented documentation.

4.1.6 Follow-up

After the action items are complete, the facilitator or person responsible for the results of the meeting checks back on the scheduled completion date to verify the actions have been done satisfactorily.

Once the fixes are complete, the facilitator follows up with the designers. They decide whether the inspection meets the "done-ness criteria" identified in planning. Usually they meet the criteria and move on. Sometimes they decide they need to re-inspect the product/tasks to verify that the fixes actually fixed the defects and didn't add new ones. These decisions are based on how severe the concerns were and how much of the product changed. If they re-inspect, they start again with planning. If they don't they have completed the inspection. The facilitator documents the results in a memo and the team moves forward with development. The facilitator generally keeps information on number of defects found and the time invested.

4.2 Scheduling Inspections

One of the questions we get asked is "how much time does a usability inspection take?" The following chart contains an estimate of the time in hours and the elapsed time for conducting an inspection. The amount you can accomplish in one meeting will depend on the number of steps in your task. The number of weeks you take to complete an inspection will depend on your schedules and on the number of user/task combinations you plan to inspect.

Usability Inspections	*Week 1*	*Week 2*	*Week 3*	*Week 4*	*Week 5*	*Week 6*
Planning	——					
Kickoff Meeting		Δ				
Preparation		——				
Logging Meeting			Δ			
Rework			——	——		
Follow-up						——
Hours Required		Δ	Δ			
Moderator	12	1 4	4	8		4
Owner	4	1 4	4	4**		2
Inspector *	0	1 4	4	4		1

* Inspector Hours = hours per inspector
** Includes only the time required for discussion with the team.
Does not include time to fix the problems found.

4.3 What Makes Usability Inspections Work?

The success of usability inspections is based on the ability of the inspectors to think like the user, understand the tasks from the users perspective, internalize usability principles, and understand, at a high level, the process that users go through in performing a task.

Additionally, we've found that there are several keys that enable *teams* to have successful usability inspections.

- The inspection environment is constructive.
- The inspection has a clearly stated objective.
- The task scenarios and user profiles are sufficiently detailed.
- Subjective and objective inspection process data is gathered and analyzed to ensure the inspections are effective and efficient.

When we have been able to do this, we have been able to utilize this process to help engineers make better decisions early in the lifecycle about the product.

5. Starting Inspections and Making Them Work

5.1 How Do Teams Get Started?

Inside HP we've found that any technique or method that really gets used fills a business need for the people using the technique. The technique can't merely be "good for you". With this in mind, any time we consider doing usability inspections with a product development team, we first identify the business goals of the team. Once we understand the goals, we give the team (including appropriate management and engineers) a menu of options they can select from to help meet their business goals.

When a team selects usability inspections, we consult with the team as they:

- Define the objective of doing usability inspections. This helps verify that an inspection is the right technique and helps focus the inspection.
- Integrate usability inspections with scheduled work. This helps usability inspections become a part of getting the product developed, not an extra thing designers are required to do.
- Define the specifics of the process based on what will work for them in their environment. We are not dogmatic about details. We have found that if we provide options to the team, they choose options that work best for them.
- Clearly define evidences that will help the team know if the inspections are working or not. As the team does usability inspections, they look for these evidences. When the evidences show inspections are not working, the team defines what might work better to meet business goals.

In our experience, when the team has:

- identified clear product level goals, usability goals, and goals for conducting usability inspections,
- scheduled the work associated with usability inspections into product and individual schedules,
- defined the specifics of the process in a way that will work for them, and
- defined evidences of success and are checking for those evidences,

there isn't much selling or resistance to doing usability inspections. It is when we don't do these things that the team has difficulties with usability inspections.

5.2 What Do We Get That Makes It Worth Doing Usability Inspections?

We continue to do usability inspections at HP because teams find they get:

- New engineering skills doing user-centered development.
- At least a 12 to 1 return on investment comparing what we have to invest to find and fix defects doing usability inspections versus finding those defects after the product has shipped.
- Confidence the product will be usable by the people who use the product because we have compared the product to users and tasks and we have gotten input from a cross-functional team.
- Lowered risks of building a product that users won't buy.

Move Out of the Cage: User-Centered Design as the Key to Creating Quality Software

"Quality assurance" used to mean acting as the developers' conscience, making sure their code did more than just compile. The age of regression testing brought a more automated approach to examining products, but tests, statistics, and snappy slogans did little to improve the real-world quality of most software products.

Today, developing successful, quality software products requires more than just good engineering, testing, and statistics. This cramped view of quality tends to focus on technological issues while ignoring the larger issues of user needs, capabilities, engagement, and satisfaction. In this presentation I will discuss the user-centered approach to product development, and show how you can use it to break out of existing QA cages and create better products — without blowing your schedule or your budget.

Michael Sellers is the founder of New World Designs, a user interface consulting firm. NWD provides user-centered design, user interface design, and training services to companies creating advanced, graphical software products for medical, scientific, engineering, legal, and office-oriented markets. Sellers was most recently published in the July 1994 issue of ACM's *interactions*. He has worked in the software industry as a software engineer since 1984, and has been the bane of more than one QA person. He holds a BS in cognitive science.

New World Designs
2913 N. Aspen Way
Newberg, OR 97132

(503) 538-2745
sellers@acm.org

An Environment for Objective Usability Measurement

Mark A. Fleming, Newton C. Ellis and Dick B. Simmons
Cognitive Systems Laboratory
Texas A&M University
H.R. Bright Bldg., Room 425
College Station, TX 77843-3112

Keywords: Usability Testing, Metrics, Non-intrusive Measurement,
Human-Computer Interface

Abstract

This paper addresses the replacement of the labor-intensive, subjective approach to user interface evaluation with an objective approach based on data gathering and analysis.

Biographical

Mark A. Fleming received his bachelors and masters degrees in electrical engineering from Auburn University, Auburn AL. He had over ten years experience in industry as a software engineer prior to becoming a Ph.D. candidate in the Department of Computer Science at Texas A&M University. His research interests include visual programming and the visual interface, software models and metrics, and objective usability metrics. He is an IEEE Computer Society member.

Newton C. Ellis received an undergraduate degree that combined three years of engineering from Texas A&M University and two years of behavioral science from Baylor University. His Masters and Ph.D. degrees are in experimental psychology from Texas Christian University. He is a Professor of Industrial Engineering at Texas A&M University. He had 14 years of industrial experience in personnel, human factors and safety engineering before joining Texas A&M University. His current research focuses on fundamental questions regarding cognitive systems, knowledge acquisition techniques, and knowledge representation models. Dr. Ellis is a Fellow of the Institute of Industrial Engineers, where he has served on the Board of Directors as Vice President of Human and Information Systems, and a Fellow of the Human Factors Society.

Dick B. Simmons received the B.S. degree in electrical engineering from Texas A&M University, College Station, and the M.S. and the Ph.D. degrees in electrical engineering and computer and information sciences, respectively, from the Moore School of Electrical Engineering at the University of Pennsylvania, Philadelphia. He is an IEEE Senior Member, and a member of ACM, AAAAI, and the Society for Engineering Education. He has served on the IEEE Board of Directors and as President of the IEEE Computer Society, Amdahl Users Group, and Data General Users Group.

An Environment for Objective Usability Measurement

Mark A. Fleming, Newton C. Ellis and Dick B. Simmons*
Cognitive Systems Laboratory
Texas A&M University
H.R. Bright Bldg., Room 425
College Station, TX 77843-3112

Keywords: Usability Testing, Metrics, Non-intrusive Measurement,
Human-Computer Interface

Introduction

This paper addresses the topic of evaluation and improvement of the human-computer interface. The goal of our research is to augment the labor-intensive, subjective approach to user interface evaluation with an objective approach based on automated data gathering and analysis. The automated approach can reduce the time, effort and costs associated with usability experiments, while increasing the measures and metrics gathered during an experiment. A set of tools which address these goals, the **Usability Measurement Integrated Support Environment** (UMISE), are described in this paper.

This paper addresses an audience of managers and practicing professionals. An understanding of how such tools can be applied in current usability labs and the benefit of such application is provided.

*This research is currently funded under grants from the Texas Higher Education Coordinating Board (Grant 999903-180), the Applications Systems Division of IBM Corporation and Hewlett Packard.

Background

Methodological approaches to usability analysis fall into four categories: formal, heuristic, empirical, and automated[1]. The *formal approach,* starting from first principles, attempts to derive cognitive models of the user interface. Formal methods, such as the keystroke model[2], are more effective in modeling the quantitative aspects of the interface than qualitative aspects. The *heuristic approach* relies on the expert opinion of a number of evaluators to analyze the user interface. The use of extensive rules and guidelines[3], which focus the attention of the evaluators, avoids the charge of subjectivity. However, the approach is expensive due to the need for a number of experienced evaluators, as well as time consuming.

The *empirical approach* is based on the observation and recording of user activity, user interviews and questionnaires, thinking-aloud and protocol analysis, psychophysical recording, and field testing. The empirical approach "is in some sense irreplaceable, since it provides direct information about how people use computers and what their exact problems are with the concrete interface being tested."[4] The approach can be time consuming; in preparing experiments, gathering information, and analyzing results. The *automated approach* seeks to derive meaningful usability information through automatic data capture and subsequent analysis. This approach has great potential, particularly when applied to the rules and guidelines of the heuristic approach or the methods of the empirical approach.

Usability analysis can and should be used throughout the Software Life Cycle (SLC). Rule notations such as the Task-Action Grammar and the Computer Description Grammar, and graphic notations, such as State-Transition Diagrams, can be used to define the interface during the specification and design stages of the SLC. Methods adopted from the formal approach can be used to analyze quantitative aspects of the proposed interface, such as efficiency and consistency. During the design phase, rules and guidelines adopted from the heuristic approach can insure the final interface will meet widely accepted or standards-imposed requirements. Finally, methods adopted from the empirical approach can insure the interface meets real user needs, particularly when the prototyping or iterative SLC models are used.

The application of automation to the task of usability analysis can take two approaches. The first approach uses expert systems and other tools to enforce policy and procedure during the specification and design phases of the SLC. The KRI (Knowledge-based Review of user Interfaces) system[5] uses the critiquing paradigm to evaluate an interface under construction. The User Interface Design Assistant[6] combines an interface composition tool with a rule-based advice tool and a hyper-media knowledge presentation tool.

The second approach to automation, monitoring the user while interacting with the interface, can significantly improve the gathering of measures. For example, the PROTEUS system[7] automates the user questionnaire and information elicitation process. An activity log can be created which records each time-stamped action of the user or response by the interface. Performance measures, such as task time, time in Help, and number and frequency of functions used, can be extracted from the log. Other metrics derived directly from the user activity log include Multistep Matching[8], Maximal Repeating Pattern[9, 10], and Markov Chain Analysis[11, 12]. Metrics derived from the activity log have proved useful, their primary shortcoming being that they do not "speak the language" of the usability engineer.

Automated Support of Usability Experiments

The following sections deal with an environment for objective usability measurement, the Usability Measurement Integrated Support Environment (UMISE). The purpose of UMISE is to serve as a testbed for automated usability data gathering and analysis for emperical usability experiments. UMISE was created at the Cognitive Systems Laboratory (CSL), a collaborative arrangement between the Industrial Engineering and the Computer Science departments of Texas A&M University. The focus of the Lab is on the areas of Human Factors, Knowledge Acquisition and Engineering, Distributed Expert Systems, and Process Improvement.

A Usability Experiment

We began the investigation of the use of automated techniques by observing the actual process of creating, refining, testing, administering and evaluating the results of a specific usability experiment[13]. The experiment used the methods employed by the empirical approach to usability analysis to evaluate and compare two versions of the Knowledge

Acquisition Support Environment (KASE) tool. This tool, implemented using Knowledge System's Knowledge Management System (KMS) hypertext authoring tool in the Unix environment[14] and Microsoft FoxPro in the MS-Windows environment[15], was designed to support the knowledge acqusition and organization process when constructing an Expert System application.

The process of creating, administering and evaluating a usability experiment involved two classes of individual. The *Usability Engineer* was responsible for creating the experiment and evaluating the results. The *Observer* administered the experiment, provided guidance to the test subject, and recorded observations on the subject's response during the experiment. We analysed the usability experiment process to determine the phases of the process life cycle. We divided the process into the following life cycle phases:

- **Specification Phase -** The Usability Engineer began by specifying the subject, measures and goals of the experiment. The experiment examined two implementations of the expert system tool KASE (Knowledge Acquisition Support Environment) which ran in the Unix X Windows and MS-Windows environments. Measures selected included task time, user errors and user perception of each product, individually and separately.

- **Design Phase -** The engineer began a peer-review centered process in which a experiment scenario was created. The scenario was a sequence of tasks which would take the test subject from initial startup of the tool, through a series of interactions with the tool which exercised functions common to both implementations, until the subject completed the experiment and exited the tool. The scenario was examined to verify that each tool function was exercised and that the sequence of actions corresponded to a typical user session with the tool.

- **Implementation Phase -** A detail sheet for each task in the experiment was created, which contained the instructions to the test subject and a separate set of notes for the observer. The observer notes identified the tool function being exercised, measures the observer was to take, and expected user errors.

- **Testing Phase -** Validation of the experiment was done first by a series of walkthroughs by the engineer, then by several dry runs with volunteer subjects. The engineer made extensive notes regarding the understandability of instructions and

actual errors committed by the volunteers. The detail sheets for each task were refined and an error checklist created for use by the observer.

- **Operational Phase -** For each test subject, the observer printed a copy of the task instructions, the questionnaires the subject was required to fill out, and the observer's notes. Each subject then filled out a background information form and was provided a brief introduction to each tool with a quick tool walkthrough by the observer. The subject would then follow the instructions provided for each task, with the observer making notes as to task time and error. The subject filled out an evaluation form at the end of each tool session, and after both tools were exercised.

- **Analysis Phase -** The questionnaire results and observer notes formed the basis for the usability analysis of the two tools.

Dividing the usability experiment process into phases allowed us to focus on the process and determine how the process could be improved. We were able identify a number of problem areas, as well as areas where computerized assistance could enhance the job of the Usability Engineer. We observed that:

- The engineer crossed the boundary between a computer representation of the experiment (softcopy) and the printed version of same (hardcopy). The engineer used a PC-based word processor to develop the material and had to print this material before each walkthrough. Handwritten notes made during a walkthrough were then used to update the softcopy. A hardcopy of materials was made before each test subject began the experiment. Observer notes and measures, as well as questionnaire responses were transcribed into a computer readable form prior to statistical processing.

- The observer was the primary bottleneck during the administration phase. Each experimental run was a one on one event between the observer and test subject. The task of the observer was primarily clerical, while occasionally answering questions from the test subject.

- Not all errors observed during the experiment were anticipated and contained in the observer's checklist. The dry runs helped a great deal with identifying possible user

errors for each task, but the range in skill and experience of test subjects precluded complete identification.

- Performance measures taken were few in number. Measures were limited to the traditional stopwatch and checklist, with a subsequent heavy reliance on the results of subjective questionnaires.

Lessons Learned

Observation of the usability experiment process revealed certain inefficiencies and bottlenecks. Our next task was the creation of an environment which could provide automated assistance to the Usability Engineer and which would extend the range of measures and metrics gathered during a usability experiment. The need for three tools to create such an environment were recognized:

- **Guidance Tool -** Eliminating the softcopy/hardcopy boundary crossing and the clerical duties of the observer would increase the efficiency of the usability experiment process. A Guidance Tool would eliminate the paper copies of task descriptions and questionnaires. The tool would also provide a help facility to answer the most common test subject questions, and administer the pre- and post-experiment questionnaires. Finally, the tool could be used interactively by the Usability Engineer during the implementation and testing phases to enter, correct and refine the experiment materials.

- **Monitoring Tool -** Gathering performance measures, user events, and application activity in an unobtrusive manner would provide the basis for objective usability metrics. Properly implemented, such a tool could reduce or eliminate the need for an observer during the operational phase. Simultaneous experiments would considerably reduce total time.

- **Metrics Tool -** The raw measures gathered during an experiment would be immediately available for processing, without the need to transcribe these measures and questionnaire results into computer-readable form. The raw measures would form the basis of usability metrics, preferably in a high level form which directly addresses the needs of the Usability Engineer. This means not only performance metrics and user

preference questionnaire results, but also such metrics as "Learnability," "Number and Severity of User Errors," "Efficiency," etc.

The Usability Measurement Integrated Support Environment (UMISE)

The Guidance tool was based on *wxClips*[16], a version of the Clips expert system with graphical interface extensions provided by the wxWindows[17] object-oriented GUI library. A markup language was created for the Usability Engineer to describe the tasks in an experiment. The engineer would enter the descriptive task text and the text of an optional help screen. Designated tasks could also be accompanied by fill in the blank queries, checkboxes, and ranking scales. A translator converted the description file into wxClips source code. The resulting Clips application would present the text of each task description in a window, allow sequencing to the next task by clicking the OK button, popup a Help window when the HELP button was clicked, and present information query forms to the user when required. The information solicited from the user would be recorded to a log file, along with timestamps of when the user sequenced to the next task or requested help.

The X Window System running on a Unix workstation was chosen as the base environment for UMISE due to the availability of source code and the number of existing support tools. To perform the monitoring task, we examined two public domain systems, *xmon*[18] and *XTrap*[19]. The first, xmon, was found to have the necessary functionality as well as an easy-to-use interface. Unfortunately, xmon was found to have a significant performance impact on bitmap graphic oriented applications. XTrap, an extension to the X11 server, was adopted for use in UMISE. XTrap allows a client process to monitor user activity and application responses mediated by the X11 server. By merging the xmon interface with the supplied sample clients of XTrap, we were able to create a monitoring tool which combined the best attributes of both systems.

The metrics tool for UMISE was primarily concerned with formatting the resulting monitoring and guidance log files in a form which could be processed with a statistical package, a spreadsheet, or imported into a word processor. Actual metrics information is derived using these tools.

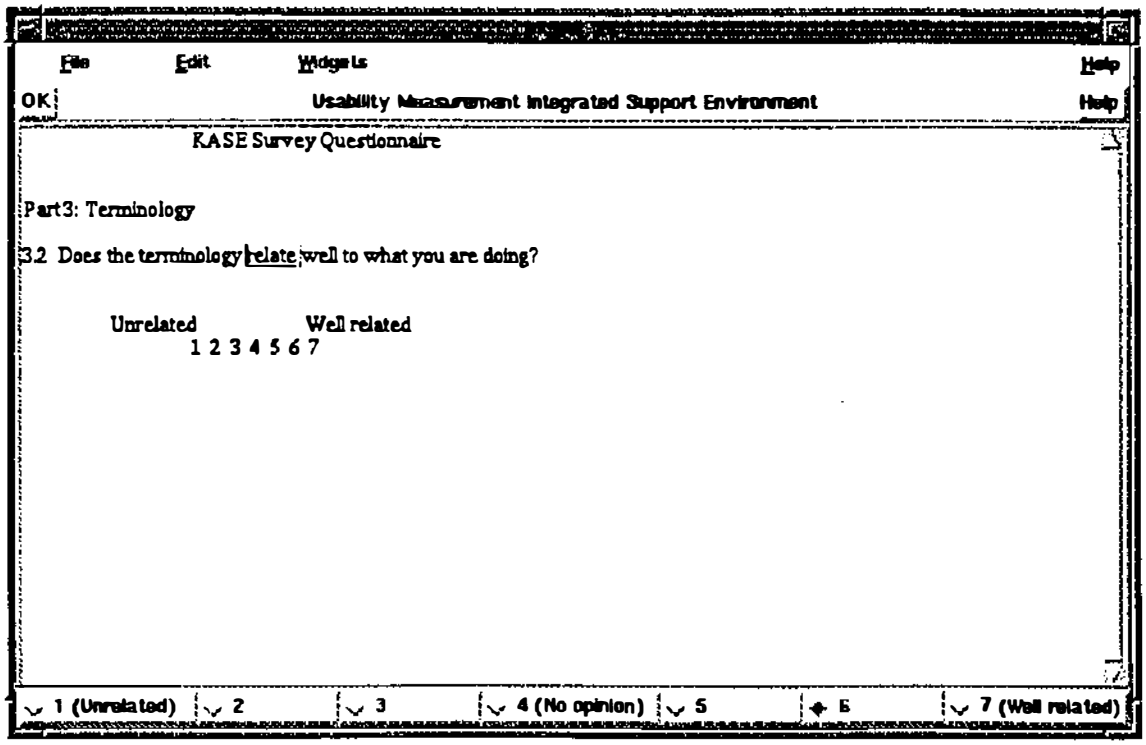


Figure 1: The revamped Guidance tool, a tcl/tk script. This expert version allows the engineer to interactively create the guidance scripts.

Experiment and Results

Our plan was to repeat the Unix portion of the KASE tool usability experiment using UMISE. Our goal was to automatically gather the same information as the manual experiments, augmented with any additional information that could be gathered using the monitoring tool. UMISE components were used throughout the experiment life cycle. The specification, design, implementation and testing phases of the life cycle involved converting the task description sheets from the previous experiment to a format which could be processed by the guidance tool. Conversion, which involved inserting markup commands into an ASCII copy of the original word processor file, proceeded quickly. User help sections were added, and the questionnaires were merged and converted to their markup representation. We followed the steps of the life cycle when adding the Help screens, i.e., walking through the experiment using the guidance tool and application together on the display. As possible user problems were noted, material was added to the corresponding task Help section. The result was converted to Clips code using the translator.

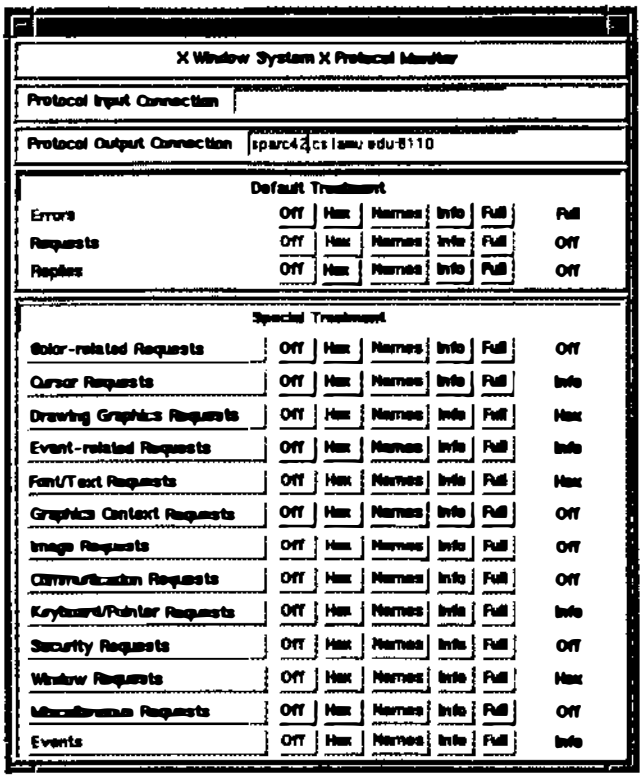
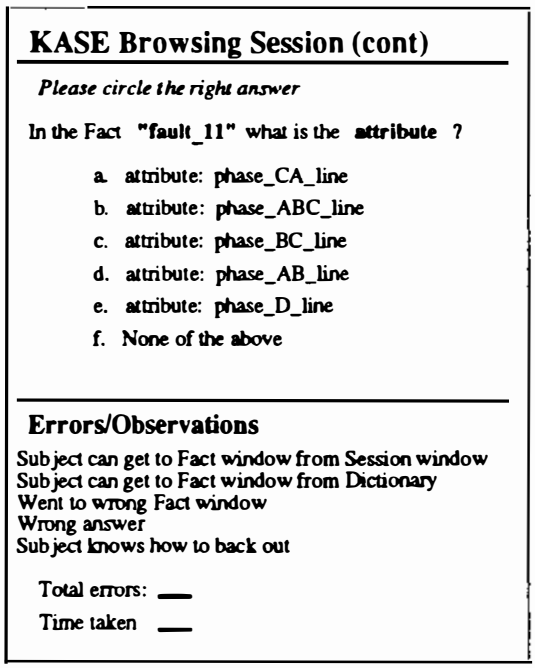


Figure 2: The tcl/tk interface to the XTrap-based monitoring tool.

The monitoring tool client was programmed to capture user events, such as button presses and keystrokes. The resulting user session log file was in an easily processed ASCII format. The metrics tool performed two tasks; clustering of data points and translation of the log files to a format that could be processed by other analysis tools. Clustering was applied to the cursor location during button press events. The position of the cursor at the time of a mouse button press was analyzed using all log files. Positions naturally clustered around the button and hypertext hot spot locations on the screen. Individual button presses in all log files were then replaced with a standardized screen position, and this position cataloged for later analysis. The event associated with clicking the OK button in the guidance tool provided a marker to divide the event log into sections corresponding to individual tasks in the experiment. With each user log record normalized, the `diff` command was then used to perform a simplified form of Multistep Matching. Each sequence of events associated with a task were extracted and compared to a test log sequence generated by an expert using the experiment material. An add/change/delete counting metric and path length metric was generated for each task in a session.



KASE Browsing Session (cont)

Please circle the right answer

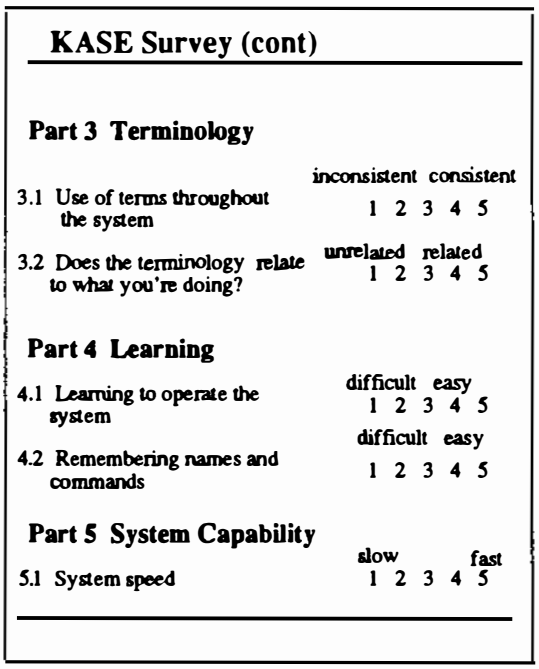
In the Fact **"fault_11"** what is the **attribute** ?

a. attribute: phase_CA_line
b. attribute: phase_ABC_line
c. attribute: phase_BC_line
d. attribute: phase_AB_line
e. attribute: phase_D_line
f. None of the above

Errors/Observations

Subject can get to Fact window from Session window
Subject can get to Fact window from Dictionary
Went to wrong Fact window
Wrong answer
Subject knows how to back out

Total errors: ___
Time taken ___



KASE Survey (cont)

Part 3 Terminology

	inconsistent consistent
3.1 Use of terms throughout the system	1 2 3 4 5
3.2 Does the terminology relate to what you're doing?	unrelated related 1 2 3 4 5

Part 4 Learning

	difficult easy
4.1 Learning to operate the system	1 2 3 4 5
4.2 Remembering names and commands	difficult easy 1 2 3 4 5

Part 5 System Capability

	slow fast
5.1 System speed	1 2 3 4 5

Figure 3: Sample observer notes and survey questionnaire pages (original experiment).

Clustering was also applied to the delta time between events. Event delta times were clustered into short and long groups. These were further divided into delta time between button or key presses (type 1), between press and release (type 2), and between release and subsequent press (type 3). Type 2 button deltas were treated as application response times, while types 1 and 3 button deltas were treated as user response times. Key deltas provided information on user typing speed and pauses.

The guidance tool script gathered user responses before and after an experimental run in the Unix environment. The response and preference data was output to a separate log file in the form of question number and user response. Responses were textual, either typed in by the user or selected from a menu. An example typed in response is the user's name. Menu responses were generated when the user selected the answer to a ratings question from a seven item menu, i.e. Strongly Agree to Strongly Disagree. The log file contents for all users was directly processed to generate user preference information.

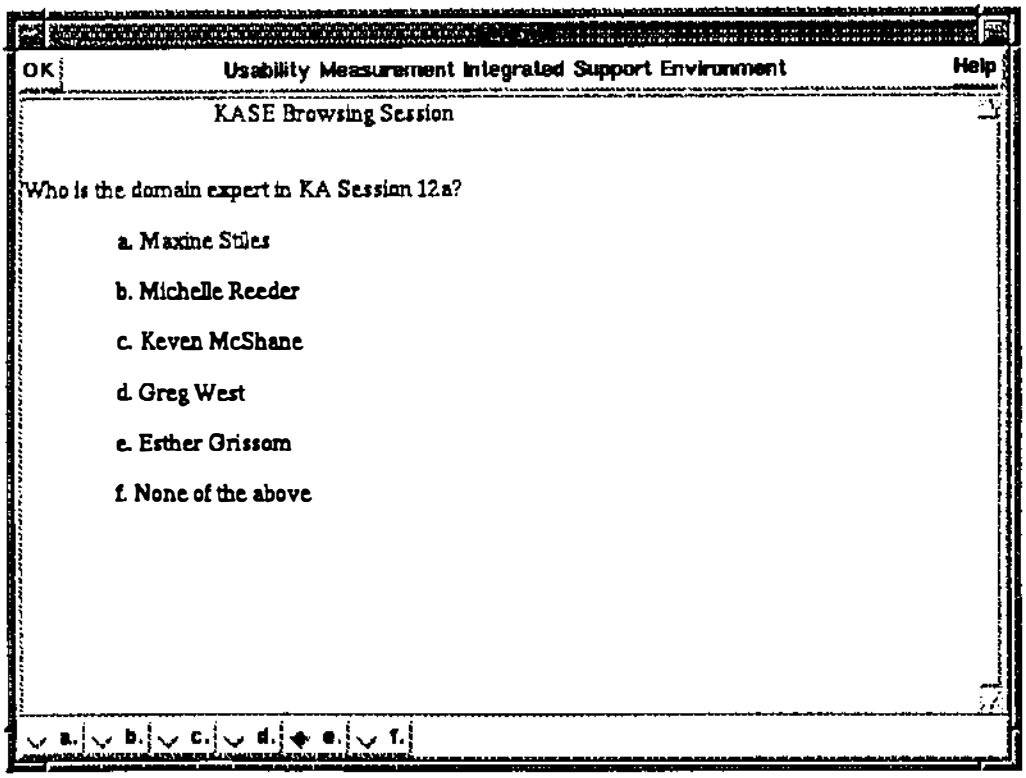


Figure 4: The guidance tool used during the experiment.

Epilogue

An important lesson was learned while implementing UMISE. We initially attempted to monitor users of the original KASE tool usability experiment using *xmon* for the Unix version and Borland's *Spy* for the MS-Windows version. In both cases, moderate to severe performance impact on the tool being monitored forced the immediate termination of monitoring activity. Users perceived the degradation in response time and interpreted it as a negative characteristic of the tool being measured. Any automated environment must be unobtrusive if it is to be of any practical use.

UMISE has shown promise as a testbed for objective usability measurement. The system addresses the usability experiment life cycle and removes some of the bottlenecks in the process. In serving the needs of the Usability Engineer, UMISE could help expand the role of usability analysis in the software development process.

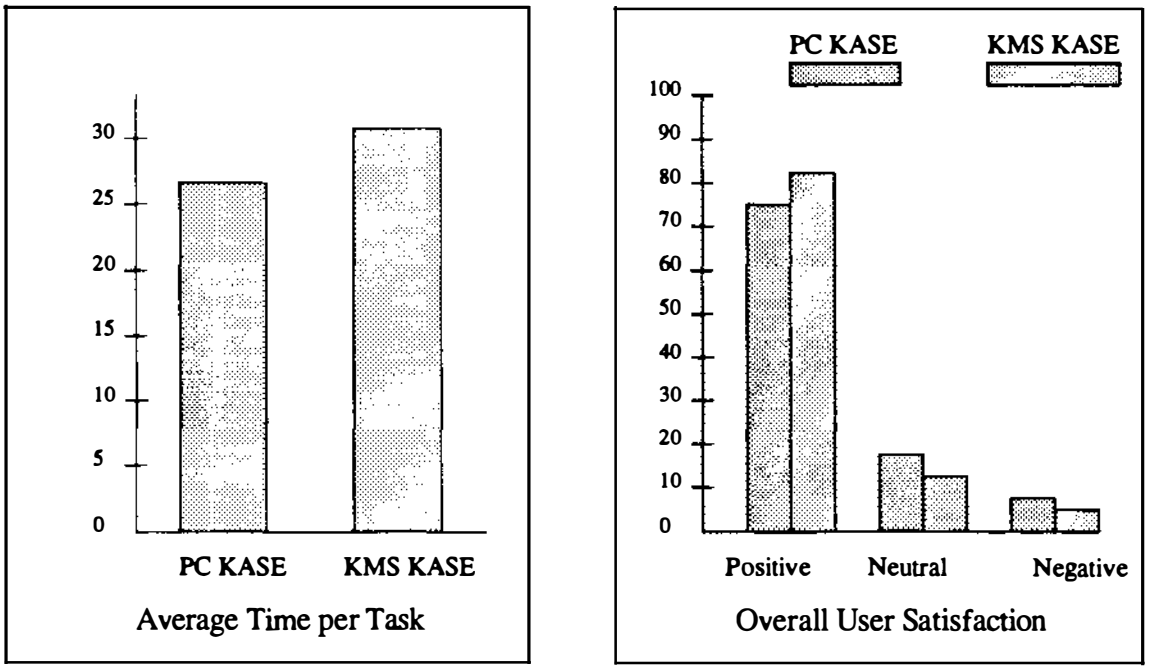


Figure 5: Sample experiment results.

Improvements

The XTrap extension allows the client monitoring process to record and playback user and application event messages. This feature can be used to review a user session at a later date, or allow the Usability Engineer to remotely monitor user activity in real time. Monitoring would be reasonably easy to implement; this, in fact, was one reason XTrap was developed. Our present experiments with recording/playback and remote monitoring of user sessions use only the event messages generated by the user. This necessitates the exact duplication of initial conditions. A usable playback or monitoring system would need to record and reproduce the application response messages to provide videotape-style reproduction.

Feedback of user and application activity from the Monitoring and Metrics tools to the Guidance tool would allow the Guidance tool to react to the actions taken by the user. This possibility was uppermost in mind when wxClips was selected as the basis for the tool. Additional help could be provided the user, the sequence of tasks in an experiment could be altered in response to the user's success, questionnaire content could be tailored to the users performance, and high level reasoning and conclusions could be drawn about

the users performance.

The use of log files to store information from the guidance and metrics tools proved somewhat cumbersome. Given the nature of data processing and the need to access the data from multiple user sessions, a database system will be used in the future to store the generated information. Adding database capability is relatively straightforward through the use of embedded SQL statements.

Future Work

Perhaps the most important area for improvement is in the Metrics tool. The metrics applied in our qualification test were the simple and familiar performance measures. Others, adopted from the literature, proved useful but did not "speak the language" of the engineer. Ideally, a metrics tool should generate results which correspond directly to the questions asked by the Usability Engineer.

In keeping with this goal, we intend to integrate a planning agent expert system into UMISE. A planning agent can be used in two ways: prior to an experiment, the agent can determine all possible paths from initial state to goal state for a given task. The precalculated paths can then be compared against the actual path followed by the user using the Multistep Matching method. The agent can also be used while the user performs the experiment. Actions taken by the user can be fed to the agent to update the initial state. Actions which do not contribute towards achieving the goal state (such as invoking the Help facility) can be tagged and filtered. Actions which negate progress towards the goal (user mistakes), or which prevent the goal from being achieved (user errors) can also be recognized.

The two primary metrics a planning agent would extract from the user record, post-experiment or real time, would be Task Completion and User Error. Using these two metrics, the features of UMISE, and with a properly structured experiment, we feel we can directly address much of Nielsens five usability characteristics;[20] Learnability, Efficiency, Relearnability and Recall, Frequency and Severity of user errors, and Subjective User Satisfaction.

References

[1] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," in *Proceedings, CHI'90*, pp. 249–256, ACM, New York, 1990.

[2] S. K. Card and T. P. Moran, "The keystroke-level model for user performance time with interactive systems," in *Readings in Human-Computer Interaction, A Multidisciplinary Approach*, pp. 192–206, Morgan Kaufmann Publishers, Inc. San Mateo, CA, 1987.

[3] S. L. Smith and J. N. Mosier, "Guidelines for designing user interface software," Tech. Rep. ESD-TR-86-278, The MITRE Corporation, Bedford, Mass., 1986.

[4] J. Nielsen, *Usability Engineering.* Academic Press, Inc., 1993.

[5] J. Löwgren and T. Nordqvist, "A Knowledge-Based Tool for User Interface Evaluation and it Integration in a UIMS," in *Human-Computer Interaction - INTERACT'90*, pp. 395–400, 1990.

[6] H. Reiterer, "A Human Factors Based User Interface Design," in *Human-Computer Interaction - VCHCI'93*, pp. 291–301, 1993.

[7] J. Crellin, "PROTEUS: An Approach to Interface Evaluation," in *Human-Computer Interaction - INTERACT'90*, pp. 389–394, 1990.

[8] N. Kishi, "SimUI: Graphical User Interface Evaluation Using Playback," in *Proc. 16th Int'l Computer Software and Application Conference*, pp. 121–127, 1992.

[9] D. Hix, D. Radin, A. C. Siochi, and D. Benel, "Computer Analysis of User Session Transcripts for Evaluation of the Human-Computer Interface," in *Proc. Southeastcon'91*, pp. 1011–1015, 1991.

[10] A. C. Siochi and D. Hix, "A Study of Computer-Supported User Interface Evaluation Using Maximal Repeating Pattern Analysis," in *Proc. CHI'91 Conf. on Human Factors in Computing Systems*, pp. –, 1991.

[11] M. Guzdial, C. Walton, M. Konemann, and E. Soloway, "Characterizing Process Change Using Log File Data," tech. rep., GVU Center, College of Computing, Georgia Institute of Technology, 1993.

[12] M. Guzdial, "Deriving Software Usage Patterns from Log Files," tech. rep., GVU Center, College of Computing, Georgia Institute of Technology, 1993.

[13] S. Kwan, D. B. Simmons, and M. A. Fleming, "Usability evaluation of two knowledge acquisition support environment (kase) tools," Tech. Rep. CSL-TR-27-94, Cognitive Systems Laborary, Texas A&M University, College Station TX, 1994.

[14] P. Reddy and D. B. Simmons, "Knowledge acqusition support environment (kase)," Tech. Rep. CSL-TR-26-91, Cognitive Systems Laboratory, Texas A&M University, College Station TX, 1991.

[15] S. Guan, "Implementation of kase in ms windows using microsoft foxpro," Tech. Rep. CSL-TR-41-94, Cognitive Systems Laboratory, Texas A&M University, College Station TX, 1994.

[16] J. Smart, "User manual for wxclips," tech. rep., Knowledge Based Decision Support Group, Artificial Intelligence Applications Institute, 80 South Bridge, University of Edinburgh, EH1 1HN, 1993. Available via anonymous ftp from skye.aiai.ed.ac.uk:/pub/wxwin.

[17] J. Smart, "User manual for wxwindows 1.50: A portable c++ gui toolkit," tech. rep., Knowledge Based Decision Support Group, Artificial Intelligence Applications Institute, 80 South Bridge, University of Edinburgh, EH1 1HN, 1993. Available via anonymous ftp from skye.aiai.ed.ac.uk:/pub/wxwin.

[18] G. McFarlane, *xmon - interactive X protocol monitor*, 1990.

[19] D. Annicchiarico, R. Chesler, and A. Jamison, *The XTrap Architecture*. Digital Equipment Corporation, 1991.

[20] J. Nielsen, "The usability engineering life cycle," *IEEE Computer*, vol. 25, no. 3, pp. 12–22, 1992.

Achieving Software Quality Through Design Metrics Analysis

Wayne M. Zage
Computer Science Dept.
Ball State University
Muncie, IN 47306
(317) 285-8664
wmz@bsu-cs.bsu.edu

Dolores M. Zage
Computer Science Dept.
Ball State University
Muncie, IN 47306
(317) 285-8646
dmz@bsu-cs.bsu.edu

Cathy Wilburn
Northrop/Grumman Electronics
Systems Division
Rolling Meadows, IL 60008
(708) 259-9600

Abstract

To aid in the search for better methods of software engineering, we are analyzing university and industry software to be able to correctly identify error-prone modules during the design phase of software development. Our research team has developed and tested our design metrics over a five-year period and has found them to be excellent predictors of error-prone modules. The Design Metrics Analyzer (DMA) has been developed for the Unix environment to automate metrics collection, calculation and analysis of Ada systems. This tool improves the metrics collection by ensuring the accuracy of metrics calculation and by minimizing the interference with the existing work load of a software engineer. This paper highlights empirical results from our research and gives an overview of the DMA tool.

Keywords and Phrases: software quality, design metrics, design metrics analyzer, software tools.

Biographical:

Wayne M. Zage is a professor of computer science at Ball State University and an active researcher in the Software Engineering Research Center (SERC) at Purdue University. His research interests include software metrics and models and software engineering environments. He is the recipient of the Ball State University Outstanding Researcher Award for 1994 and had previously won the University's Outstanding Young Faculty Award. Zage received his doctorate from the University of Illinois at Chicago.

Dolores M. Zage is an assistant professor of computer science at Ball State University and an active researcher in the SERC. She has been the co-principal investigator (with Wayne) of the highest rated research project in the SERC, as voted by industry, for the last four years. Her primary research interests are software metrics and models and software design analyzers. Zage received an MS degree in computer science from Purdue University.

Cathy Wilburn is a software engineer at Northrop/Grumman Electronics Systems Division. She has conducted metrics research for several years, and is currently conducting design metrics analyses of large-scale development projects at Northrop/Grumman. Wilburn received an MS degree in computer science from Ball State University.

Introduction

The story of software development is the tale of identifying and tackling complexity. The ability to uncover troublesome spots in a system can provide a tremendous advantage to designers, developers, testers, maintainers and managers as they strive to accomplish their tasks. Meaningful measurements of software complexity and quality are desirable to help ensure system dependability, evaluate development methods, contribute to process control, and verify that the product meets a set of quality goals. Further, on large and/or complex systems, it is important to take measurements during development so that potentially troublesome designs or code can be detected early enough to perform corrections with minimum additional effort.

The design of software is similar to the design process of building a physical structure. Suppose you are overseeing the building of a mountain lodge. An architect first would help you determine and analyze the requirements for the lodge. For example, you may want eleven rooms built on four levels. The architect then would produce the overall design of the house, including the placement of bedrooms, bathrooms, the kitchen and so forth, all the while keeping an eye on traffic flow patterns developing in the design. This early, architectural design would focus on the appearance and functionality of your lodge. A physical model might be developed as a prototype of the desired structure. Then the architect would develop a more detailed design of your building, including the placement of pipes, electrical outlets and the specification of materials needed, such as the foundation needed to support the lodge. He feels confident that the design depicts a lodge that not only satisfies your requirements, but also is functionally sound and aesthetically pleasing to the customer.

In designing software, software engineers also produce a model or blueprint of a developing system. However, unlike the architect's model of the mountain lodge, the resulting software design cannot be easily assessed. An approach for analyzing software designs that helps designers engineer quality into the design product needs to be established. In our research, we have developed design metrics that are calculated during architectural design and detailed design that will determine how a design is progressing. Using design metrics will provide the designer with a firm foundation to begin software construction, just as the results of the architect's analysis of his model gives him confidence to begin physical construction of the mountain lodge.

This paper illustrates the application and benefits of design metrics in the software development process. In particular, we focus on the architectural design metric D_e, the detailed design metric D_i, the concept of design balance and the automation of our approach through our Design Metrics Analyzer.

Our Approach

The goal of the design metrics calculations is to identify error-prone modules during the design phase of software development. The research method employed is to calculate the particular design metric for each module under consideration. Those modules whose metric value is more than one standard deviation above the mean for that metric over all of the modules considered are identified as outliers (or stress points). Later, when error reports are available, we determine if the stress-point modules actually contained errors. Over a five-year

metrics evaluation and validation period, on study data consisting of university-based projects and large-scale industrial software, our design metrics have been excellent predictors of error-prone modules.

The design phase is divided into two parts, architectural and detailed. During architectural design, software engineers create and assemble the components of a system into a structure that defines the interconnections between modules. Hierarchical module diagrams, data flows, functional descriptions of modules and interface descriptions are products of the architectural design phase. After completing detailed design, all of the previous information plus the chosen algorithms and in many cases either pseudocode or a program design language representation for each module are available. Detailed design is an extension of architectural design, where each module's functionality is more clearly identified and refined.

To parallel these two phases of design, the Software Engineering Research Center (SERC) Design Metrics Research Team at Ball State University has developed two design metrics: an external design metric D_e and an internal design metric D_i. The calculation of D_e focuses on a module's *external* relationships to other modules in the software system and is based on information available during architectural design. The metric D_i incorporates factors related to a module's *internal* structure and is calculated during the detailed design phase of software development.

Definition of D_e

D_e is calculated for each module in a system and is defined as

$$D_e = e_1(\text{inflows} * \text{outflows}) + e_2(\text{fan-in} * \text{fan-out})$$

where

inflows is the number of data entities passed to the module from superordinate or subordinate modules,

outflows is the number of data entities passed from the module to superordinate or subordinate modules,

fan-in and *fan-out* are the number of superordinate and subordinate modules, respectively, directly connected to the given module,

and e_1 and e_2 are weighting factors.

The term *inflows* * *outflows* provides an indication of the amount of data flowing through the module. Because a quadratic expression best explains the relationship observed in our research between data-flow through a module and the likelihood that the module was error prone, the degree of this term is two. The term *fan-in* * *fan-out* captures the local architectural structure around a module since these factors are equivalent to the number of modules that are structurally above and below the given module. This product gives the number of invocation sequences through the module. Through our early research, we found that the use of both of these products in D_e gave the best predictor of error proneness in a system.

To show an elementary calculation of D_e, consider the module *Perform_Word_Count* in Figure 1. Inflows to this module are file_name, status_flag and word_count (twice). The only outflow is file_name. Therefore, using unit weights, $D_e = (4 * 1) + (1 * 3) = 7$.

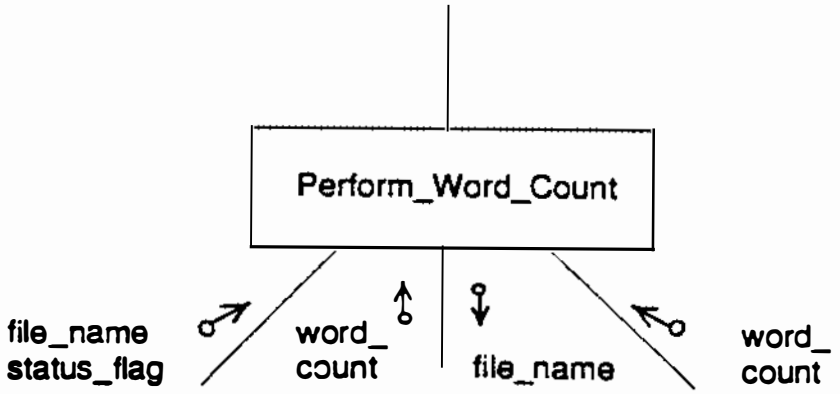


Figure 1: Module With D_e Value of 7

The Practical Application of D_e

Suppose that a software design was represented by the structure chart shown in Figure 2. Glancing at the figure, the system appears to have a reasonably well-designed structure. This is perhaps all that can be said about this design without closely examining every module's specification and checking such design heuristics as the levels of factoring, cohesion and coupling. Our intuition that the design possesses good quality may be assured by the knowledge that the design team checked and rechecked the design against these important design criteria. However, large designs that may require several volumes of documentation cannot be quickly reviewed. Moreover, designs that are concatenated from several partial designs must be reevaluated to reveal the goodness of the interconnections between the components. What is needed is a comprehensive design analysis that is easily accomplished and provides an insight into design quality. D_e can serve as a significant component of this analysis.

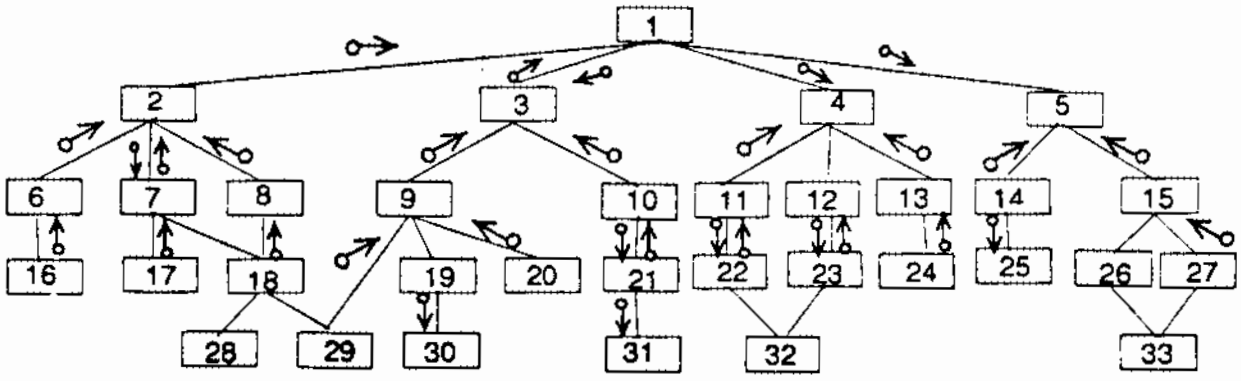


Figure 2: Hypothetical System's Structure Chart

After architectural design or a subset of the design is completed, a software designer can calculate D_e values for the modules in the developing system. The designer then would

observe which modules were outliers with respect to D_e. Referring to Figure 2, suppose modules 3, 7 and 21 possess D_e values that are at least one standard deviation above the mean for the modules in the design. Even in this small example where the entire design can be viewed at once, the designer may not have been led to review these modules. These outliers, considered as stress points in the design, then are reviewed to determine if a redesign, such as dividing a module with too much functionality into two modules, is in order. (There may be circumstances where, due to the inherent complexity of the application or management's rationale for a given structure, redesign is not the best alternative.)

After reviewing this architectural design, the designer can either proceed to detailed design or revise the architectural design of the software, and then recalculate the D_e values. Once this iterative process is completed to the satisfaction of the designer, detailed design begins. Calculating D_e during architectural design uses the information currently available about the developing system and provides an indication of project status and complexity early in the life cycle.

Experimental Results Using D_e

Why should designers reevaluate the stress points identified by D_e? In our research, we first tested D_e on projects developed at Ball State University in a software engineering sequence. These projects varied in size (from 2,000 to 30,000 LOC) and in the application language. We asked if D_e can be effective as an indicator of error-prone modules for the projects in our database. Over all of the projects studied, D_e highlighted 12% of the modules as stress points. Half of these actually contained errors. Furthermore, 53% of all of the errors detected were in those highlighted modules. Incidentally, 58% of the error modules were not found. We felt that these results were positive since they were obtained using only the information available during the architectural design of the system.

At this point in our research, we looked forward to working on larger projects developed in industry to see if our results would hold. In 1986, the Standard Finance System Redesign (the STANFINS-R) contract was awarded to the Computer Sciences Corporation (CSC) at Fort Benjamin Harrison in Indiana. It contained approximately 532 programs, 3,000 packages and 2,500,000 lines of Ada. We completed our design metrics study using the design documentation, code and testing results of 21 programs (approximately 24,000 lines of code) that were selected by CSC development teams. On the STANFINS data, D_e performed even better than on the university database by targeting 67% of the known errors, as compared to 53% on the university database, when identifying 12% of the modules as stress points. We also compared the error versus errorless modules with respect to their D_e values in this study and found that the D_e mean for error-prone modules was more than six times the mean for errorless modules!

The results just stated were based only on the calculation of D_e at the end of architectural design. However, the design of software becomes an incremental process as the designer adds new information or explores alternative designs. In another study, we focused on the integration of the construction of architectural design and the measurement of this process by a sequence of D_e calculations. Taking these snapshots, or time-slices, of design may help us identify problematic components of a design, and take corrective actions while it is still relatively inexpensive to do so. By evaluating D_e over time, we gained an understanding of

design dynamics in order to highlight favorable trends and avoid unfavorable ones in software design. This project, the Time-Slice Transformations on Architectural Design Metrics or TSTADEIM project, tracked the transformations of D_e on developing software.

A major effort of our TSTADEIM project research has been a collaborative study with Magnavox Electronic Systems Company. The target software in this study was the Advanced Field Artillery Tactical Data System (AFATDS) project. Our goal was to observe the transformations of D_e in maturing software. Code from three releases of a minor component called the Human Interface of the AFATDS project was selected for this study. From analyzing the transformations of the architectural design metric D_e, insights into how software developers build software and the process's effects on metric values were uncovered [ZAGE92a].

Definition of D_i

The internal design metric D_i is defined as

$$D_i = i_1(CC) + i_2(DSM) + i_3(I/O)$$

where

CC, the *Central Calls*, is the number of procedure or function invocations

DSM, the *Data Structure Manipulations*, is the number of references to complex data types, which are data types that use indirect addressing

I/O, the *Input/Output*, is the number of external device accesses,

and i_1, i_2 and i_3 are weighting factors.

There were many candidates to select from for incorporation into D_i. Our aim was to choose a small set of measures that would be easy to collect, would identify stress points, and would highlight error-prone modules. During the analyses of errors for the projects in our database, our research team identified three areas as the locations where the majority of errors occurred. These were at the point of a procedure invocation, in statements including complex data structure manipulations, and in input/output statements. Thus we selected three measures, CC, DSM and I/O to form the components of D_i.

The Application of D_i

As an illustration of the calculation of D_i, consider the pseudocode shown below for the module *Perform_Word_Count* depicted in Figure 1. The statements in *italics* are central calls, those in UPPER CASE letters are data structure manipulations, and those in **bold** font are I/O.

```
Perform_Word_Count
    get_input(validated_filename, status_flag)
    if (status_flag) is false then
        ERROR_TABLE(1) = filename
        print ("Error 1: file is not a text file.")
    else
        count_number_of_words(validated_filename,word_count)
        if(word_count) = -1 then
            ERROR_TABLE(2)="does not exist"
            print ("Error 2: file does not exist.")
        else
            process_output(word_count)
        end if
    end if
```

In this example, using unit weights, $D_i = CC + DSM + I/O = 3 + 2 + 2 = 7$. If this module was in a developing system, the designer would compare its D_i value to the stress point cut-off value to determine if further review of this module was in order. The reason for calculating D_i during detailed design is to use the information currently available about the developing system to provide an indication of project status and complexity during the detailed design phase of software development. Armed with this information, the designer either can look for alternatives to a particular part of a design, assign difficult components to experienced developers, or plan to provide extra testing effort for the indicated stress points.

Experimental Results Using D_i

In our quest for a D_i which would be a good predictor of error-prone modules, we studied D_i with several weighting schemes. The best results occurred on our test bed of projects when we set $i_1 = i_3 = 1$ and $i_2 = 2.5$, stressing the data structure manipulation usage within the modules. We found 94% of the errors occurred in 89% of the highlighted modules, with false positives occurring 11% of the time. Thus using $i_2 = 2.5$ gave excellent results as a predictor of error-prone modules. We also asked how these results compared to the time-honored metrics cyclomatic complexity V(G) and LOC (obviously calculated later in the life cycle) as a predictor of error-prone modules. In this study, a module was identified as a stress point using V(G) if its metric value was greater than or equal to 10 (the value that McCabe thought was a reasonable upper bound [MCCA76]), and using LOC if the size of the module was greater than one standard deviation above the mean for all modules of that particular project. The results are summarized in Table 1.

Table 1: Comparing Metrics' Ability to Identify Error-Prone Modules

	$V(G)$	LOC	D_i
Modules Highlighted	11%	11%	11%
Highlighted Mods w/Errors	44%	56%	89%
Detected Errors Found	37%	51%	94%
False Positives	56%	44%	11%

Note that the highlighted modules with errors, the percent of detected errors found and the percentage of false positives were not nearly as favorable using V(G) or LOC as when using D_i. And yet the D_i metric is available earlier than specific LOC counts! None of the other well-known measures that we tested performed better than the D_i metric as a predictor of error-prone modules [ZAGE90]. D_i also performed well on industrial data, where the stress points highlighted contained 74% of the detected errors [ZAGE93]. The design metrics, as shown by experimental results, give a software designer a good indication of where trouble spots exist.

Experimental Results Using the Design Metric $D(G)$

The Design Metrics Research Team has developed a design metric $D(G)$ of the form

$$D(G) = D_e + D_i$$

where D_e and D_i are our external and internal design-quality components, respectively. Weighting factors are not needed in this equation, since weights are associated with each of the primitive components in the equations for D_e and D_i. We asked if $D(G)$ could do any better as a predictor of error-prone modules than either D_e or D_i on study data consisting of university-based projects. The answer to this question is shown in Table 2. Note that all of the modules highlighted as outliers by $D(G)$ contained errors, and in fact 97% of all of the errors detected in the systems were contained in these highlighted modules! The number of false-positives was 0 [ZAGE90].

Table 2: Our Design Metrics' Ability to Identify Error-Prone Modules

	D_e	D_i	$D(G)$
Modules Highlighted	12%	11%	12%
Highlighted Mods w/Errors	50%	89%	100%
Detected Errors Found	53%	94%	97%
False Positives	50%	11%	0%

We then needed to determine if these results would hold as we tested $D(G)$ on large-scale, industrial-based software. The study data was again the same subset of CSC's STANFINS-R software system. The $D(G)$ metric was very successful in detecting the

STANFINS modules with errors, and only 18% were false positives. On this test bed, 74% of the errors were detected for modules with 10 or less errors, 82% of the errors were detected for modules with 11 to 20 errors, and 100% of the errors were detected for modules with 21 or more errors [ZAGE93]. In other words, modules with high concentrations of errors were always identified as stress-points by $D(G)$.

Design Balance

Theoretically speaking, software engineers understand that an appropriate amount of modularity is a necessary component in a quality design of a system. Too few modules can lead to monolithic units of code that are hard to maintain and enhance. Such modules have a low level of cohesion and high D_i metric values. Too many modules can lead to excessive communication overhead and extensive coupling between modules. These modules would typically have high D_e values. A good design possesses an appropriate level of modularity, as reflected in a balance between average D_e and D_i values. This design balance is measured by the *design balance* metric, DB, defined as

$$DB = AvgD_e / AvgD_i$$

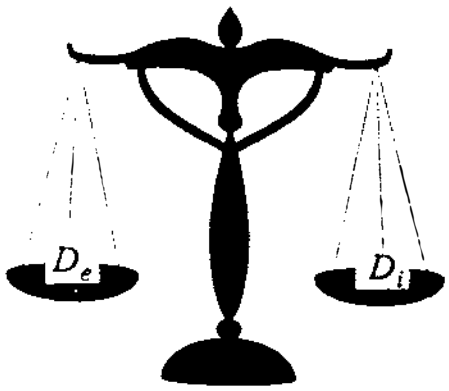


Figure 3: Balancing the External and Internal Complexity in a System

As D_e and D_i become ''balanced'', the designers achieve a spread of complexity between the internal functionality of the modules, as given by D_i, and the external coupling between modules, as given by D_e. From previous studies, we have seen quality software systems having a DB value between 3 and 10 [ZAGE92b]. Although this range is only an observation and is not statistically supported, we believe this range to be a reasonable guideline based on its foundation in the theoretical principles of software design.

The Design Metrics Analyzer

Tools must be available to successfully integrate the use of software metrics into the software development process. Tools ensure consistent measurements and minimize the interference with the existing work load. Several large Ada projects from industry were

offered as data for this research. Ada is increasingly being used as a design representation language for several reasons [AGRE92]. First, if both design and implementation are done in the same language, the translation from one stage to the next is made more easily. Also, the Ada design syntax can be checked by an Ada compiler and tools available for Ada can be used to evaluate the designs.

To make our analyses more efficient and consistent, and to assist software engineers in determining which components in a system need to be reexamined, a Design Metrics Analyzer (DMA) for Ada was created for the Unix environment. The DMA calculates and collects several metrics on each Ada module. The metrics collected are our design metrics, as well as other metrics, including various size metrics. The design metrics results are analyzed to yield a list of stress points, which are modules that are considered to be error-prone or difficult for developers. In addition to stress-point reports, other reports that are helpful for documentation purposes and reengineering and reverse engineering efforts are also produced by the DMA.

Since the metrics are collected on a module level, the first step in the DMA development was to determine the definition of a module for Ada. For the purpose of the design metrics collection and analysis, procedures, functions, tasks, packages (specifications and/or bodies), and generic procedures, functions and packages are considered modules. If both the specification and body are present for a package, then they are treated as one entity.

The next step involved matching the design metrics primitives to the constructs in Ada. The metrics specific to the design metrics calculations include the number of:

- modules called
- modules called by
- parameters passed to and from a module
- global variables referenced and modified
- complex data types (records, arrays, access types) accessed
- central calls (total number of module and task entry calls)

Some Ada constructs required special analysis when determining their relationship to the design metrics. For example, the *separate* construct, which allows large programs to be broken into small manageable pieces that can be developed and compiled independently, was examined. To measure the design dynamically, the DMA treats separate subprograms as though they are not separate.

Generic modules, which are templates that are instantiated to produce non-generic subprograms or packages, also required special handling. The DMA credits every call to an instantiation of a generic template as a call to the generic template. Similarly, the metrics required in the calculation of D_i are calculated on the generic template body. Therefore, generic instantiations do not have metric counts because their complexity lies in the generic template that they instantiate.

Tasks also required further examination. The DMA considers each entry call into a task as if the task were a subprogram being called with the entry's parameter list. Each of these calls adds to the inflow, outflow, and fan-in metrics for the task depending on the parameter list of the task entry. Metrics then are reported for the task.

Finally, in the case of subprograms that are renamed, metrics are tallied on the original version of the subprogram that has been renamed since the original and the new name both

refer to the same subprogram. The renaming convention provides a mechanism to use a more meaningful or shorter name for a subprogram. Renamed variables are handled in the same way.

One of the primitive components in the internal design metric D_i is the number of external device accesses. This metric is not included in the D_i calculation for Ada due to the difficulty in identifying standard Ada Input/Output (I/O) operations. However, calls to I/O libraries mimic module calls to user-defined modules. Therefore, the I/O metric will be captured in the central calls parameter.

The DMA tool does not require that Ada input files be in compiled order, and thus an entire system or subsystems of a developing product can be evaluated. However, the files must be syntactically correct. If a syntax error is detected, an error message is displayed and the tool halts. Syntax errors could produce incorrect metric values if the tool did not halt.

From the given input, the DMA generates a variety of reports in the categories of metrics reports, stress-point reports, and content information reports. These are listed in Table 3.

Table 3: Reports Generated by the DMA

Metrics Reports	Stress-Point Reports	Content Information Reports
design metrics	D_e	overloaded
detail	D_i	files analyzed
$D_e D_i D(G)$	$D(G)$	module information
design balance		call structure
		variable table
		set analysis
		venn

The metrics reports consist of the design metrics, detail, $D_e D_i D(G)$ and design balance reports. The design metrics report consists of a listing of the values of the primitive metrics required to compute the design metrics, D_e, D_i and $D(G)$ for each module. The detail report is a listing of metric values of seventy-five metrics for each module. These seventy-five metrics include various size metrics and counts of some syntactic constructs used, such as *if* and *case* arms. (The Appendix contains a complete listing of these seventy-five metrics.) The $D_e D_i D(G)$ report consists of each module name and its module type (package, procedure, function or task) followed by the metric values for D_e, D_i and $D(G)$. In addition, the files that contain the module body and specification are listed for each module. The design balance report gives the design balance for the system being evaluated.

The stress-point reports are the summaries of the analysis of each of the design metrics, D_e, D_i and $D(G)$. Each report contains a statistical view of the analysis by including the mean, standard deviation, maximum, minimum and stress-point cutoff values. These reports also contain the names of the modules that have been identified as stress points. If an overloaded module is identified as a stress point, the module's formal parameters are printed in addition to its name. These reports can aid managers and developers in their decisions of which modules should be redesigned or examined more closely.

The content information reports produced are called overloaded, files analyzed, module information, call structure, variable table, set analysis and venn. The overloaded report contains a listing of all the overloaded module names followed by their parameters. Overloaded modules are modules that have the same name and are nested in the same manner, but have a different number of parameters or parameters of different types. These modules are uniquely identified in the report by appending three underscores and a constant to the module name. This report informs the developers of the number of modules with the same name and same parentage and the differences in their parameters. The files analyzed report provides a reference point for a user of the DMA to validate that the appropriate files were used as input. Also included in this listing is a physical line count for each file. This report is helpful to organizations that have standards for the number of lines allowed per file.

The module information report is a textual summary of available information for each module. This report can be used in software reengineering and reverse engineering efforts or for documentation purposes. For example, this report includes variable scope and usage, modules called and called by for each module, and whether those modules called were included in the files submitted to the tool.

The call structure report is a textual representation of the call structure of the modules included in the analysis. Since partial or developing systems can be submitted to the tool, the call structure for each of the subsystems in the analysis is shown. Knowing the call structure is important to maintainers and developers as they try to enhance and fix errors in the software. In the report, subordinate modules are represented by indented notation.

The variable table report provides a listing of variables and their corresponding types. The modules in which the variables are declared, used and modified are also reported. This report is helpful to maintainers, testers and developers. For example, if it has been discovered that a variable is being changed incorrectly, developers and maintainers may reference the report to check the modules that access this variable.

The set analysis report is a listing of the modules that are identified as stress points by one, two or all three of the design metrics, D_e, D_i and $D(G)$. Managers or developers might want to only consider those modules identified as stress points by all three metrics for reexamination or redesign. Finally, the venn report is a postscript file that produces a Venn diagram that illustrates the information in the set analysis report.

Although the DMA tool only accepts Ada specifications and Ada code, the theory behind and the application of the design metrics applies to any design format or programming language. For example, our database of projects analyzed includes projects written in FORTRAN, C, C++, COBOL, Ada and Pascal. Keeping this in mind, the DMA tool was designed so that portions of the code could be reused.

The DMA is composed of two distinct parts, the parser and the analyzer (Figure 4). The DMA first executes the parser where the Ada files are parsed and evaluated, and metric data files and some of the reports are produced. The analyzer is invoked next to perform statistical analyses of the design metrics, and produce stress-point reports and a postscript file that graphically displays in a Venn diagram the number of modules highlighted by various combinations of D_e, D_i, and $D(G)$. The analyzer component takes as its input a metric data file produced by the parser, *dedidg.dma*. Thus, this component is independent of the language being analyzed and is reusable. Figure 4 is a diagram of the components of the DMA tool, with their inputs and reporting capabilities.

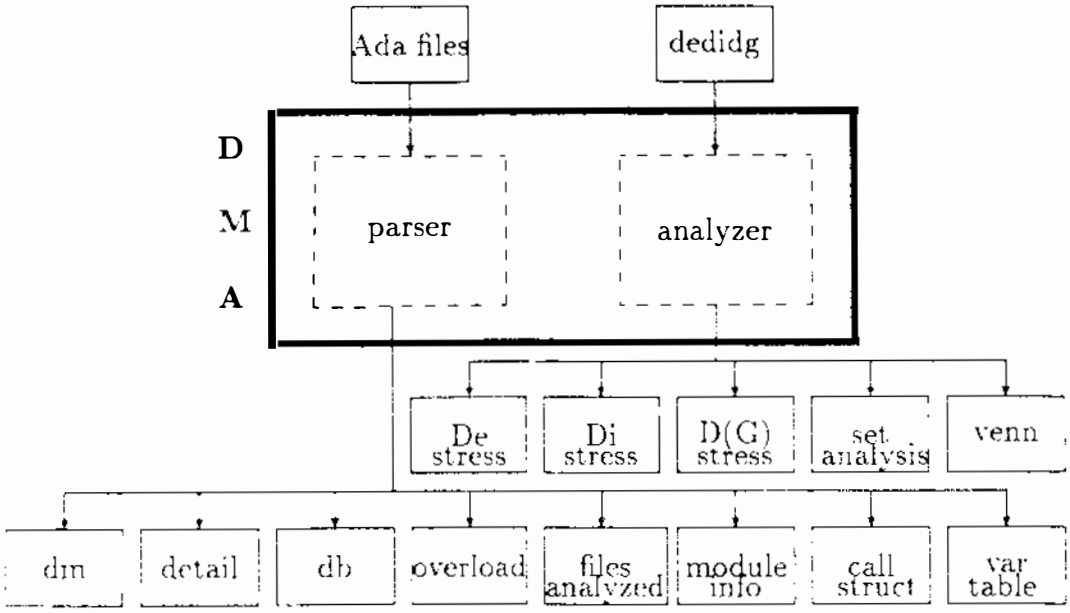


Figure 4: Schematic Diagram of I/O for the Design Metrics Analyzer

Unix programming tools lex and yacc were used in the construction of the parser component to aid in reusability. The Ada lex and yacc specifications can be replaced easily with the appropriate lex and yacc specifications for other languages. Lex and yacc were also chosen because they make it easy to identify syntactic constructs relevant to the calculation of the design metrics. Additional customized C subprograms are the backbone of the parser component and provide semantic meaning to these constructs. These C subprograms were also written and organized in such a way that routines handling Ada specific constructs could be replaced with routines handling constructs specific to another language.

Lex and yacc both work by reading a specification file and translating it to a file of C code, which is then compiled and linked with the rest of a program. For the DMA, *ada.l* and *ada.y* are the lex and yacc specification files, respectively. *Lex.yy.c* and *y.tab.c* are the C source code files produced by lex and yacc. These files then are compiled and linked with our C subprograms, *main.c*, *parse.c*, *process.c*, *print.c*, *disjoint.c* and *analyze.c*, producing the Design Metrics Analyzer. (See Figure 5.)

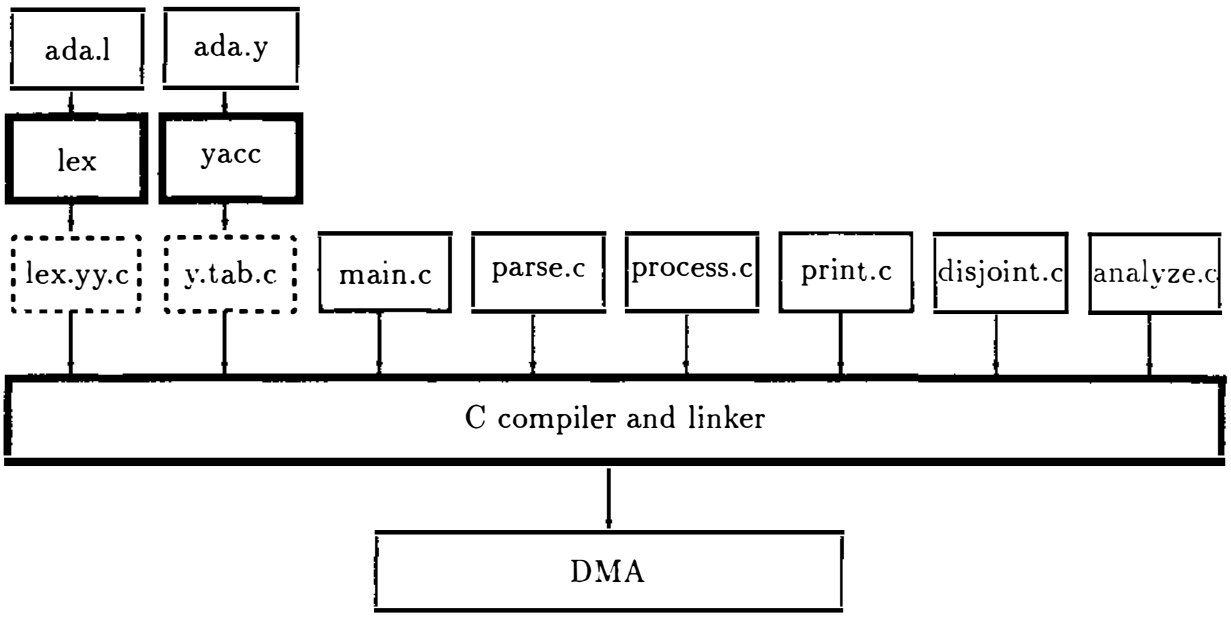


Figure 5: Files Required for the Construction of ada_dma_parser

Before this tool was available, the design metrics analysis of a 14,000 line Ada program required 420 person-hours of effort. The analysis of the same program requires less than two minutes using the DMA. This tool has been used on over one million lines of Ada. It is currently being used at Northrop/Grumman Electronics Systems Division, Computer Sciences Corporation, Harris Corporation, Northrop/Grumman Melbourne Systems and GTE Government Systems.

Current Design Metrics Research

In the SERC Design Metrics project, we validated our design metrics as predictors of error-prone modules on completed university and industry-based projects. In the SERC TSTADEIM project, we worked to identify time-slice transformations on architectural design metrics to determine the effects of design enhancements on a developing system. Using the metrics developed in the Design Metrics project and the insight as to how to apply the metrics gained from the TSTADEIM project, we are now determining if our metrics, when actually used during the design process to affect the design, lead to an improved software development process or product. The goal of this research is to evaluate the performance of our design metrics as they are applied to ongoing software development in controlled industry and university experiments.

We are providing a test of applicability of our internal and external software design metrics by applying those metrics to both completed and ongoing real-time development programs. In the first year of this project, we conducted two metrics studies at Northrop/Grumman Corporation's Electronics Systems Division. The programs in these studies were the QRC (Quick Response Capability) program, which is an infrared detector and counter-measures beam director system, and the 162 Update program, an enhancement of an

existing radio-frequency aircraft-servability jamming system. The design metrics analyses of both of these systems have identified stress points in the programs' designs, and have been enthusiastically received by the Northrop/Grumman managers responsible for those projects.

This year's research will focus on a design metrics analysis of another Northrop Ada project. This project's design schedule will allow design changes based on our metrics data. We are using the first integration phase, consisting of approximately 100,000 lines of Ada, to form the control system and a second integration phase, an additional 70,000 lines of Ada, will serve as the experimental system. Our approach permits us to first establish a baseline of design and implementation defect densities and their correlations to the design metrics prior to the introduction of these metrics into the second phase.

To further validate our model, we will analyze the results of our metrics applied to controlled experiments at Ball State to determine if the results from the industry study were indeed due to the quality of our metrics, were intrinsic to that application, or were simply coincidental. The design balance metric also will be evaluated to determine its relationship to error-prone software and its utility to designers in providing insight into the design process.

This project will provide the SERC affiliates with an analysis and evaluation of the design metrics applied to ongoing projects. We have the rare opportunity to apply the metrics, as they are theoretically intended to guide the design process, to SERC industrial projects as well as to university-based projects in this coming year. Based on the outcome of this analysis, we may refine and enhance the design metrics and our guidelines for applying design metrics to software designs.

Acknowledgements

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 No. ECD-8913133).

We thank Computer Sciences Corporation for the STANFINS data used in this research and Dale Gaumer at Magnavox Electronic Systems Company for his analysis of our design metrics applied to this project. We also thank Northrop/Grumman Electronics Systems Division for having the confidence in these design metrics to incorporate them into their software development methodology.

References

[AGRE92] Agresti, W.W., W.M. Evanco, M.C. Smith, D.R. Clarson, "An Approach to Software Quality Prediction From Ada Designs", Technical report RL-TR-92-315, Rome Laboratory, Griffiss AFB, New York, December 1992.

[MCCA76] McCabe, T.J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, SE-2, 4, 1976.

[ZAGE90] Zage, W. and D. Zage, "Relating Design Metrics to Software Quality: Some Empirical Results," Software Engineering Research Center Technical Report SERC-TR-74-P, May 1990.

[ZAGE92a] Zage, W., D. Zage, D. Gaumer and M. Bhargava, "Design and Code Metrics Through a DIANA Based Tool," *Lecture Notes in Computer Science, Ada: Moving Towards 2000*, J. van Katwijk editor, Springer-Verlag, pp. 60-71, June 1992.

[ZAGE92b] Zage, W., D. Zage, and M. Franck, "A Design Metric Evaluation of COBOL to Ada Code Conversion," *Proceedings of the Fourth Annual Software Quality Workshop*, Rome Laboratory, NY, August 1992.

[ZAGE93] Zage, W. and D. Zage, "Evaluating Design Metrics on Large-Scale Software," *IEEE Software*, July 1993.

Appendix: Metrics in detail.dma

1 Inflows
2 Outflows
3 Fan-in
4 Fan-out
5 Number of global data items flowing in
6 Number of global data items flowing out
7 D_e
8 Number of input parameters
9 Number of output parameters
10 Central calls
11 Data structure manipulations
12 Physical lines (blank + comment + code)
13 Comment lines (only comments on a line)
14 Blank lines
15 Attached comment lines (comments attached to a line of code)
16 NCNB (Number of Ada source code non-comment/non-blank lines.)
17 LTSC (Number of Ada source code Limited Terminal Semicolons determined by counting all semicolons EXCEPT those in comments, quoted strings, and formal parameter lists)
18 Number of variable declarations
19 Number of constant declarations
20 Number of deferred constant declarations
21 Number of integer number declarations
22 Number of real number declarations
23 Number of type declarations
24 Number of incomplete type declarations
25 Number of sub-type declarations
26 Number of package declarations
27 Number of package body declarations
28 Number of procedure declarations
29 Number of procedure body declarations
30 Number of function declarations
31 Number of function body declarations
32 Number of package rename declarations

33 Number of procedure rename declarations
34 Number of function rename declarations
35 Number of object rename declarations
36 Number of exception rename declarations
37 Number of generic package declarations
38 Number of generic procedure declarations
39 Number of generic function declarations
40 Number of package instantiations
41 Number of procedure instantiations
42 Number of function instantiations
43 Number of task declarations
44 Number of task body declarations
45 Number of task type declarations
46 Number of entry declarations
47 Number of exception declarations
48 Number of procedure subunits
49 Number of function subunits
50 Number of package subunits
51 Number of task subunits
52 Number of subprogram formal parameters
53 Number of generic formal parameters
54 Number of null statements
55 Number of assignment statements
56 Number of procedure call statements
57 Number of exit statements
58 Number of return statements
59 Number of goto statements
60 Number of entry call statements
61 Number of delay statements
62 Number of abort statements
63 Number of raise statements
64 Number of code statements
65 Number of if statements
66 Number of case statements
67 Number of loop statements
68 Number of block statements
69 Number of accept statements
70 Number of select statements
71 Number of conditional entry call statement
72 Number of timed entry call statement
73 Number of if arms
74 Number of case arms
75 Number of record fields

Pattern Languages:
A Way to Write the Software Architecture Handbook

Reusable software trades one set of burdens for another. As languages, libraries and frameworks grow in complexity, a designer will find that just properly reusing software requires more knowledge and experience than writing software from scratch did a few years ago. Even so, good designers eventually discover workable usage patterns for any package and can link these patterns together with others they already know. Finally acquired, a system of patterns allows a designer to wield complicated structures effortlessly while focusing full attention on the problems to be solved.

Can this knowledge be written down? A small community of framework designers thinks so and has been struggling to do so in a series of recent OOPSLA workshops. A handbook of (mostly) C++ design patterns will be out this fall and promises to change the way people think about that language. And, an international conference has been founded to create a literature of software patterns.

This talk will describe patterns in both their mental and written forms and explain how both will fit into the software development process of the future. Expect also a status report from the first patterns conference with pointers to organizations putting these principals to work today.

Ward Cunningham is a founder of Cunningham & Cunningham, Inc., a consulting company providing instruction, architecture, organization and components for object-oriented programming. Ward is well known for his contributions to programming practice such as the CRC Card design method and his work with Patterns. He recently served as program chair for PLoP '94, the first annual conference on the Pattern Languages of Programs.

Ward Cunningham
Cunningham & Cunningham, Inc.
7830 SW 40th Avenue
Portland, Oregon 97219
503-245-5633

The Role of Domain Analysis In Quality Assurance[1]

Michael F. Dunn

Industrial Software Technology

HC1, Box 93
Earlysville, VA 22936

Tel: (804) 978-7475
mfd3k@virginia.edu

John C. Knight

University of Virginia

Department of Computer Science
Thornton Hall
Charlottesville, VA 22903

Tel: (804) 982-2216
knight@virginia.edu

Abstract: Domain analysis, which refers to identifying and structuring information common to a family of software applications, is a key activity for organizations adopting reuse-oriented development methods. The benefits of domain analysis extend beyond simply supporting reuse, however. In this paper, we show how domain analysis can serve as an important technique for quality assurance by serving as the basis for software component certification.

We describe how domain analysis can be used to gather quality requirements for a family of systems. We then describe our approach to certification, ending with some preliminary results.

Keywords and Phrases: Reuse, domain analysis, certification.

Michael F. Dunn is an independent consultant specializing in domain analysis, re-engineering, and software quality assessment. He has been involved with software development, maintenance, and research projects with organizations as diverse as IBM, Sperry-Marine, Motorola, and the U.S. Army.

John C. Knight is a professor of computer science at the University of Virginia. His research interests are in the area of software engineering of high dependability systems.

1. Supported in part by the National Science Foundation under grant number CCR-9213427, in part by NASA under grant number NAG1-1123-FDP, and in part by Motorola.

1 Introduction

When a software engineer designs a new system, there is an overwhelming tendency to start the job with a clean slate. The field of software development is still sufficiently young that there are few references that one can use to guide the design process of most applications. As a result, applications tend to be one-of-a-kind products, with limited potential for extension or usage in other contexts. From a productivity standpoint, this means that time is wasted by constantly developing similar systems. From a quality standpoint, it means that the mistakes of the past are doomed to be repeated.

Contrast this with other engineering disciplines. For example, when a civil engineer sets out to design a new bridge, he has a body of experience to draw from that spans several thousand years. Although the location of the bridge might be unique, other crucial factors, such as material strength, maximum load, and expansion and contraction characteristics, are likely to be well understood and documented. The engineer's job is to assimilate these known physical properties and apply them to the needs of the current project. In effect, he is *reusing* a great deal of knowledge and a set of general models to aid him in the design process. The knowledge and models are associated with a collection of similar civil-engineering products, i.e., bridges.

The contrast with other engineering disciplines and the relatively low level of software productivity has fostered a great deal of interest in software reuse. Reuse of all work products, of processes, and of knowledge have been proposed as is common in other engineering disciplines. As well as productivity improvements, reuse is expected to improve software product quality also.

Domain analysis is a set of techniques used to gather the underlying information associated with a collection of similar software products, and is the first step undertaken by an organization in developing a reuse-based software development process. Domain analysis has been an active research topic in the software reuse community for the past several years, and researchers have proposed a number of analysis methods. While similar in their objectives, these methods differ in their complexity and focus. Although it is generally agreed that domain analysis is a necessary first step in implementing an organized software reuse process, what is often ignored is the overall quality improvements that an organization can gain from this process, with or even *without* software reuse. Specifically, domain analysis forces the organization to examine the problem area for which it is building software systems, and determine the quality criteria to which these systems must adhere in order to serve the needs of the domain.

If reuse is to be employed, however, and domain analysis is being performed as an initial step, the emphasis in existing domain analysis methods is to characterize the products being produced and to define a process to facilitate the development of such products. We claim that explicit quality criteria for products can and should be an output of domain analysis. Presently, such criteria are implicit in domain analysis work products. In addition, we claim that a critical yet usually overlooked output of domain analysis is *how* this will be achieved and what impact it has on the resulting development process and in the quality characteristics of reusable assets.

In the following sections, we explore domain analysis from the standpoint of how it can be used to support the process of assuring the quality of reusable software components. Further, we

describe a method by which one can draw inferences about the quality of a component-based system if the quality of each component has been well-established by a process known as *certification*. We begin by providing background on domain-analysis concepts, using two well-documented methods as examples. We then describe various software component certification techniques, and tie certification in with domain analysis by introducing a method called *domain-driven certification.* The concluding major section provides some detail about an on-going project being performed in cooperation with Motorola, Inc.

2 The Nature of Domain Analysis

Domain analysis differs from traditional systems analysis in that the intent of a systems analysis is to determine the requirements of a particular system, while the intent of a domain analysis is to determine the requirements of a *family* of related systems. Once determined, these requirements can then be used to:

- Determine variants of systems within the family.
- Determine a set of components that can be used to construct systems within this family.
- Ease future maintenance by providing a conceptual base by which to gauge the impact of modifications.

Domain analysis is a fundamental engineering activity. All branches of engineering rely on three aspects of domain analysis:

(1) Understanding the needs of the various groups of people involved in the creation and usage of the product, and how these needs might change in the future.

(2) Understanding what has been done in the past, so that previous knowledge and experience can be used in future products.

(3) Creating a development environment that enables the production of new products in the domain

Fig. 1 shows a high-level view of the domain analysis process. The input information, process, and output information are described in the following paragraphs.

Input Information

The information collected during domain analysis provides insight into the current state of the domain and how it is likely to change in the future. Unfortunately, this information is typically voluminous, unstructured, and difficult to assimilate. Included in this collection is undocumented expert knowledge, source code from existing systems, system documentation, and organizational strategy plans. Also, the most useful information, expert knowledge, is difficult to capture because doing so requires sophisticated knowledge acquisition techniques [BrB89]. Source code from existing systems can be used to determine the differences and commonalities between systems,

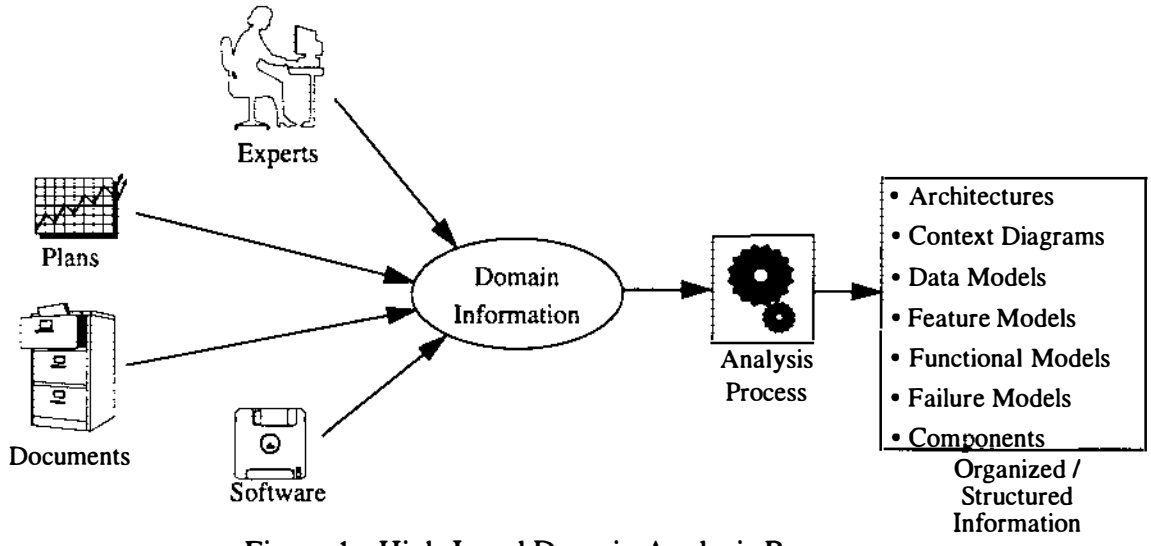


Figure 1 - High-Level Domain Analysis Process

which can suggest sets of reusable components. System documentation can be used to get an overview of how the system works, both from a user's and a programmer's perspective. Strategy plans can be used to determine future requirements. If the analyst knows that a set of systems will require a particular set of functions, he can use that information to derive a set of components to support these requirements.

Analysis Processes

Domain analysis is not one, but rather a set of complementary analysis processes, each one focused on a specific aspect of the domain. Commonly-used processes include:

- *Feature Analysis* for analyzing the services that the software provides from the perspective of the end user.
- *Functional Analysis* for analyzing the static and dynamic behavior of the system.
- *Architectural Analysis* for analyzing the various methods by which a set of related systems can be structured.
- *Failure Analysis* for analyzing the various failure modes of the systems and the impact that failures can have on the system's operational environment.

Output Work Products

The work products generated from the domain analysis process often resemble a generalized version of system specifications. Entity-relationship diagrams, structure charts, and state transition charts are all typical deliverables. These products are often parameterized, however, to factor out system-specific requirements.

These work products have two main uses. First, they provide a multi-view picture of the domain. This is analogous to a house blueprint. To keep the blueprint simple, house plans often

have separate diagrams to convey structural information, wiring, plumbing, and room layout. Combining these into one diagram results in picture that is busy and difficult to assimilate. The same principle holds in the case of domain analysis - one uses different work products to convey different information about the systems in the domain.

The second use of these work products is that they serve as generic templates to use in future system development. If the basic structure of these assets is present, much of the application-specific data can be filled in with specific requirements.

2.1 Domain Analysis Methods

A number of domain analysis methods have been developed in recent years. Two that have received a great deal of attention are those developed by the Software Productivity Consortium (SPC) and the Software Engineering Institute (SEI). We summarize both of these methods here and discuss them from the standpoint of how well they support the quality aspects of reuse. We introduce the general concepts associated with the enhancement of domain analysis to better serve aspects of the development process other than component development, aspects such as testing and maintenance. We refer to this enhanced process as *extended domain analysis.*

We stress that our descriptions of both the SPC and the SEI methods are necessarily very brief, and the reader is urged to consult comprehensive descriptions [SPC92, Kan90] for details.

Software Productivity Consortium

The SPC's method is based on identifying the characteristics common to a family of applications within a business area. The main work products that result from this method include a set of decision models, specification templates, and generic code modules. The method also devotes a great deal of attention to the process by which a family of systems is developed. Thus, an additional work product is a specification for domain-specific development environments to facilitate building such systems. In other words, not only does the method consider systems in a domain, it also considers the infrastructure (tools, components, and environments) to facilitate the creation of those systems.

As Fig. 2 shows, there are three high-level activities performed within the domain analysis process - *domain definition, domain specification,* and *domain verification.*

Domain definition is the process of determining the domain's scope, and characterizing the similarities and differences between systems in the domain. The term "scope" refers to an informal delineation of the set of systems that are included in the domain versus those that are not. The

Figure 2 - SPC Process - Major Phases

deliverables resulting from this stage include four items:

- A *domain synopsis* that describes the functionality of the systems in the domain, high-level design issues, characteristics of the operating environment, and types of work products needed for applications built in the domain.
- A *domain glossary* that establishes the vocabulary necessary to discuss systems within the domain.
- A set of *domain assumptions* that establish the characteristics which are common among or variable between systems in the domain, or which would exclude systems from being included in the domain.
- A description of the *domain's status* that describes the domain's current level of technical maturity, and its potential for change in the future.

Domain specification is a set of activities for describing the work products used to build systems in the domain, and the process for using these work products effectively. It includes:

- Generating *process requirements* that define the work products needed at various lifecycle phases of the development of the systems in the domain.
- Identifying *work product families* that are related applications which can use a given set of reusable parts.
- Creating a *decision model* that provides the decisions which the engineer must make when defining a particular member of a work product family.
- Creating a list of *product requirements* that specify, in abstract terms, the problems solved by families of work products within the domain.
- Developing a *product design* that describes the design of each work product family.

Domain verification is a set of activities for verifying that the implementation of the work products in the domain and the process for using these products match the requirements and designs produced during the domain specification phase.

The inputs to the domain analysis process cover three main areas:

- *Knowledge of the domain* that can be derived either from system users or developers. The information sources listed earlier, such as existing systems in the domain and system documentation, can also be used to derive this knowledge.
- *Knowledge of past and ongoing projects* that refers to understanding the subtleties of implementing systems in the domain. This includes understanding the implementation techniques used in those systems and problems that have occurred in previous development efforts.
- *A set of domain plans* that are a combination of two work products - a *domain evolution plan*, and a *domain development plan.*

  A domain evolution plan is analogous to an organizational strategy document. Its purpose is to determine the needs of near-term projects, and determine existing systems that are similar to these projects that might provide reusable components.

A domain development plan deals with organizing the resources needed to support reuse-based development in the domain. This includes determining the risks involved in each stage of the process, the objectives of each iteration of the development process, and the resources needed to accomplish each iteration.

Although the SPC method deals with quality issues in system family development, much of the relevant information is scattered across several sets of work products. For example, a number of work products provide information on component usage constraints, restrictions on allowable data values, and responses to undesired events. In addition, other quality factors, such as maintainability and portability, are captured to some degree by the decision models and variability assumptions generated by the method. However, the fact that this type of information does not reside in a centralized location can be a significant hindrance to the quality certification processes described later.

Software Engineering Institute

The SEI's method is referred to as *Feature-Oriented Domain Analysis* (FODA). It is based on analyzing the similarities and differences between the various features, functions and behaviors possessed by a sample set of systems in the domain. The work products include entity-relationship diagrams, feature models, and functional models. The SEI process follows the three stages shown in Fig. 3.

The first stage, *context analysis*, is much the same as the SPC's domain definition phase. The goal is to determine the relationship between the domain under study and other domains. This can be illustrated by means of a structure diagram that is a hierarchical chart showing the domains that subsume, or are subsumed by, the domain of interest. Data flow diagrams are then created to show how information flows to and from the domain of interest. By performing a context analysis, one can begin to quantify the variability between systems in the domain.

The second stage, development of a *domain* model, is a process of analyzing the services that applications in the domain are expected to provide, and the expected run-time environments of the applications. This stage is divided into three subprocesses:

- *Feature analysis* that describes from a user's perspective what services are provided by the software.
- *Entity-relationship modeling* that describes the data used by and produced by applications in the domain, and the relationships between these data.
- *Functional analysis* that describes the static and operational aspects of the application's functionality. Static aspects can be characterized by such techniques as data flow

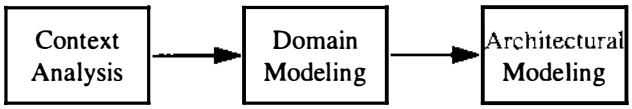


Figure 3 - SEI Process - Major Phases

diagrams. Operational aspects can be characterized by techniques such as state transition diagrams.

The third stage, development of an architectural model, is where high level designs of applications in the domain are created. The goal is to create an architectural model, which can thus be used to derive specific components.

As with the SPC method, required quality characteristics for systems in the domain are implicitly, rather than explicitly, specified. Feature modeling provides information on the services that are either required, optional, or mutually exclusive with other features in the domain. The entity relationship model gives an idea of required data and data formats. The functional model shows invalid or erroneous function states. And architectural modeling provides information on required portability characteristics. But as with the SPC model, all of this information is scattered across the set of output work products thereby complicating the certification process.

2.2 Extended Domain Analysis

While typical domain-analysis methods devote a great deal of attention to characterizing the functional and behavioral aspects of an application family, their focus on system quality characteristics is implicit rather than explicit. The intent of the *extended domain analysis* framework is to address this by identifying quality characteristics common across an application family and by determining the quality characteristics that must be present in a set of domain-specific components to support the system-level needs.

The term "extended" domain analysis is used not simply because it adds a new set of deliverables to the domain analysis process, but also because it uses as input the traditional set of deliverables. The process can be used on the outputs from any domain analysis method, as is shown in Fig. 4.

If we use the SEI process as an example, extended domain analysis uses the following process outputs to develop the required quality criteria:

- *Context analysis*

  Context analysis is useful in deriving maintainability, usability, and portability properties because it provides insight into the different user communities of the system and the different platforms on which applications are likely to be targeted.

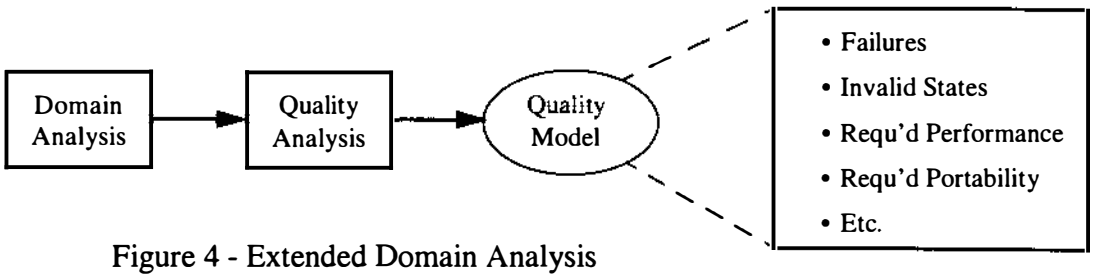


Figure 4 - Extended Domain Analysis

- *Feature analysis*

  Feature analysis enables the analyst to determine the potential usage characteristics of the application family, predict the types of faults that are likely to arise, and evaluate the impact these faults will have on the systems operational environment. Feature analysis is also important in gathering expected system performance characteristics.

- *Entity-relationship models*

  Entity-relationship models are useful in identifying the format and precision of data items shared between sets of components.

- *Functional analysis*

  Functional analysis is used to identify invalid or hazardous states of operation in the target application.

- *Architectural models*

  Architectural models enable the analyst to determine the interactions between the different components of the system, and enable him to determine how different faults in one component will ripple through and impact components in other parts of the system.

Thus, by performing an extended domain analysis, the engineer is paying attention explicitly to the quality requirements of products in the domain and generating outputs from the process that describe required characteristics such as functional correctness, portability, maintainability, and documentation in substantial detail. The first significant *additional* output generated by an extended domain analysis is, therefore, a list of the required quality criteria for products in the domain.

These criteria can be organized to show the specific relationship between the major functional areas in the domain as generated by the SEI's functional analysis process, and a categorized set of quality criteria. As an example, consider the domain of product shipping scheduling systems. Suppose that we have discerned the following major functional areas:

- *Customer management* that includes the entry of customer address and order information.
- *Transportation management* that includes information on the locations of warehouses, airports, and railway and truck terminals. It might also include information on transit times, costs, and schedules.
- *Schedule optimization* that takes customer order information and transportation information, and produces an optimal delivery route.
- *Report generation* that allows the user to generate reports and perhaps do "what-ifs".

For each of these functional areas, one would use the information contained in the domain analysis to produce the quality criteria to which each function must adhere. These criteria can be organized according to the quality classification scheme developed under ISO Standard 9126 that includes the categories of functionality, reliability, usability, efficiency, maintainability, and portability [Vol93].

3 Quality Properties and Certification

If a set of related systems in a domain is required to possess a common set of quality characteristics, it is reasonable to expect that the reuse-based process for developing these products should help to ensure that they have these required properties. If this were possible, it would constitute a second area of productivity gain attributable to reuse - rather than reuse just enhancing productivity through faster creation of work products, it would help reduce costs in areas such as testing.

How might this be achieved? The approaches we advocate for achieving this goal are based on the use of components that adhere to rigorously-established, domain-specific quality characteristics. If it can be shown in an unambiguous manner that *system-level* properties hold as a result of using such components, then we have taken an important step towards establishing the quality of these systems. For example, if we know that a set of components has been checked to assure freedom from memory leakage, and we know where these components are being used within a system, then we will not have to include those parts of the system when we perform system-level testing for memory leakage. The process of establishing such component quality characteristics is called *certification*, and it has become a major issue in reusable software development in recent years for two reasons:

- The increasing acceptance of the idea, if not the practice, of reuse as a development method, and the growing number of tools and libraries to support it.
- The unevenness of the quality of components currently available to the reuser.

The availability of a set of trustworthy "plug and play" components will help to make reuse a standard practice. The purpose of certification is to provide a consistent set of criteria by which such components can be judged. More importantly, once the engineer knows that a component possesses a particular set of characteristics, it might be possible for him to draw inferences about the systems that use these components.

In this section we discuss three certification schemes, *levelled*, *scored*, and *statistical* that have appeared in the literature, and introduce the *domain-driven* approach developed by the authors.

Levelled Schemes

In a levelled scheme, components are assigned quality "levels" according to the amount or type of quality assurance processing that has been applied to them. Components assigned to low certification levels have had little or no quality assurance applied to them, while those assigned to a high level have typically undergone some form of prescribed testing, inspection, and/or analysis, sometimes by a third party. There is no upper limit on the number of levels that can be used in such a scheme. However, most schemes currently in use include only four or five levels.

In a variation on levelled schemes, each level is divided into sublevels. The levels are often referred to as "primary levels" and the sublevels are referred to as "secondary levels". As in the basic levelled scheme, different primary levels correspond to the different cumulative quality

criteria being used. Secondary levels refer to the amount of confidence the reuser should have that the component has achieved the primary level, based on the quality assurance processes that have been applied to the component. Additional information on levelled certification schemes is available [PPS92, REW93].

Scored Schemes

In a scored scheme, scores are assigned to a component based on a weighted set of quality criteria. Criteria can be determined by the organization based on desired domain properties. Many variations on this scheme focus on reusability-enhancing characteristics such as module complexity or number of parameters in each function. However, other quantifiable properties, such as average execution time over a given input sample, can also be used as criteria. A component's overall score is a composite of its scores in each quality area. The goal is to enable the user, if he is presented with a set of similar assets, to choose the asset with the highest overall rank.

Statistical Schemes

Statistical schemes address only one major aspect of system quality: overall reliability. The goal is to determine the reliability of each component, and then infer the reliability of the system based on a composite of each component's reliability.

Statistical schemes are based on an understanding of system usage characteristics. The certifier creates a model of system execution characteristics that enable him to make an educated guess about which parts of the system are most likely to be executed and the likelihood that execution of one part of the system will be followed by execution of another. By understanding the frequency of execution of the different parts of the system, and the transitions between these parts, the certifier can begin to create a Markov model that captures these transitions. Using this model the engineer can assess the reliability required in each part of the system in order to meet an overall quality goal for the entire system.

The main certification criteria is mean time to failure. There are two ways of looking at this value. The first way is time-based, and is defined as the average length of time between manifestations of system faults. The second way is event based, and is defined as the probability of a fault occurring while processing a given input value.

Domain-Driven Scheme

The methods described above each give a measure of confidence that a set of components has undergone some standard process of quality assurance. But each suffers from the same set of weaknesses:

- *Lack of tailorability* - The schemes are rigid in their definitions. A levelled scheme, for example, defines what the levels will mean and libraries certified to a given level must comply with that definition. This approach fails to address the needs of different organizations or domains.

- *Loss of information* - In each method, a large volume of information is compressed into a single number, making it difficult to determine what factors went into the quality assessment process.
- *Lack of ability to draw quality inferences* - As noted above, given that we have a set of components whose quality properties are well-defined, we would like to be able to use this information to understand the quality properties of the systems using the components. The methods described above do not support this ability.

We summarize domain-driven certification in this section but omit substantial detail for brevity. The interested reader is referred to the authors previous publications on the subject [Kni92, DuK93].

Domain-driven certification addresses the three problem areas outlined above by making explicit the criteria by which a set of components is evaluated. In fact, this is the definition of component certification:

Definition: A reusable component is defined to be certified if it possesses a prescribed set of properties.

Notice that this definition is arbitrarily flexible. Different definitions can be established for different organizations and domains. Notice also, that the quality criteria possessed by a component are listed explicitly thus permitting engineers to understand completely what qualities a component possesses. Finally, notice that the properties can and *will* be tailored to permit their exploitation in the demonstration of domain quality properties in systems built with the certified components.

The ability to exploit component properties is an extremely important point. The set of properties that are included in a certification definition is defined to be that set of qualities that, if present, would facilitate the demonstration of required domain quality properties in systems - nothing else. Thus, only component properties that can be exploited to do something useful, i.e., show that a system possesses domain quality properties, are included in the certification definition.

Once the certification definition is in place and libraries of components have been shown to meet the criteria, it is possible to develop systems using those parts and establish domain properties relatively easily by depending on the component properties.

The domain-driven certification process can be summarized as follows:

- Perform an extended domain analysis and, using its output, determine the necessary quality criteria for products in the domain.
- From the *domain* quality properties determine the set of qualities that should be held by *components* in libraries used to build products in that domain.
- Develop systems in the domain of interest using a reuse-base development process and exploit the component certification properties to establish domain properties of the systems.

Finally, it is clear that domain analysis in its extended form as we define it is playing a critical role in the whole certification process and is in fact an essential contributor to system quality. Fig. 5 shows how domain analysis and certification fit together in this scheme.

4 Experience

For the past two years, we have been experimenting with these ideas on certification in cooperation with the Cellular Tools department of Motorola, Inc. In this section, we provide some preliminary results of this work. A more detailed explanation can be found elsewhere [Dun93].

4.1 Problem Domain

The Cellular Tools department is responsible for developing software for simulating and testing the digital switches Motorola manufactures for routing and managing cellular phone calls. This domain is useful for validating the concepts of domain-driven certification because it possesses two important characteristics:

- The software created in the domain is intended to automate expensive processes. The

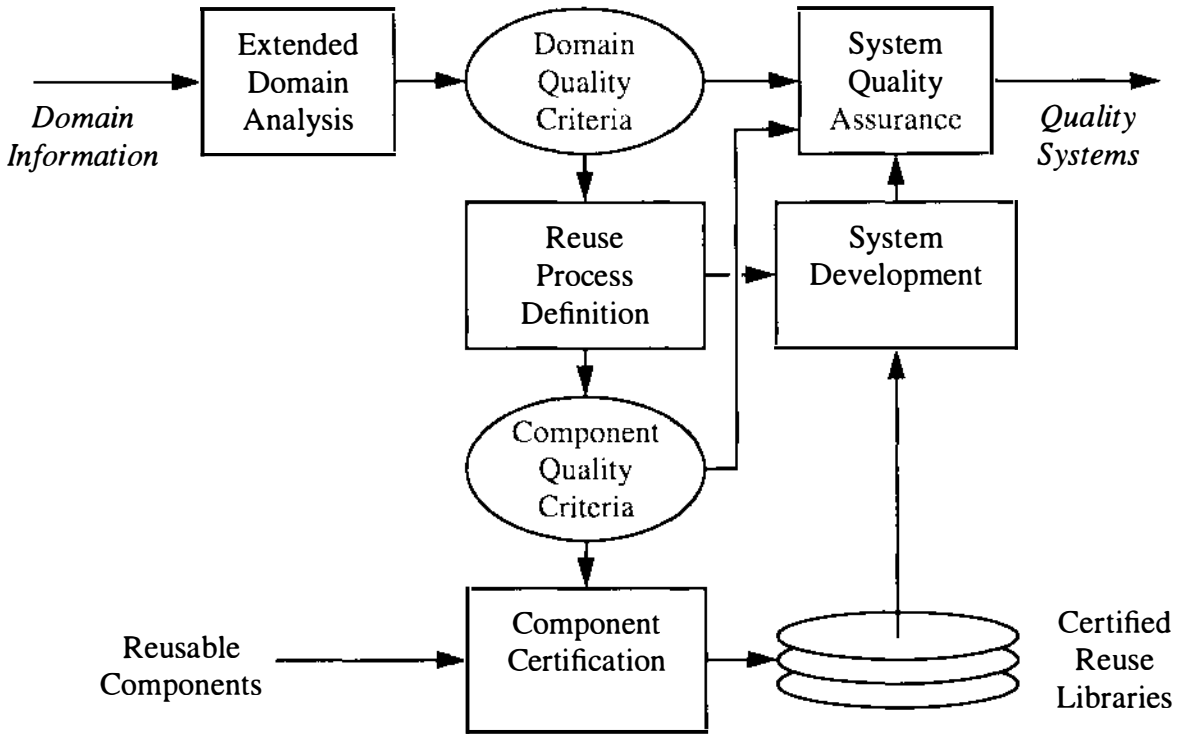


Figure 5 - Domain Analysis and Certification

switches under test are quite costly, and making the test process more efficient means possibly reduced development resources.

- The domain itself requires high dependability. As telephone switches have become controlled by software, software flaws have become the most likely source of failures. A single failure in a single switch can mean that calls cannot be placed, which might cause significant and serious inconvenience. Further, faults can sometimes lead to a cascade effect, causing an entire switching network to fail. The software thus must be submitted to rigorous testing procedures. This includes testing each feature of the switch, testing an integrated set of features, and testing how the switch reacts to different combinations and load-levels of calls.

4.2 Analysis Process

Fig. 6 shows the process we used for performing our extended domain analysis. The process consisted primarily of meetings with domain experts to determine the scope and content of the testing domain. It also included analyzing system source code, requirements and designs documents, and development strategy documents. Also included were these activities:

- *Analysis of the organization's mission.*

  This yielded a description of the services that the organization is expected to provide to its customer set. Included is a set of goals, and for each goal, sets of assets and processes required to achieve that goal are described.

- *Analysis of the organization's strategy.*

  This yielded a description of how the organization needs to prepare for future changes

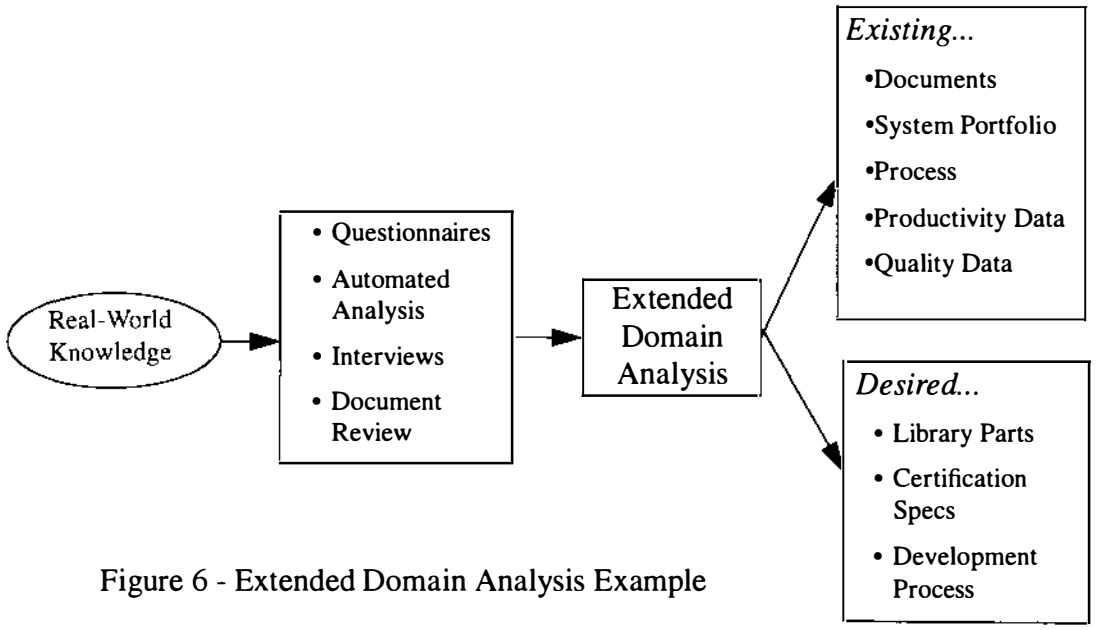


Figure 6 - Extended Domain Analysis Example

to its product line, supporting technology, and operational environment. Associated with each change is a set of required assets and processes.

- *Analysis of production goals.*

  This yielded a description of the organization's production goals, the means by which these goals should be measured, and the assets and processes required to support them.

- *Analysis of the technological infrastructure.*

  This yielded a mapping between lifecycle phases, lifecycle work products, and development tools. The purpose is to assess how well the organization's current technology base supports its development process.

- *Analysis of subdomains.*

  This yielded a breakdown of the domain into its subdomains. For each subdomain, we identified required functionality and implementation models. For each implementation model, we identified design trade-off's, example systems, possible failures, and supporting sets of assets.

4.3 Initial Results

The main result of the work to date is a preliminary set of quality certification criteria for six subdomains within the message protocol verification domain. For each of these subdomains, we derived properties according to the six-category ISO 9126 quality classification scheme: functionality, portability, maintainability, performance, dependability and usability

As an example, one of the organization's goals was to make a subset of their systems cross-compatible between their Sun platform running Unix, and their PC platform running DOS. This had obvious ramifications on the portability characteristics of the components, since it prohibited them from using platform-specific constructs without adequate parameterization.

A side-effect of performing this process was that it forced the engineering team to identify and prioritize the organization's quality and productivity goals. This is a direct result of identifying which components would have the greatest impact on the development of new systems in the domain and determining what their quality attributes should be.

Additionally, the process caused the engineering staff to come to an agreement about the meaning of certain commonly-used terms within the domain. Engineers will often use the same word or phrase to mean different things, and, clearly, this can inhibit using work products across different projects.

5 Conclusions

We have presented an overview of domain analysis and concluded the traditional definition of domain analysis does not highlight domain quality considerations to the degree needed by a rigorous certification process. We consider this to be essential if reuse is to pay off as a productivity vehicle and we have defined the concept of extended domain analysis as a result.

To support the goal of facilitating the development of systems possessing significant quality characteristics, we have established a precise meaning for component certification and shown how the definition and extended domain analysis work together to provide a framework for developing high quality systems. We also reported our preliminary experience with extended domain analysis.

As often happens, more questions were raised than answered during the course of the project. Among the questions raised are

- How can the knowledge acquisition process be streamlined?
- How can the domain-driven certification process best be used by organizations at different levels of process maturity?
- What types of tool support will best facilitate the domain analysis and certification process?
- How should the certification properties be organized?
- How do the various quality standards proposed by such organizations as IEEE or ISO fit into our framework?

Acknowledgments

It is a pleasure to acknowledge many useful discussions about the subject matter of this paper with Allan Willey of Motorola. This work was sponsored in part by the National Science Foundation under grant number CCR-9213427, in part by NASA under grant number NAG1-1123-FDP, and in part by Motorola.

References

[BrB89] Brule, J.F., and A. Blount., *Knowledge Acquisition*, McGraw-Hill, Inc. New York, NY, 1989.

[Dun93] Dunn, M.F., *Cellular Tools Extended Domain Analysis*, Univ. of Virginia CS Tech. Report CS-93-50, August, 1993.

[DuK93] Dunn, M.F. and J.C. Knight, *Certification of Reusable Software Parts*, Technical Report CS-93-41, Department of Computer Science, University of Virginia, 1993.

[Kan90] Kang, K.C., S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson, *Feature-Oriented Domain Analysis Feasibility Study: Interim Report*, SEI Tech. Report CMU/SEI-90-TR-21, August 1990.

[Kni92] Knight, J.C., *Definition and Framework for the Certification of Reusable Parts*, Proceedings International Software Quality Exchange '92, San Francisco, CA 1992.

[PPS92] Poore, J.H., T. Pepin, M. Sitaraman, F. L. Van Scoy, *Criteria and Implementation Procedures for Evaluation of Reusable Software Engineering Assets*, CDRL S45.11, March 28, 1992.

[REW93] *Proceedings of the Second Annual West Virginia Reuse Education and Training Workshop*. October, 1993.

[SPC92] *Domain Engineering Guidebook*, SPC Tech. Report SPC-92019-CMC, December 1992.

[Vol93] Vollman, T.E., *Software Quality Assessment and Standards*, IEEE Computer, Vol. 26, No. 6, June 1993.

Using the Pareto Principle to Allocate Scarce Testing Resources

A.A. "Tony" Templeton

IBM Consulting Group
3600 Steeles Ave. East C2/569
Markham, Ontario, Canada L3R 9Z7
aat@vnet.ibm.com

Abstract

There is now virtual consensus both in academe and industry that software quality can not be "tested in" at the tail end of the development process. Rather, the investment in quality must begin at a project's outset. Nevertheless, as a consultant who is regularly called upon to perform project reviews, the author continues to find situations where this investment has not been made, where it is late in the project life-cycle, and where code quality is indeed suspect. In these circumstances, client resources are often severely constrained (i.e., short of time, short of money), and customary options such as adding testing resources or extending the testing schedule are not practical. The question then arises, "What other options exist?". One such option is to apply the Pareto Principle (sometimes called the 80/20 rule) to target specific areas of the application for focused or *directed* testing. The issue then becomes one of determining whether the Pareto Principle can be applied to a given project, and if so, how to allocate the test resources to minimize risk. The author explores this topic in the context of his project review experiences.

Biographical: Tony is a senior consultant with the Application Solutions Consulting Practice in IBM Canada. His primary area of expertise lies in "project management" consulting with particular emphasis on assisting clients to meet their software quality and risk management objectives. Tony has 25 years experience in building applications and application-enabling software. He began his career as a PL/1 programmer has held many line and staff management positions in several software development organizations within IBM. Through the years, he has worked to understand better the seeming conflict between meeting schedule and quality commitments, and recently published an article on the topic in the June 22, 1994 edition of Computing Canada. Tony has an Hons. BSc. in Mathematics from Queens University, and an MBA from the University of Toronto.

Introduction

Fifteen years have passed since Glen Myers wrote his oft-quoted book, "The Art of Software Testing" [MYE79]. In the preface, indeed in the very first line of the preface, Myers writes, "It has been known for some time that, in a typical programming project, approximately 50% of the elapsed time and over 50% of the total cost are expended in testing". Statistics accumulated in the years since then, continue to bear out Myers' observation; for example, Richardson [RIC93] affirms that testing typically consumes 30% - 60% of the development effort.

How is it then, in the face of such accumulated evidence, that one can be brought into a review of a $90 million mission-critical development project, and find that the project budget allows a mere 15% for testing? What meaningful advice can one give a client in such circumstances to help salvage the situation, given that the client has no more money to put into the project, and the project is, in any case, months behind schedule?

This is not an isolated occurrence. Despite years of academic progress in turning "the Art" of Software Testing into "the Science of Software Quality", the commercial world apparently has a lot of catching up to do. By way of explanation, Humphrey [HUM89] has suggested a hierarchy of five levels of organizational sophistication, called the Capability Maturity Model, which helps explain how and why it is that even well managed systems organizations find it difficult to escape development mediocrity. There are exceptions of course, but the exceptions aren't often the ones asking consultants to conduct project reviews.

It was against this backdrop of chronic, and pervasive, under-estimating of testing effort, that I became interested in searching for new options. I was looking for the kind of advice that might help clients turn their hopeless situations into ones which were merely less desperate. A partial answer suggested itself one day when a client asked if there wasn't some kind of "80/20 rule for testing" ... a way to beat the odds and get 80% of the bugs out for only 20% of the effort.

Pareto's Law

At the time I had no answer. However, with this question in mind, I began to notice how often, in reading the testing literature, one came upon phrases suggesting that, at least in some circumstances, a small percentage of the code apparently contained a disproportionately large number of the defects. Relying upon statistics gathered by studying large software systems at such major corporations as IBM, DEC, and AT&T, Jones states, "As a rule of thumb, about 5% of the modules in a large system may receive almost 50% of the defect reports" [JON91]. This observation echoes Myers who, years earlier, reported the example of a large operating system for which "47% of the (defects) were associated with only 4% of the modules [MYE79].

Is it possible to capitalize on this "clumping" effect. Intuitively, one senses that testing efficiency would be enhanced if there were a way to determine, in each software situation, whether such a subset of defect-prone modules existed, and if so, where, specifically, they might be located. In other words, can we indeed put the 80/20 rule to work.

The expression "80/20" is, of course, the familiar name given to a concept developed in the nineteenth century, by the Italian economist, Vilfredo Pareto, who remarked upon the phenomenon in which a small percentage of a population accounts for a large percentage of a particular characteristic of that population.

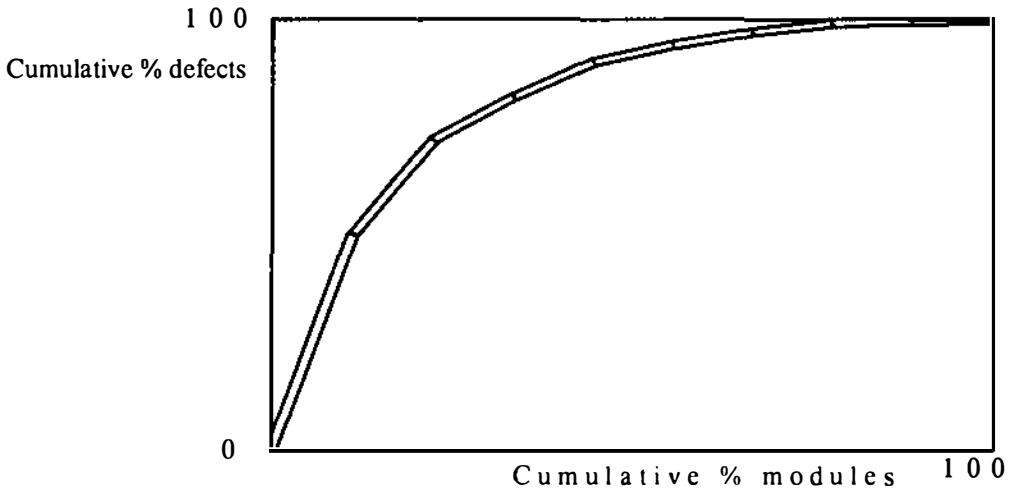


Fig 1. Example of a Pareto Distribution

Fig. 1 is an example, in graphical terms, of a defect distribution following the Pareto Principle (also known as Pareto's Law).

Pareto's Law distinguishes the vital few from the trivial many. The key to benefiting from Pareto's Law is to recognize its applicability to a given situation, and to act accordingly. Abdel-Hamid and Madnick point out that "as the error density increases, the distribution of errors among the system's modules generally also increase; errors become less localized" [AHM91]. In other words, the more bugs there are to be found, the more widespread they are likely to be. In such a circumstance, Pareto's Law is unlikely to help since, with so many modules at risk, targeting a few for specific testing attention would not materially increase the overall system quality.

Applicability Criteria

Clearly one of the criteria to be used then in deciding on the applicability of Pareto's Law is an assessment of how effective or ineffective prior White-Box testing has been at whittling down the defects.

To this end, Beizer estimates that up to 65% of defects can be caught by unit-testing [BEI90]. Conversely (and supportively), Jones suggests only 30% of defects will be found through function testing [JON91]. Taken together and re-framed slightly, these statements suggest that the relative efficiency of White-Box testing versus Black-Box testing is about 2:1. Therefore, when reviewing a project which is in the latter stages of its life-cycle, i.e., with unit-testing and

integration-testing essentially complete, it is crucial for the consultant to determine how well this prior testing was done.

Generally, we find that although their understanding of current theory on White-Box testing is weak, development organizations have nevertheless benefited greatly from the tremendous number of automated testing tools commercially available. In 1987, DeMillo & McCracken compiled a comprehensive list of the test tools then available [DEM87], and the list certainly has not stopped growing. So while application developers may not generally appreciate the nuances of White-Box testing theory, they nevertheless benefit as these advances are incorporated into the tools and methods they use. For example, from Japan, Cusho reports on a technique for determining which subset of all possible paths are the critical paths to be tested, and how to minimize redundant path test data. His theoretical technique is implemented in a systematic method for generating test cases, called AGENT [CUS89].

If the client has used such White-Box test tools to conduct unit-testing, then the probability is that the defect density will have been reduced to the threshold where Pareto's law may be applied to assist with Black-Box testing.

A second important criteria for deciding upon the applicability of this approach rests with the nature of the application itself. Increasingly many corporate systems are 7/24, i.e. operational around the clock, seven days a week and twenty-four hours per day. These so called life-blood systems are clearly not good candidates for testing short-cuts. At issue is the question of how reliable the software must be, and hence the need for a metric one might call the Target Level of Quality or TLoQ.

In the kinds of difficult project situations I am describing here, the client is almost always prepared to work with a narrow definition of quality focused on achieving software reliability. It is understood this may mean that other types of quality issues will remain undiscovered until after implementation (refer Yasuda) [YAS89]. In this context, we define TLoQ to be some metric which correlates to reliability.

Jones reports a correlation between defect levels and reliability such that a defect rate of 5 or more defects per thousand lines of code (KLOC) gives an approximate Mean Time To Failure (MTTF) of only a few hours, while improving the TLoQ to 1-2 defects per KLOC results in a 1 week MTTF maximum [JON91]. Software which is critical to the business must aim for a still higher TLoQ. For example, in compiler development, the goal is to have less than 0.1 defect per KLOC. To achieve such a high TLoQ, every module and function must be quality-assured and thoroughly tested. There is no room here for picking and choosing which pieces of the code to test. Still, there are many types of commercial applications which remain as candidates for this approach.

Locating the Clumps

If defects do tend to "clump", how do we find them? Several authors [AHM91] [BEI90] [BRO74] [HUM89] [JON91], have suggested factors which might cause clumping. These are listed below (there is no rank order implied although we perhaps have our personal favorites).

1. *Schedule pressure* Determine whether any project staff have been working excessive overtime...a likely indicator of severe schedule pressure. Determine which modules they are responsible for, and target these for extra testing.

2. *Differences in programmer ability* Determine who among the programmers are the least experienced...select samples of their work for further probing.

3. *Inadequate unit test* Look for evidence of inferior unit testing. Such evidence might be in the form of work plans which appear unrealistic, or inferior documentation describing the combinations of inputs and outputs which have been tested

4. *Excessive size and complexity* Look for modules which violate shop standards (for example, modules which are too big, or which have been patched and re-patched).

5. *Lynch-pin modules* In every application, there are a few modules which are at the absolute core of the system. For example, in a Mortgage application, the module which calculates the monthly interest would be one such.

6. *Data-Mapping modules* Pay particular attention to data conversion programs whose purpose is to map old data elements onto new ones. These programs can either be in the form of one-time conversion programs, or programs creating interfaces to existing systems.

Probing the Clumps

On a large project, the number of modules identified in this way is still likely to be too many to handle given the cost and schedule constraints.. So a further sieve must be applied to reduce the candidates to a more manageable number. This is accomplished through "probing".

For the on-line (GUI) portion of the system, assign a user test representative to "sample the goods". Similarly assign a programming tester to check out the batch component. If the probing finds no defects, move on. If on the other hand, some defects come to light, test with a vengeance the functions involved. Finding defects is like finding a quarter on the street ...if you find one, look for more, someone probably had a hole in his pocket!

An Example

As an example consider the real, but somewhat disguised, case of a new Customer Information (CIS) and Billing System for a public utility. The CIS had been under development for some time, and had already experienced two major plan resets. Almost every current system was replaced or changed by the new CIS which handled everything from customer billings, to wireless dispatching. The size of the application was thought to be (no-one knew for sure) 8-10 million lines of COBOL. With two previous schedule slippages already behind them, the client was strongly motivated to meet the current plan, but was concerned that there was not enough resource available to thoroughly test the application.

The client's team had done much of the development using CASE technology and code generators, but had been forced to write a substantial amount of native COBOL to create numerous interfaces between the various applications, as well as to a number of existing applications. The client was concerned not only with the quality of the code, but also with the quality of the data. It was estimated that about 1000 existing data elements were being replaced by 5000 data elements. (Recall Beizer's 3rd Law: Code migrates to data. Beizer estimates that 25% of the defects are in the data. [BEI90])

The client was flexible on cut-over strategy, and agreed that rather than going "Big Bang", it would be acceptable to implement in geographic stages, and to precede the staging with a pilot. The client further agreed that, given the staged implementation, the risk associated with adopting an 80/20 testing strategy was preferable to another schedule slip.

The strategy was two-pronged. First, it was agreed that a large representative set of transactions and data would be tested using an automated test tool (such as TPNS). This automated testing is the machine equivalent of having many users on the system. (Brooks observes, "more users find more bugs". [BRO74]) The purpose of this testing was to exercise the routine side of the business. It was estimated that, as soon as the testdata was available, this could be completed within weeks.

Secondly, the client agreed to bring a number of talented user and programming staff into a central independent test department. (The DeMillo survey recorded a consensus that "integration testing requires a few key people ... your best people". [DEM87])

The individuals to be selected would be expected to have exceptional knowledge of the current business and its supporting information systems.

These individuals would work to identify the subset of functions and modules thought to be at risk, then probe them for defects. If defects were found in a given area,, they would direct their testing efforts into a concerted effort to "break the code". The focus of these test efforts would be to find and fix only the most severe problems (Severity 1 and 2). Lesser defects were to be noted and set aside for correction some time after the pilot implementation. As this paper is being written, the client is optimistic about the eventual success of the strategy, and is rebuilding the work plans to reflect this approach.

Conclusion

Many authors over the years have noted the uneven distribution, or clumping, of defects, particularly in large software projects. In certain circumstances, when a project is in difficulty, and it is too late to establish a proper Quality Assurance and Full Life-cycle Testing foundation, Pareto's Law provides a rationale for a sharply focused test approach which may make it possible to approach the desired Target Level of Quality. This "Directed Testing" which concentrates on a small subset of the overall application complements other forms of testing, particularly those which use automated volume-test tools.

References

[AHM91] - T. Abdel-Hamid and S. Madnick. Software Project Dynamics. pp 101,113 Prentice Hall, 1991.

[BEI90] - B. Beizer. Software Testing Techniques. pp 39,90,439 Van Nostrand Reinhold, 1990.

[BRO74] - F. Brooks. The Mythical Man-month. pp 30,121 Addison-Wesley, 1974.

[CUS89] - T. Cusho. "Functional Testing and Structural Testing". pp 155,156 from Japanese Perspectives in Software Engineering. Matsumoto & Ohno [Ed]. Addison Wesley, 1989.

[DEM87] - Demillo, McCracken, Martin, and Passafiume. Software Testing and Evaluation. pp 204 Benjamin/Cummings Pub, 1987.

[HUM89] - W. Humphrey. . Managing the Software Process. pp 195,196 Addison-Wesley, 1989.

[JON91] - C. Jones Applied Software Measurement. pp 269,281,282,288. McGraw Hill, 1991.

[MYE79] - G. Myers. The Art of Software Testing. pp 15,16 Wiley InterScience, 1974.

[RIC93] - D. Richardson. A Tutorial on Software Testing, pp 17 SIGSOFT'93: Foundations of Software Engineering. Publication of the ACM, 1993.

[YAS89] - K. Yasuda. "Software Quality Assurance Activities in Japan". pp 188 from Japanese Perspectives in Software Engineering. Matsumoto and Ohno [Ed.] Addison Wesley, 1989.

THREE SOFTWARE TESTING TECHNIQUES AND THEIR EFFECTIVENESS FOR A FOURTH GENERATION LANGUAGE

Kimberlyn C. Mousseau
Idaho National Engineering Laboratory
P.O. Box 1625
Idaho Falls, Idaho 83415-3730

William S. Junk
Computer Science Department
University of Idaho
Moscow, ID 83844-1010

ABSTRACT

Despite the fact that many software testing techniques exist and their effectiveness extensively studied for applications developed using third generation languages, their effectiveness for applications written in fourth generation languages is basically unknown. In practice, testing of fourth generation language applications is often approached in an ad hoc manner. The study reported here applied three software testing techniques used extensively for third generation languages and applied them to two systems developed in ORACLE – a fourth generation language. The three techniques were statistical testing, data-flow testing, and syntax testing.

All three test methods were applied after the two ORACLE systems had been tested by the developing engineers and the systems were considered ready for release. The three testing techniques found several errors in each system that were not detected during the initial system test and showed the importance of developing and applying a systematic testing strategy. With the statistical testing method we were also able to estimate the number of remaining modules containing errors.

1.0 Introduction

The number of systems developed in fourth generation languages has risen steadily over the past ten years, yet little research has been conducted in the area of test methods for these systems. With this situation prevailing, we investigated the effectiveness of software testing methods previously applied to third generation language applications by applying these methods to applications written in a fourth generation language. Three software testing techniques widely used in testing third generation language applications were selected and applied to two applications developed in ORACLE. The three techniques evaluated were:

- syntax testing,
- data-flow testing, and
- statistical testing.

ORACLE is a fourth generation language based around a relational database facility. It provides Standard Query Language (a standard query language commonly known as SQL), a report generation package, and a package to assist in screen development for data entry. ORACLE provides many internal, preprogrammed routines so that the programmer can concentrate on defining what these programs are to control and how the user will interact with the application. To accomplish certain unique operations, the programmer may be required to produce application specific routines that do not exist as part of the internal library. With fourth generation languages, many of the implementation details are hidden from the application developer by the abstraction mechanisms it provides. It is important to study test methods that will accommodate this type of system.

The primary objective of this study was to determine if the three testing techniques listed above were effective for a fourth generation language. The effectiveness was determined by measuring how well the software system conformed to its system requirements specification (SRS) when confronted by a battery of tests. For our study, we determined the effectiveness of each testing technique by whether or not it identified discrepancies between the application's behavior and the SRS.

2.0 Background

Syntax testing is performed to ensure that input into an ORACLE system is valid. As with every language, ORACLE has a defined syntax for each field (entity of a database table) that exists in the system. Fields may also have a programmer specified syntax associated with them, such as a valid range on a field defined to contain numbers. Syntax testing is most beneficial to the programmer when the correct syntax of the language has been formally defined. Therefore, both the ORACLE syntax and the programmer specified syntax will be determined. This allows the programmer the ability to develop creative test cases that will identify the system's ability to decipher between incorrect input and valid input.

Data-flow testing is performed to discover how well it uncovers data-flow anomalies that may occur during the execution of an ORACLE application. The All-Predicate-Uses/Some-Computational-Uses strategy was applied and will be explained later in this paper. The main goal for using this technique is to trace each data object through the system as it is executed to determine if the objects are created, killed, and/or used as intended.

Statistical testing is a method used to validate software requirements through software execution in representative usage environments. By using probability distributions to define the use of software with their likely inputs and by randomly selecting which functions are tested with which inputs, the test environment is statistically representative of the real operating environment. It allows the tester to make reliability predictions, based on statistical significance, by taking a random sample from the total population of all functions and their possible inputs, and then to determine the operational effectiveness of the entire system.

To investigate the effectiveness of syntax testing, data-flow testing, and statistical testing on software systems developed in ORACLE, two different systems were chosen. Each system was developed by a different team. The first system is referred to as System I, and the second system is referred to as System II. Both systems were developed to support a client/server environment where the client resided on an IBM 386 or 486 PC, and the server resided on a DEC 5500 workstation.

System I was a waste tracking system developed for the Department of Energy to monitor and control hazardous waste packaging and waste cargo transport at the National Laboratories. The system was designed to monitor waste packaging, physical characteristics of the waste contained within the package, and waste manifest activities.

The waste tracking system consisted of 32 tables where each table contained anywhere from three to 36 entities. The system had a total of 41 screens. It was designed and developed by a team of five engineers working full time for eleven months to complete the project. All five engineers participated in the requirements, design, development, and testing phases of the project. Each engineer tested their own work until the system became integrated. The system was then tested by two engineers for the duration of one week. All errors found during this phase were corrected.

System II was developed for one of the Department of Energy's National Laboratories to monitor strategy and schedule for developing, prioritizing, implementing, and monitoring appropriate response actions in accordance with DOE policy and federal law. The system aids in facilitating cooperation and exchange of information, and minimizes duplication of analysis and documentation.

System II consisted of seven tables where each table contained anywhere from five to 19 entities. The system had a total of 20 screens. The system was designed and developed by a team of three engineers, only one of them working full time, over a period of five months. Only one engineer was responsible for the requirements, design, and development phases of the project. Two engineers participated in the testing phase, where one of the engineer's sole responsibility was to detect errors in the system. The third engineer was responsible for verifying the correctness of activities at each phase of the project. All errors found during the test phase were corrected.

3.0 Methods and Procedures

ORACLE Systems I and II were tested using syntax, data-flow, and statistical test methods. The testing of these systems did not begin until after the systems had been ad hoc tested by the developers for a one week duration and pronounced ready for release. The number and types of defects found during the initial test period were not recorded. Following the ad hoc testing, statistical testing was performed first, data-flow testing second, and finally, syntax testing. Each error found in Systems I and II during statistical testing was corrected at the time it was found. System II was copied and baselined before statistical testing began, System I was not. This resulted in Systems I', II', and II being available for data flow and syntax testing. The results of the statistical testing were analyzed using Bayesian prior and posterior distributions. Syntax

testing and data-flow testing techniques were performed on these systems without corrections between tests. All screens that were randomly selected for the statistical test method (across all trials) were tested using the syntax and data-flow test methods. This decision was largely driven by the fact that the statistical test method was performed first, and by testing the same screens, a baseline could be established for a final comparison of all three test methods. All three test methods proved to be effective for testing ORACLE applications. Results and findings of the three software testing techniques are described below.

3.1 Syntax Testing

Syntax testing is a data-driven technique that systematically tests the input data recognition properties of a program. Syntax testing was developed as a tool to test input data to ensure that only proper forms are accepted and improper forms are rejected. Syntax testing was performed to ensure that input into an ORACLE application was valid.

ORACLE allows the programmer to use predefined syntax for each field (entity of a database table) that exists in the system, but fields may also have a programmer specified syntax associated with them, such as a valid range on a field defined to contain numbers. Syntax testing is most beneficial to the programmer when the correct syntax of the application's input data has been systematically defined. Therefore, both the ORACLE syntax and the programmer specified syntax must be determined. This helps the programmer develop creative test cases that will identify the system's ability to differentiate between incorrect input and valid input.

All of the fields that existed in the screens tested for the three systems (I', II', and II) were identified. If the field had a programmer specified syntax, such as a valid number range or a specified character format, the syntax was identified. Otherwise, the ORACLE default syntax was assumed. The formal Backus-Naur Form (BNF) syntax and the syntax trees for each field were created. Both valid and invalid test cases were derived. Test cases began with the first field on each screen. For invalid test cases, errors were introduced one at a time for each level of the syntax tree until all errors that could occur were identified.

Only those screens that allowed input data (add and update screens) and were selected for statistical testing, for both Systems I' and II', were selected for syntax testing. Screens not selected for statistical testing were not tested. The same screens selected for System II for statistical testing were selected for System II'. This resulted in 21 screens selected for testing from a total of 41 screens for System I and 11 screens selected for testing from a total of 20 screens for Systems II and II'.

Numerous errors in Systems I', II', and II were found by using syntax testing. The types of errors found were commonly shared by all programmers, i.e., it was believed that no one programmer was responsible for one type of error or even the majority of one type of error (though this was not rigorously investigated).

As a result of syntax testing, 44 errors were found in System I' and 32 errors were found in both Systems II' and II. The most common types of errors detected in System I' were:

- null values were accepted where the user was expected to enter a value,
- numbers not in the valid range of numbers indicated by the system were accepted,
- invalid two-character state identifiers were accepted, and
- invalid telephone numbers/telephone number formats were accepted.

The most common errors found in Systems II' and II were:

- the systems accepted invalid date formats and values,
- invalid numbers and/or characters were accepted where only a particular range was acceptable.

3.2 Data-Flow Testing Background

As defined by Beizer, "Data-flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects." [BEIZ90] Data flow testing recognizes that the data properties of a program are affected by the program's execution dynamics in ways that are not always observed under simple control flow testing. Since database applications are data-intensive, it is important to perform testing that is likely to discover data manipulation errors. Two different strategies were considered for this study:

- All-du Paths
- All-predicate-Uses/Some-computational-Uses and All-computational-Uses/Some-predicate-Uses.

The All-du data-flow testing strategy provides the most rigorous coverage of the four data-flow strategies listed. It requires that every definition of every variable to every use of that variable be tested. Flowgraphs must be developed that identify each variable in the program and every path that variable follows. From the flowgraph, test cases can be developed to ensure that the branch or statement coverage outlined in the graph is adequately tested. For example, a variable, say alpha, has been defined at the entry node of a control flowgraph and is used at a subsequent node without redefinition. Any test that starts at the entry satisfies the all-du-paths criterion. However, the test situation becomes more complicated if alpha has been redefined in may places. In this situation, paths that include all subpaths where alpha is used, after redefinition, must be exercised. If alpha has a predicate use at some particular node, and the subsequent path to the end is forced from two directions, the all-du-paths strategy for this variable requires that all loop-free entry/exit paths and at least one path that includes the loop be exercised. If subsequent paths exist that do not define alpha, yet they provide alternate du paths to the use of alpha at some later node, these paths must be included in the test set because they each lead to the use of alpha.

The all-predicate-uses/some-computational-uses (All-p-uses/some-c-uses) strategy requires that every variable and every definition of that variable be exercised along at least one path, where the variable is not redefined, to every predicate use of that variable. In other words, for every variable, select a path where the test begins at the variable definition and continues through every predicate use and ends at the next variable definition. If there are definitions of the variable that are not covered by the above strategy, then computational-use test cases must be added to ensure that these definitions are covered.

Data-flow testing was performed to discover how well this approach uncovers data-flow anomalies that may occur during the execution of an ORACLE application. Because of time constraints, the all-p-uses/some-c-uses strategy was chosen for this study. Data flowgraphs were developed for each ORACLE screen and their data objects defined. Each screen was tested as a separate system module. The main goal was to trace each data object through the system (as the system executed) to determine if it had been created, killed, and/or used as intended.

Data flowgraphs for Systems I', II', and II were developed and each system was tested for data-flow anomalies using the data-flow test method. The data objects defined for each system were the

ORACLE key triggers, global variables that were defined in a screen, any local variables that may have been defined in an ORACLE procedure, and reading from and writing to a record. Each graph consisted of nodes and directed links, as illustrated in Figure 3-1.

ORACLE triggers are sets of commonly used subroutines that are associated with function keys on the keyboard and are called from ORACLE SQL*Forms. They are a set of processing commands written by the system programmer. Different types of triggers are associated with different event points of an executable file. When an event occurs with which a trigger is associated, the trigger "fires", executing the commands that it contains. Triggers can be internal code (formally known as packaged procedures) made available to the programmer by a simple call statement such as "NEXT-FIELD" or subroutines that are actually coded by the programmer in PL/SQL language triggers can be defined at the screen level, the block level, or the field level. (Each block represents a different database table, and each field represents an entity within that table.)

A global variable is a data store that exists across forms, i.e., if one screen defines a global variable, and at some time during its execution calls another screen, the global variable will still exist and contain the same data it contained in the calling form. Once a global variable is created, SQL*Forms maintains that variable until either the application is exited, or the variable is redefined.

Local variables are PL/SQL variables that are active only within the block or form-level procedure in which it was declared. When the procedure is done executing, the local variable no longer exists.

Reading from a record occurs when a query has been performed on a database table. It is considered a success if the query retrieves the specified record. Writing to a record occurs when a record has been committed to a database table. If a record and/or changes to a record are successfully committed, the write was a success.

Because System I was so large (it was estimated that 12,000 test cases would have to be produced if the All-du Paths strategy had been chosen), the All-p-Uses/Some-c-Uses strategy was chosen. Each data object was tested from the point at which it was defined and along at least one definition-free path from the definition to every predicate-use or computational-use. However, most test cases were created so that more than one definition-free path was covered per data object.

Test cases were created to determine the state of a data object at any given time during program execution. Specifically, a data object can be in one of four different states: 1) K (undefined, previously killed, does not exist, 2) D (defined but not yet used for anything, 3) U (has been used for computation or in a predicate, or 4) anomalous. This was accomplished by selected test path segments for each data objects, and noting its state at each node along the path.

ORACLE systems I', II', and II were tested using data-flow methods. Each screen selected for statistical testing, for both Systems I and II', were also tested using data-flow techniques. Screens not selected for statistical testing were not tested using data flow techniques. The same screens selected for System II were also selected for System II'. This resulted in 29 screens tested from a total of 41 screens for System I and 16 screens tested from a total of 20 screens for Systems II and II'.

The number of errors found in System I' as a result of data-flow testing were 11, the number of errors found in System II' were 16, and the number of errors found in System II were 18. The most common types of errors detected were:

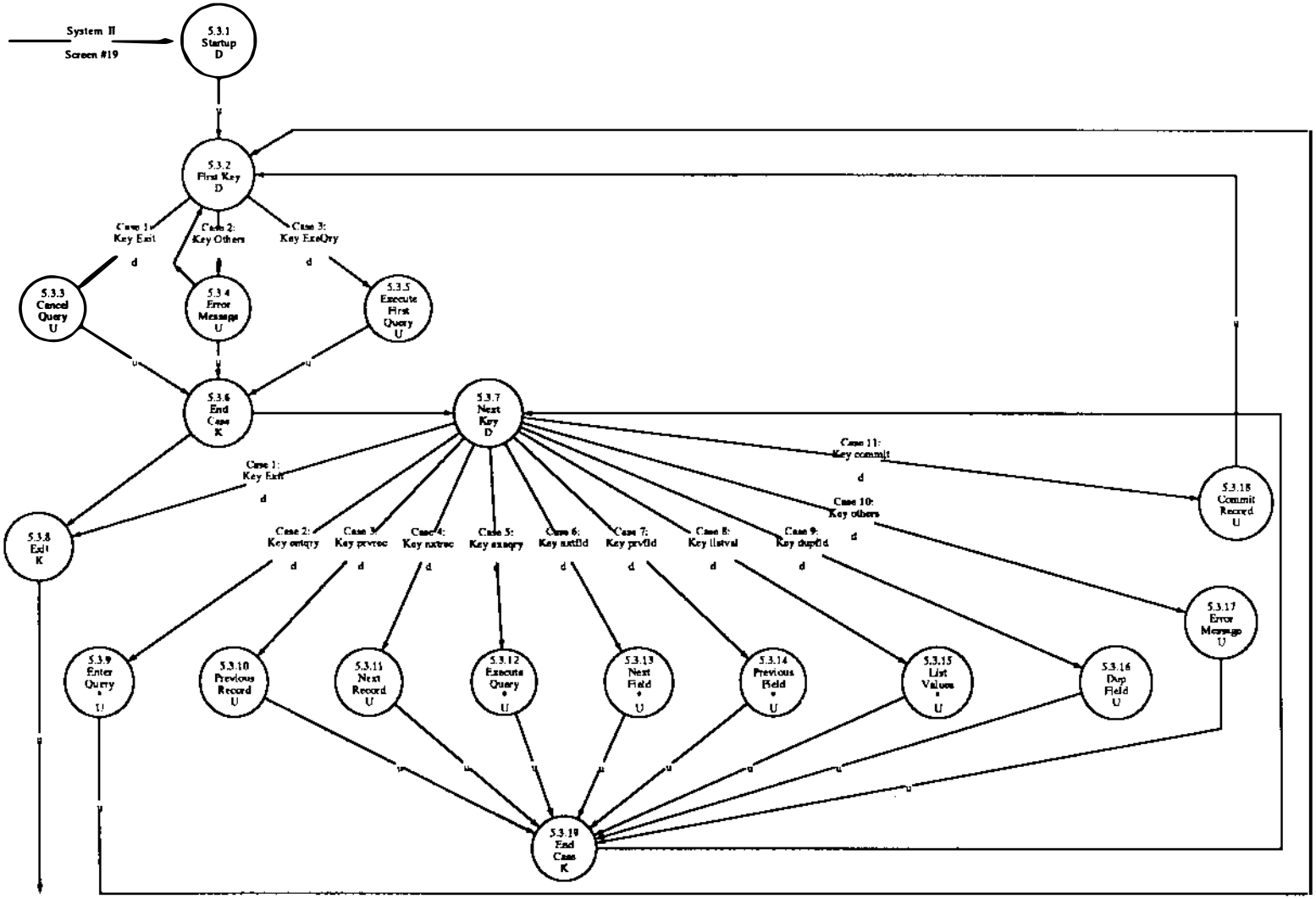


Figure 3-1. Data-Flow graph for screen 19 of System II'.

- triggers that had been defined earlier in the program were still active at points in the program when they should have been either redefined or killed,

- triggers previously defined that should still have been defined at a given point in the program had been killed, and

- triggers that needed to be defined were not.

These errors existed as a result of an incomplete software requirements specification, i.e., initially some software requirements were overlooked and were not detected until the system was systematically tested.

3.3 Statistical Testing Background

Statistical testing is a method of verifying the quality of software by scientific measurement. It differs from other methods in that samples of "real" data are used for testing as opposed to the specially constructed test data usually required by other test methods. Statistical testing has three basic steps. First, the intended use of the software is characterized, then random sample test cases are taken from this environment. Second, each input test case produces an output that must be determined to be either right or wrong. And third, the quality of the software is quantified in terms of its measured reliability over a probability distribution of the usage scenarios. The more frequently wrong results are obtained, the lower the application's reliability.

Our study utilized the basics of statistical software testing techniques reported by Michael Dyer [DYER92]. For the past several years these techniques have been used effectively for third generation languages. The goal of Dyer's method is to validate a system's conformance to its software requirements through software execution in a representative usage environment. By using probability distributions to define the use of software with their likely inputs and by randomly selecting which functions are tested with which inputs, the test environment is statistically representative of the real operating environment [DYER92]. This approach allows the tester to make reliability predictions, based on statistical significance, by taking a random sample from the total population of all functions and their possible inputs.

To make the determination of how well an ORACLE application conformed to its system specification, the probability of the number of bad functions still contained within the application (before its release) was estimated. A Bayesian prior distribution expressing prior information about the possible number of bad functions was combined with direct sample evidence from the software testing to produce a posterior distribution on the number of bad functions. A Bayesian confidence interval was then determined.

The goal of this method is to estimate the value of B, the number of bad functions contained within the software system. The Bayesian approach quantifies the experimenter's belief about the estimated B by a probability distribution.

The posterior probability is the probability that an inference about B is the prevailing one, given the sample outcome of the true unknown B. The outcome state probabilities P(B) are called prior probabilities because the sample information subsequently provides further information about the prevailing B. The mean of the posterior distribution can be used as a point estimate of the total number of bad functions contained within the software system. A 90% Bayesian confidence interval consists of the 5th and 95th percentiles of the distribution. Therefore with 90% certainty, as a result of the posterior distribution, B lies somewhere in the interval estimate. To obtain an estimate of the number of bad functions still remaining in the untested portion of the system after

the system test has been completed, subtract b, the actual number of bad functions found, from the point estimate for B.

Systems I and II were tested by randomly selecting the components to be tested, statistically testing the components, correcting any bad functions found, randomly selecting components again, statistically testing those components, correcting the bad functions, and so on. The methods are described below.

For both Systems I and II, the required initial analysis began with identifying the software functions to be tested and the associated inputs that triggered execution of those functions. Test cases were prepared according to the system's software requirements specification.

The second phase of the initial analysis examined the software input values. System I was a newly developed system that had been mandated by the Department of Energy Headquarters to be used nationwide. Details of the input values for System I were gathered from one of the end users of the system, from the Resource Conservation and Recovery Act Regulations (RCRA) manual, and from the system's SRS. Input values for System II were derived from the system's design document, supporting literature and federal regulation manuals, and from the system's SRS. In order for the statistical test to be successful, it was necessary to identify the total set of software inputs, along with any design or operating constraints such as boundary conditions and legal input combinations [DYER92]. The total set of software inputs were identified and documented in test cases developed for each system.

A series of trials was conducted where system screens were selected randomly and then tested. Errors uncovered as a result of the tests were corrected, and corrections were verified by running the same test cases before another trial was generated. Upon completion of testing, the probability of the observed data, assuming the number of bad functions B, was calculated. A Bayesian prior distribution on B combined with the direct sample evidence from the software testing was used to produce a posterior distribution on B. A Bayes confidence interval was then determined.

The prior distribution should reflect what is initially believed about B. For this study, two diffuse prior distributions were used – the uniform and the maximum entropy prior distributions. Diffuse priors were chosen because the actual data found from testing can prevail over and correct a diffuse prior distribution.

The first prior distribution selected for B was the uniform, where the smallest outcome is 0, and the number of distinct outcomes in N. This distribution allowed B to take on integer values within the predefined interval for B with equal probabilities. It is also pessimistic (hopefully) because it makes the claim that on average, half of the functions contained in the software specification are bad. This prior combined with the testing data yields a posterior distribution that has a longer right tail than would have been obtained from a more realistic prior distribution.

The second prior distribution used was the maximum entropy prior. This prior was selected because it was capable of providing a more realistic estimate of B. It was assumed that one tenth of all of the functions might be bad. Though there was no direct sample evidence to lead to this determination, it was believed that other ORACLE applications tested in the past indicated that one tenth of the functions were bad. Therefore, a prior distribution was sought with mean equal to $N/10$. The maximum entropy distribution was chosen, having the specified mean above, and minimizing

$$\Sigma\ (p_i \bullet \log p_i\ ),$$

where p_i is the probability that B equals i.

Since it is not always feasible for an entire system to be tested, statistical testing incorporates the use of random selection of the number of screens that will be tested. For random selection in this study, the screens for each system were identified and a sequential number, beginning with one, was manually assigned to each screen. Because at the onset of this study it was not known exactly what prior distribution would be used, the uniform distribution was simply selected to determine an appropriate trial size. A trial size that would yield at least an 80% confidence interval was chosen. As a result, it was decided that a trial size of thirteen (i.e., the number of screens chosen for testing) for System I and a trial size of eight for System II would be adequate. A random number generator was executed for each system to identify which screens to execute for each trial. During the first trial for System I, it was found that one screen had not yet been completed by the programmer, and was therefore dropped from the trial although it was later randomly selected and tested in another trial. This resulted in only twelve screens being tested in the first trial.

The number of trials conducted on each system was determined by whether a trial resulted in any bad functions. Each trial consisted of testing the number of screens defined above. If bad functions were found, more trials were conducted. Otherwise, testing was considered to be complete. For System I, a total of four trials were conducted; each trial tested 12 or 13 screens, and in all 29 screens were partially or fully tested. System II resulted in three trials, with eight screens per trial, and a total of 16 screens partially or fully tested.

System I had a total of 592 functions and subfunctions in 41 screens. Of those screens selected for testing, 327 functions and 62 subfunctions were tested from a total of 372 functions and 97 subfunctions. Because of the stopping criteria, not all functions and subfunctions randomly selected were tested. This resulted in 84% of the functions and subfunctions being tested from the total number of functions and subfunctions from the screens selected in all four trials, and 66% of the total number of functions and subfunctions in the entire system.

System II had a total of 258 functions. Of those screens selected for testing, 160 functions and 16 subfunctions were tested from a total of 211 functions and 20 subfunctions. Because of the stopping criteria, not all functions and subfunctions randomly selected were tested. This resulted in 76% of the functions and subfunctions being tested from the screens selected in all four trials, and a 69% of the total number of functions and subfunctions in the entire system.

There were a total of 8 bad functions found in System I, therefore, the proportion of bad functions was $p = 0.01705$. There were a total of 9 bad functions found in System II resulting in $p = 0.05110$.

Both the uniform and the maximum entropy prior distributions resulted in similar posterior distributions, for both System I and II, as can be seen in Figures 3-2 and 3-3. For System I and II, both the uniform and the maximum had skewed bell-shaped distributions, with the maximum entropy prior peaking just slightly to the left of the uniform prior. Both distributions for System I peaked at approximately $B = 12$. Both distributions for System II peaked at approximately $B = 13$. Both the maximum entropy and the uniform rise from 0.0 to 1.0, but the distribution based on the maximum entropy distribution rises a little faster.

For System I, the posterior mean was equal to 12.67 based on the uniform prior and equal to 12.56 based on the maximum entropy prior. The 90% confidence interval for B, produced by both distributions, was $9 \leq B \leq 17$. Because B is discrete, not continuous, the posterior probability that B lies in the interval is not exactly 90%, and is actually 0.929 based on the uniform prior and 0.932 based on the maximum entropy prior. This means that with a 93% probability, based on the uniform prior, and on the maximum entropy prior, the estimated total number of bad functions in System I lies somewhere between 9 and 17 out of a total of 592 functions.

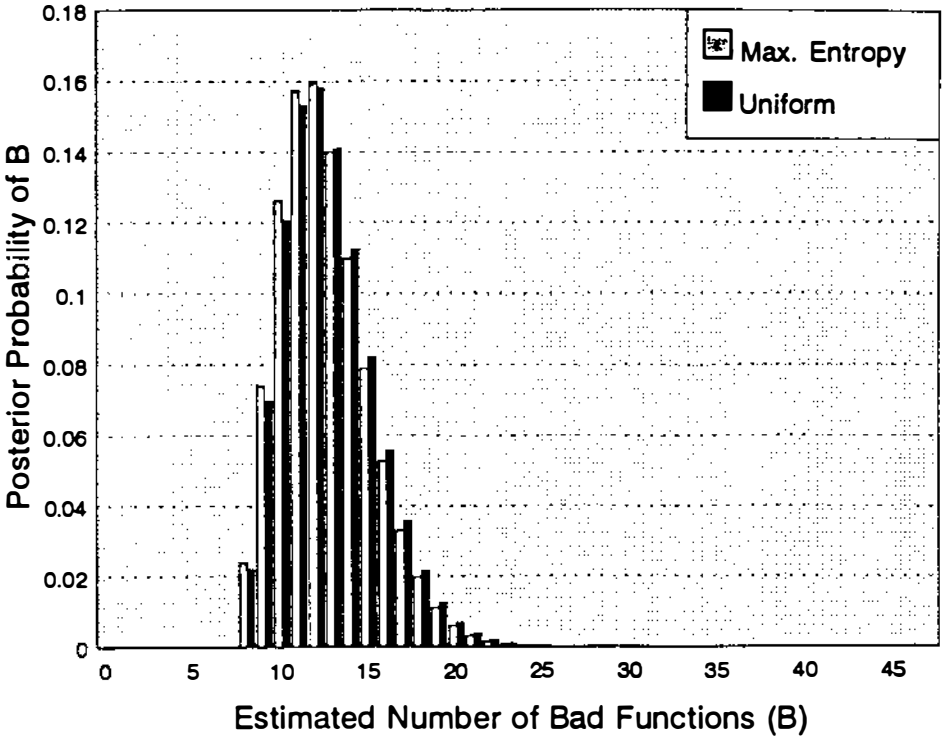


Figure 3.2. Posterior Distributions, System I

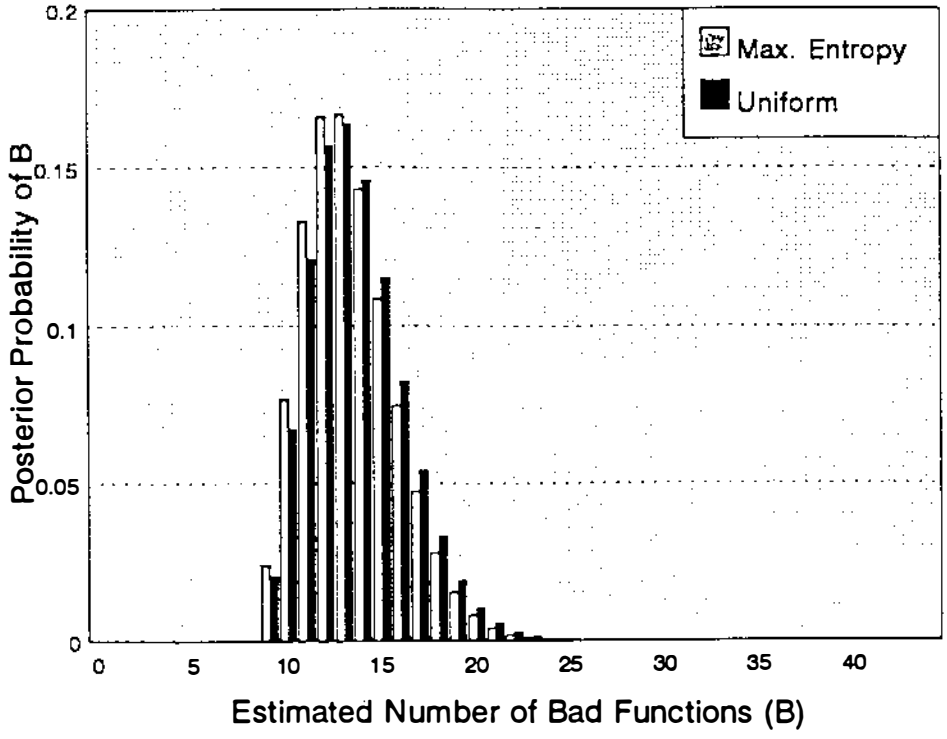


Figure 3.3. Posterior Distributions, System II

The posterior mean for System II, based on the uniform prior, was 13.61 and 13.37 based on the maximum entropy prior. The 90% confidence interval for B, again produced by both distributions, was $10 \leq B \leq 18$. The posterior probability that B is in the interval is 0.939 based on the uniform prior, and 0.944 based on the maximum entropy prior. This means that with a 93% probability, based on the uniform prior, and with a 94% probability based on the maximum entropy prior, the estimated total number of bad functions in System II lies somewhere between 10 and 18 out of a total of 258 functions.

Because both the uniform and maximum entropy prior distributions resulted in similar posterior distributions, it can be determined that the distribution is based primarily on the data, not on the prior belief about B. A prior distribution that tended to be closer to zero could have lead to a shorter 90% interval, but the interval obtained for both Systems I and II provides a credible estimate of the total number of bad functions contained in the software system.

To calculate the estimate of the number of bad functions still remaining in the untested portion of the system, after the system test, simply subtract b (the actual number of bad functions found) from the point estimate of B. In System I, 8 bad functions were found. Therefore the posterior mean of $B - b$ is $12.67 - 8 = 4.67$ based on the uniform prior or $12.56 - 8 = 4.56$ based on the maximum entropy prior. This value is an estimate of the number of bad functions still remaining in the untested portion of the system. To obtain the estimated *fraction* of bad functions still remaining, divide the posterior mean by the total number of functions, *4.67/592* or *4.56/592*, which is approximately *0.8%*. A 90% interval estimate for number of remaining error producing functions becomes $1 \leq B - b \leq 10$.

There were 9 bad functions found in System II. The posterior mean of B is therefore $13.61 - 9 = 4.61$ based on the uniform prior or $13.37 - 9 = 4.37$ based on the maximum entropy prior. This is approximately 1.8% or 1.7% depending on the prior used. A 90% interval for the actual estimate of the number of remaining error producing functions is $1 \leq B - b \leq 9$.

4.0 Comparison of Defects Found by Each Approach

The amount of overlap in the types of defects found by each of the three test methods, syntax testing, data-flow testing, and statistical testing for both System I and System II was minimal. Common defects found in System I by syntax and statistical testing were either errors where the system accepted values that were not within the specified valid range or where the system accepted a null value where a null value was not acceptable. The only common type of defect found by syntax and data-flow testing was the nonacceptance of a value that was within the specified valid range. No common defects were detected by data-flow and statistical testing.

In System II, there were no common defects found by syntax and statistical testing, and there were no common defects found by syntax and data-flow testing. There were three types of defects found by data-flow testing and statistical testing and are as follows: 1) the inability to maneuver to the next record, 2) the inability to maneuver to the next correct field, and 3) the ability to enter a field that was supposed to be non-enterable.

For System II', only syntax testing and data-flow testing are compared. Statistical testing was not performed on System II' – statistical testing produced System II' from System II. As was true for System II, there were no common defects found by syntax testing and data-flow testing. While testing System II, data-flow testing found two more defects that were not found in System II'. This was due to the corrections that were made to System II during the statistical tests. Otherwise,

syntax testing and data-flow testing found the same types of defects and the same number of defects in both System II and System II'.

Syntax testing, data-flow testing, and statistical testing uncovered entirely different errors in both System I, System II, and System II'. Syntax testing uncovered errors at the field level of each screen – acceptance of incorrect syntax or the nonacceptance of correct syntax. Data-flow testing uncovered errors where triggers (function keys) were incorrectly defined, i.e., they did not perform the function they were supposed to perform at a given point in time during execution. It was surprising, however, that the errors found by statistical testing did not overlap more with both syntax testing and data-flow testing. There are a couple of reasons that may explain this. First, the stopping rule in statistical testing may have prevented portions of the screen from being tested where the errors resided, i.e., testing may have ceased before the errors were reached. Second, the errors may not have been detected because either the corresponding requirements in the system's SRS were not explicitly defined or the requirements were overlooked altogether.

The most serious defects discovered in System I were the acceptance of data values that were not in the acceptable range, and the acceptance of null values when null values were not acceptable. These data values are required for formal reporting purposes and an audit trail must exist for these data items. It is very important that they are not invalid or missing. These defects were discovered by syntax testing.

The most serious defects discovered in System II and System II' was the ability to update data that resided in a protected field. These data values are required for formal reporting and they must be valid entries and the entries must be correct (i.e., not contain a valid but incorrect value). Again, these defects were discovered by syntax testing.

5.0 Conclusions and Recommendations

The syntax test method used in this research proved to be effective in determining whether the system would accept invalid data and/or invalid data formats. Though ORACLE provides the selection of field types, such as character or number, the programmer must still specify, in most cases, the character and number formats and the valid ranges associated with each. This test method was also extremely effective in identifying the most common errors shared by programmers. Because these types of errors were so common, measures to improve them are easily realized.

The results obtained from data-flow testing were the most unexpected compared to the other two methods. At the onset, it was thought that this method of white box testing could not possibly be effective on a black-box system. However, the results proved to be quite the contrary. Data-flow testing was extremely useful in determining whether a function key (procedure) was defined at the appropriate time during the system's execution. This method forced the user to test all the function keys, at separate time intervals during execution, that had been previously defined. It immediately becomes clear that functions keys that should have been killed or redefined after their intended use were not. As a result, all types of errors occurred.

Though only 60 to 70% of the entire system is tested using statistical methods, this method provides a good approximation of the system's operational effectiveness. For our study, however, this method does not provide a good measure of the seriousness of the errors remaining in the system and is not recommended when the system is meant to control life threatening situations. As systems become large, this type of method can be extremely beneficial. In many cases, the developer may not have the time or budget to test an entire system.

All three test methods, syntax testing, data-flow testing, and statistical testing proved effective on testing applications developed in the fourth-generation language ORACLE. Syntax testing found the largest amount of errors and proved to be highly effective for finding errors that allowed users to input invalid data. Data-flow testing found the second largest amount of errors and was very effective for determining whether the proper procedures were available when they should have been, and not available when they should not have been. Statistical testing found the least amount of errors, but uncovered errors where the system did not provide the functionality that was detailed in the software requirements specification. The testing activities performed on these systems improved the product quality both in terms of improved usability and improved correctness of the system. Each method took approximately one week to conduct on each system. Therefore, the entire testing effort (excluding the one week ad-hoc testing performed by the programmers) took approximately three weeks. From corrections made as a result of these testing activities, the systems will be more reliable and easier to use.

ACKNOWLEDGMENTS

The authors wish to thank Scott Matthews for his continued support, editorial comments, and friendship for the duration of this project. They would also like to thank Drs. Corwin Atwood, David Barber, and Earl Marwil for their suggestions that led to the success of this project.

The authors would also like to thank Christine Lee and David Simmons for providing the funding to support this work, and Brent Stacey for helping to secure the funding and for approving the time necessary to complete this study. Without these people the study could not have taken place.

REFERENCES

[BEIZ90] 1. Beizer, Boris, Software Testing Techniques, Second Edition, Van Nostrand Reinhold Company Inc., 1990.

[DYER92] 2. Dyer, Michael, The Cleanroom Approach to Quality Software Development, John Wiley & Sons, Inc., 1992.

TESTING FOR QUALITY
Survey of Research in the State of the Art

Dick Hamlet†
Portland State University
Center for Software Quality Research
Box 751
Portland, OR 97207
Phone: (503) 725-3216
Internet: hamlet@cs.pdx.edu

Keywords:

testing to find faults, testing for reliability, testing for correctness, unit vs. system testing

Biography:

Dick Hamlet is Professor of Computer Science at Portland State University. His current research interest is software engineering, particularly the theory of testing and software testing tools. He has been actively involved in theoretical program-testing research and in building testing tools for more than 20 years. He has implemented major software systems for a simulation language, a systems programming language, the first mutation testing system, a transportable image-processing system, and a system for constructing prototype testing tools. He was a member of the software engineering research group at the University of Maryland for 12 years, and a visiting lecturer at University of Melbourne in 1982. He is the author of two textbooks and about 50 refereed conference and journal publications. His paper assessing software reliability won the best paper award from *IEEE Software* in 1993. He has worked as an operating-systems programmer and systems-programming manager for a commercial service bureau and for a university data-processing center. Hamlet received his BS in electrical engineering from the University of Wisconsin (1959), an MS in engineering physics from Cornell University (1964), and PhD in computer science from the University of Washington (1971).

† Work partially supported by NSF grant CCR 9110111

TESTING FOR QUALITY
Survey of Research in the State of the Art

Dick Hamlet

Extended Abstract

Software testing is the most technically sophisticated and precise part of development (if programming languages and their compilers are excluded), yet its techniques are used uncritically. Most testing effort is just that—diligent work by motivated engineers—without solid scientific foundation. In the current rush to codify procedures of the software development process, it is easy to forget that a good *process* is not the goal, but rather quality software. A labor-intensive testing phase, no matter how well defined and controlled, is a wasteful mistake if it cannot be demonstrated to actually contribute to software quality. Structural coverage testing is a particular example. Coverage is a systematic idea, supported by measurement tools. But there is no data relating coverage and the effort expended to attain it, to how many defects escape detection. Of course, each bug found and removed is a gain. But testing that leaves bugs that a customer finds immediately is a mistake, no matter how many bugs were found or how hard the testers worked to find them.

Research into testing techniques should establish their scientific basis. Sadly, in the all-too-typical case study, success of coverage testing is *defined* in terms of coverage. (Or, one kind of coverage is shown to lead to another kind of coverage, which is defined as success.) It is rare for a case study to relate coverage to failures found in testing, yet even this falls short of the mark. What counts is not how many defects were found, but how many might remain unfound. It is a characteristic of wishful thinking to replace a real goal (quality software) with a measure of effort (test coverage attained).

Supporters of process analysis argue that the link between process and product will be established by feedback from failures. A particular software failure could be traced through the (well defined!) testing process—why wasn't it caught?—and the process modified so that things will be better in the future. To continue with the coverage example, why does structural coverage sometimes fail to find failures that later appear in the field? (And just how often does this happen?) There is a dearth of published data, because experiments on real software are notoriously hard to carry out. Process corrective feedback will not work if the "correction" is a directionless "try something else." There are too many possibilities, and it is far too expensive to conduct significant experiments on even a few of them.

In the last five years, research has begun to address the relationship between testing techniques and the goals that they might support, such as:

A) to find failures;

B) to ensure high reliability;

C) to predict release date.

The methods appropriate to one goal may fail to address the others. For example, finding failures does not necessarily lead to reliability or advance toward release. Or, methods that can certify

reliability, when the reliability is *not* acceptable, may fail to be helpful in finding and fixing defects to improve it.

This paper reports on research (from a dozen sources, recent conferences and journal publications) that ties practical methods to each goal. Briefly:

A) *Find failures.* The commonly accepted goal for testing is to uncover as many problems as possible, so that they may be fixed before release. Perhaps because the goal is so natural, recent research has addressed it with some success. The results have clear practical application, particularly to unit testing. The central concepts of this research are first, the use of probabilistic models of testing; and second, study of the relationship between the subdomains induced in the test space by a testing method. The way in which these subdomains overlap, and how they are related to potential faults, allows a comparison of methods. (Work of Duran, Frankl, Hamlet, Nfafos, Taylor, Thévenod-Fosse, Weyuker.)

B) *Ensure reliability.* In safety-critical applications or where a software failure can have important economic consequences (the so-called "ultra-reliable" region), the goal is really perfection. The developer seeks guarantees that any failure is unlikely. Dijkstra's famous aphorism that testing can reveal only the presence of failure, not its absence, is being questioned by research into probabilistic testing. The "10^9 problem" is that straightforward sampling techniques are impractical because the time needed to test is the same order of magnitude as the desired result. Thus to achieve the fabled prediction of 10^9 runs without failure requires roughly 10^9 tests, which would take hundreds of years to perform. The new ideas of *testability* and of *self-checking algorithms* show promise of solving the 10^9 problem. Methods aimed at perfection may also be applied to less critical software projects, and realize less striking results, but at low cost. (Work of Blum, Butler, Finelli, Hamlet, Howden, Littlewood, Miller, Podgursky, Voas.)

C) *Predict release date.* Reliability is a measure of failure behavior for a single program. In contrast, reliability-growth models describe the debugging process in which a program fails under test, is changed, then again the new program fails (we hope for a different reason!), etc. The models attempt to predict when debugging will have lowered the failure rate enough so that the program can be safely released. However, empirical data are so noisy that they cannot demonstrate that the models work. The focus of reliability-growth research is on incorporating program characteristics in the model parameters. If the model parameters can be calculated from the program *a priori* it not only makes application of the model easy, but improves its plausibility. (Work of Everett, Malaiya, Munson, Musa.)

The most exciting possibility opened up by recent testing research is a connection between failure-finding methods and reliability. Reliability and reliability-growth models apply only at the system-test level, while failure-finding methods are most practical at the unit level. In particular, reliability and unit testing are fundamentally incompatible, because operational profiles cannot be obtained (nor should they be, if reusability is an issue) at the unit level. Yet practical testers believe that failure-finding methods are indispensable for attaining system reliability. It is the business of the developing theory to support this belief.

Integrating the Developers' and the Management's Perspective of an Incremental Development Life Cycle

Andreas Zamperoni

Leiden University
Dept. of Computer Science
P.O. Box 9512, 2300 RA Leiden
The Netherlands
Tel.: ++31/71/27 7103
Fax.: ++31/71/27 6985
email: zamper@wi.leidenuniv.nl

Bart Gerritsen

TNO Institute of Applied Geoscience
Information Systems Oil&Gas
P.O. Box 6012, 2600 JA Delft
The Netherlands
Tel.: ++31/15/69 7196
Fax.: ++31/15/56 4800
email: gerritsen@igg.tno.nl

Abstract

Purely sequential life cycles can't cope with the developer's need for relatively unrestricted and "unordered" creativity when dealing with the development of innovative software systems. Nevertheless, project management usually favors sequential project phasing, because sequential processes are easier to plan, control and monitor.

In this paper, we present a life cycle plan that integrates an incremental and iterative development style, relying on evolutionary prototyping, with the management's perspective of clear and unambiguous project phasing. As consequence of applying this "hybrid" life cycle plan, the quality of innovative, complex software systems can be increased, while risks can be detected better and earlier.

We support validity of our approach by also reporting experiences we made when applying the life cycle plan to two large, multilateral software projects, and compare them to two systems developed following the traditional approach.

Keywords: incremental development, life cycle models, evolutionary prototyping, project management, risk minimization

Biographical: *Andreas Zamperoni* received a master degree in computer science from the Technical University of Braunschweig (Germany) in 1992. Since 1992, he works on a PhD on Software Engineering Methodologies at the Leiden University (The Netherlands), in the research group for Software Engineering and Information Systems. At the same time, he works at TNO Institute of Applied Geoscience (Delft, The Netherlands), where he investigates, evaluates, and enhances software development of geoscientific software systems. His research topics include object-oriented software engineering methodologies, software process modeling, and system architectures. He has also published in the field of quality enhancement for conceptual specifications. *Bart Gerritsen* received a master degree in mechanical engineering specializing in CAD from the Delft Technical University in 1986. He has been a senior system developer and project leader at Cap Gemini. Since 1991, he is project leader and software quality assurance officer at TNO Institute of Applied Geoscience (Delft, The Netherlands). He has been a project leader of the TNO participation at a 20+ man year international project funded by the EC. Currently, he is leading a project for the innovation of the entire information infrastructure of an oil company in South America. He is also working on a PhD on Subsurface Modeling Methodologies.

1 Introduction

When dealing with the development of innovative, complex software systems, such as e.g. neural networks for seismic simulation, traditional software engineering approaches are no longer suitable. *Prototyping* has emerged as one of the prime methods to facilitate, or even enable, the development of such innovative, experimental software products. Applying prototyping in software development consequently implies an adjustment of the software process model, too. Prototyping is strongly related to *incremental* or *evolutionary life cycle models* [Boe88], as opposed to the traditional sequential life cycle models [Boe76]. Nevertheless, project management often favors traditional sequential life cycles as the *waterfall model* and its variations because sequential processes are much easier to plan, control and monitor. Our paper describes how an incremental life cycle incorporating *evolutionary prototyping* as main concept can be integrated with the management's perspective of *sequential project phasing*, and reports the experiences we made applying such an integrated process plan during the development of two large geophysical software systems, compared to two similar systems developed earlier in the traditional manner.

At TNO[1], we succeeded to define and to establish a "hybrid" *software process plan* which bridges the gap between the developers' needs for relatively unrestricted and "unordered" creativity (*progress-by-experience*) and the management's need for organization, clearness and controllability of the development process (*progress-by-planning*). With this approach, we were able to increase the quality of our software development by capturing better the requirements of our customers, and at the same time limiting development time and technical risks of the final product.

Section 2 shows how an incremental and iterative life cycle as consequence of applying evolutionary prototyping looks like from the developer's perspective, and how it can be consistently integrated with the project management's sequential view of the software life cycle. In section 3, evolutionary prototyping as basis for a more sophisticated software engineering is motivated and described. Experiences with two recent, multilateral projects, carried out at TNO Institute of Applied Geoscience, are given and evaluated in regard of development time and risk management in section 4. Section 5 concludes the paper by summarizing assets and constraints of our approach.

2 An Evolutionary Life Cycle: Two Perspectives

The two basic components of a life cycle model are the *set of development phases*, characterized by the activities to be performed in them and their resulting deliverables, and the *execution order* in which the activities are combined to produce these deliverables [LRR93]. In practice, life cycle models are not only used to *describe* different forms of system development. The goal of using normative models is to have a validated *prescription* of the development process and the behavior of the people involved in it. Models used in this way to control a project are more adequately described by the term *life cycle plan* [BKKZ92].

Today, most of the large software development projects follow the line of a life cycle plan. The most common plan is the sequential *waterfall model* [Boe76]. In the waterfall model, the activities are fixed and arranged in a temporal sequence. The basic assumption behind this is that a system can be developed by a sequence of translations of well-defined specifications from an abstract problem description to a program that meets all the necessary quality criteria. The influence that these kind of sequential life cycle plans have exercised on software development in recent years can't be neglected: they match the project management's requirements of a clearly organized and

[1] At the TNO Institute for Applied Geoscience, Delft, The Netherlands, we develop experimental information systems and simulation programs for oil & gas exploration and production (cf. section 4).

verifiable development process. Nevertheless, there are important factors which indicate that this is not the (only) way how software is developed:

- By applying the waterfall model, software development is primarily seen as *technical problem* of stepwise transforming a problem description via a set of specifications into its program equivalent. The *communication problem* between users and developers is mostly totally neglected, as users are only involved at the beginning (requirements specification) and at the end (β-testing). The relatively late feedback can result in a serious mismatch between the users' expectations and wishes regarding the solution of their application-specific problem on the one hand, and the technical interpretation, i.e., the system, offered by the developers.

- Even if there is exchange of ideas, i.e., the specification documents are discussed with users, it has to be taken into account that *formal, non-executable system specifications* are unsuitable for communication between developers and users. As the software changes the working situation, the user needs "material" to visualize the new requirements, and on which to develop creativity and a feeling for the new possibilities. This is where *prototyping* comes into the game.

- Sequential life cycle plans do not meet the developing and the developer's reality: specification and implementation are distinct activities, but they are also tightly related in a *bidirectional* relationship. On the one hand, the implementation realizes the abstract specification of system functionality and structure. But on the other hand, the type of the potential implementation influences what aspects and parts of a system have to be specified [GS93]. The more is known about the projected implementation, the more precise and stable the specification can be, helping to avoid unrealistic or inconsistent requirements. But this kind of "change management" is often not implemented in sequential life cycle plans. Even more, in the case of research & development software (cf. section 4), the appropriate requirements are simply not sufficiently specifiable at the beginning of a project.

- Last, but not least, the gaining attention that is paid to *system maintenance* conceals the true problem concerned with *adapting a system to the application context*. Furthermore, the maintenance phase often falls outside the managed development cycle, resulting in unplanned, instant, bug-and-change-fixing activities.

Considering these factors, the traditional life cycle model has to be revised for projects expected to need more freedom in their development. Within this revision, the set of activities and resulting deliverables does not have to defer from sequential life cycles. But its execution order has to take the iterative and incremental nature of an evolutionary system development into account, without loosing sight of the managerial control aspects.

At TNO, we established a life cycle plan that incorporates relative *unrestricted, but controlled evolution* of the software product, based on an *evolutionary life cycle model*. Note that the base document defining the life cycle plan is a software quality assurance plan [Ger93] that conforms entirely to the ANSI/IEEE Std. 730-1984[2]. Interpretation of this standard is in most cases consistent with the ANSI/IEEE Std. 983-1986[3] and other standards[4].

An overview of such an evolutionary development life cycle can be found in figure 1 to figure 3. In these figures, the iterative approach and its related sequential perspective can be depicted. The

[2]IEEE Guide for Software Quality Assurance Plans [IEE84]
[3]IEEE Guide for Software Quality Assurance Planning [IEE86]
[4]These standards are collected in [IEE89].

inner part of the cycle shows the development process from a technical, developer's perspective. The *network of nodes* indicates *tasks and activities* to be carried out, while *arrows* show their *order and interconnection*. In order to have the desired flexibility of an incremental approach, multiple transitions are possible at certain stages (cf. figure 2).

The *outer circle of deliverables* lists the desired *order of delivery* from the management perspective. Management, software quality assurance, etc. can be seen as taking along this temporal frame. For clarity, not all individual deliverables are shown in this picture. Configuration plan, verification & validation plan, user documentation, among others, are not depicted here, but mentioned in the following description of the life cycle.

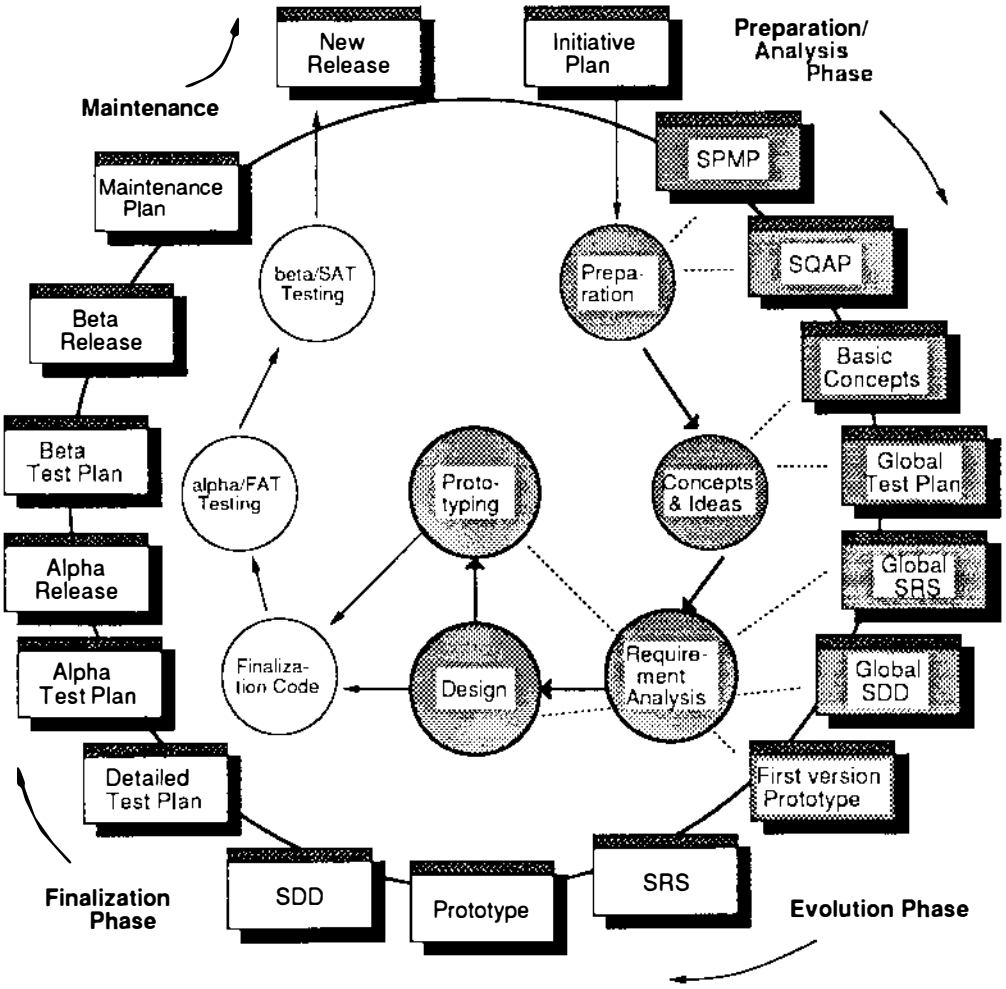


Figure 1: The evolutionary software life cycle, part 1: Preparation/Analysis Phase

The main phases reflect the subdivision of the life cycle into iterating and non-iterating parts: *preparation/analysis*. *evolution*, and *finalization* (plus operation/maintenance).

The developer's perspective

Triggered by the Initiative Plan, during *Preparation*, activities are carried out to produce all deliverables required at the project start. Principle deliverables are the Software Project Mana-

gement Plan (SPMP) and the Software Quality Assurance Plan (SQAP). Also in this phase, the contractual and financial arrangements are settled.

In *Concepts & Ideas*, the Basic Concepts, considered fundamental for the application are gathered, worked out, and documented. Once established, they are not to be changed anymore, as they represent the core ideas of the project. In contrast to the basic concepts, design and implementation concepts may change during the project.

The *Requirement Analysis* activity concentrates on a description of how the resulting software has to look like, and will eventually result in a global Software Requirements Specification (global SRS), together with the result of the first *Design* activity, a global Software Design Description (global SDD). It is crucial that the requirements are *sufficiently* worked out (but do not have to be complete) and concise to start *Prototyping*. The construction of a first prototype concludes the first, non-iterating part of the development process. All these activities (and deliverables) are shown in figure 1 (the gray symbols).

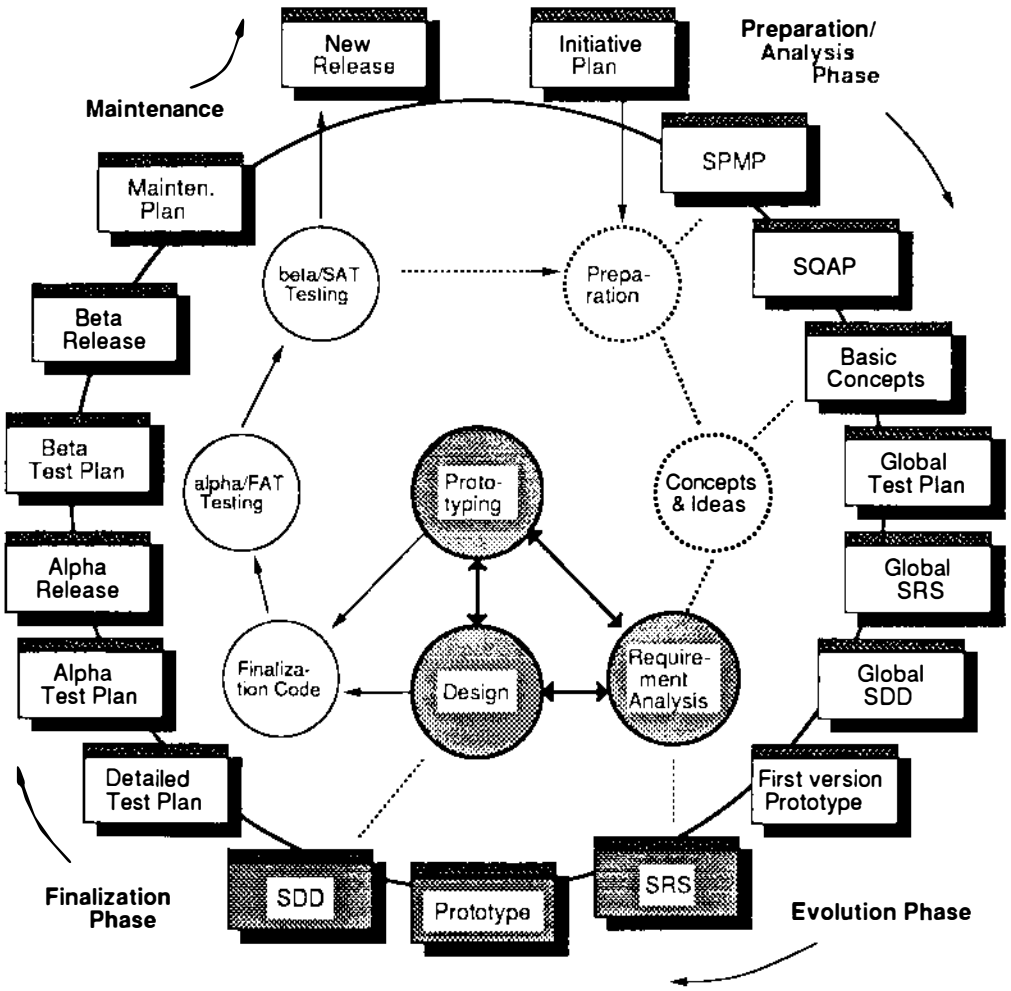


Figure 2: The evolutionary software life cycle, part 2: Evolution Phase

Subsequently, the global system layout, as laid out in the deliverables produced so far, is worked out in full detail. Focus is on those concepts and technical issues that are expected to be most critical for the future system. Attention may easily shift between *Requirements Analysis*, *Design*

and *Prototyping*, as can be seen in figure 2, indicated by the gray process symbols. By this iterative, unrestricted approach, individual parts of the system can further be worked out in a number of cycles of *requirements-design-prototyping*. By using evolving versions of the same prototype as means of communication with the users, the discussion concerning the relative merits of alternative (design and implementation) concepts and solutions can be narrowed down. The special role of the prototype is further discussed in section 3.

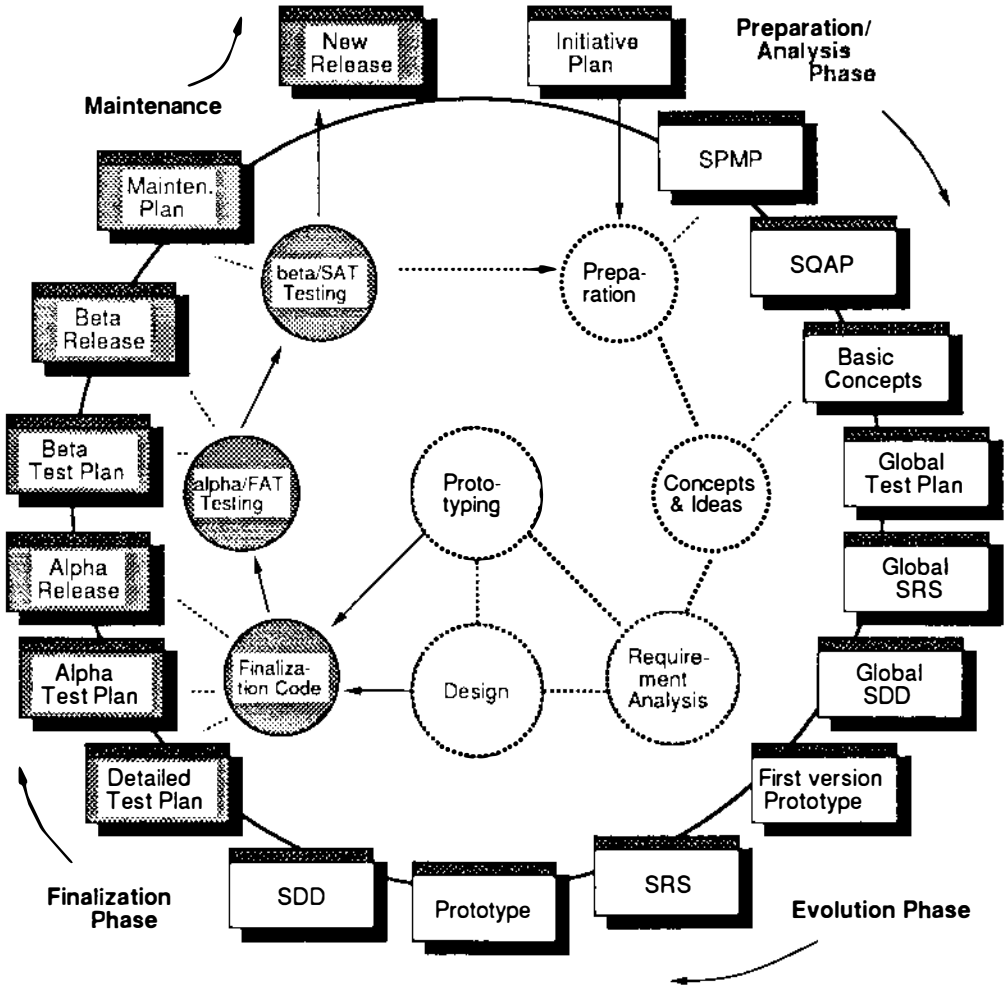


Figure 3: The evolutionary software life cycle, part 3: Finalization/Testing Phase

At the end of this evolution, the code of the incrementally developed prototype is completed and brought into a state of *finalization*, meeting the software quality standards defined in the SQAP.

Acceptance is ensured by α- (factory acceptance) and β- (site acceptance) *testing* (FAT, SAT), and preparations for carrying out maintenance are taken. The tasks and deliverables of this - again sequential - part of the life cycle are marked gray in figure 3.

Operation and maintenance are not shown in figure 3, but close the software life cycle, and can trigger further evolution of the project.

The project management perspective

Whereas the flexibility of the incremental approach of tasks and activities presented above may

appear convenient from a technical perspective, the evolutionary approach is generally too fuzzy and out of control from a managerial perspective. Having a complex item, like a prototype, function as a form of reference "documentation" may also blur the ability to determine exactly the current project status. Therefore, for the management, tasks and activities as well as deliverables need to be put in a *clear and unambiguous time frame* in order to avoid misplanning.

This can be best accomplished by "unfolding" the network of activities in figure 1 to 3, resulting in distinct *project phases* and a clear sequence of deliverables that each can be attributed to a project phase (cf. figure 1 to 3, the outer circle).

The *Preparation/Analysis Phase* emphasizes on a global work out of the concepts, and a global layout of system requirements and system architecture. In figure 1, the shaded deliverables indicate the deliverables completed during this phase. Again, this phase is not intended to be carried out iteratively.

As can be read from figure 1, deliverables include:

• a description of the basic concepts applied • a global SRS • a global SDD • a global Test Plan • a first version of the Prototype

These deliverables typically mirror the global layout of the system so far. In addition, deliverables serving as preparation for further development will be produced:

• an SPMP • a SQAP • Programmer's Guidelines • a Configuration Plan • a Verification & Validation Plan

The *Evolution Phase* hosts the cyclic part of the system development. During this phase, individual parts of the system are analyzed in detail, implemented by, and added to the prototype. SRS and SDD may *no longer be anchor points*, i.e., validated baseline documents, for supervising and controlling development. As consequence, the prototype incorporates the crucial role of representing the state-of-the-art with respect to conceptual solutions and the system architecture. The prototype is used to discuss and validate decisions taken. Checkpoints for the progress of the project are when a "cycle" is finished by evaluation of, and feedback about the last prototype version by the user. Implications for the project managment which arise from this new role of the prototype within the project are discussed at the end of section 4. In figure 2, the deliverables marked gray represent the deliverables of this project phase. At the end of this phase, the

• SRS • SDD

will have reached their final form. Moreover, the code of the

• Prototype

needs to be in such shape that *Finalization*, the next phase, can be started. To work with the prototypes, (short) User Instructions are written, but the final User Manual is not supposed to be completed before the finalization phase, as the prototype might change extensively.

During the *Finalization/Testing Phase*, the system to deliver (and its related products) will be brought in the form prescribed in the SQAP, and performing as described in the SRS. Project management's focus is on the deliverance to and acceptance by the customer. Also in this project phase, the preparation for carrying out maintenance takes place. The gray deliverables in figure 3 indicate the products and documents completed in this project phase:

• User Manual • Detailed Test Plan • α-Test Plan • α-Release • β Test Plan • β-Release
• Maintenance Plan • New Release

The delivery of the New Release usually marks the end of this - sequential - project phase, and of the software development. The *Operation/Maintenance Phase* is covered by the Maintenance Plan, and may evolve in such a manner that the development of a further release or the development of a related system is envisioned.

3 Evolutionary Prototyping

Prototyping offers a practicable solution to the problem of validating and capturing better requirements, hence constructing a more stable system [Bud84]. While *throwaway prototyping* can deliver useful insights about certain limited details of a system, usually not much effort is taken to make those throwaway prototypes qualitatively persistent in regard of software qualities as e.g., efficiency, completeness, etc. [GJM91]. But nowadays, in major projects, the investment in prototyping has become too expensive for the organization to afford to throw the prototypes away. Furthermore, in experimental software development, prototyping has to go beyond addressing single, isolated details of the projected system, but has become an essential means to stimulate imagination and creativity of the non computer-scientific target users. Assisted by new technologies, such as object-orientation, prototyping has matured from offering very limited working models which helped to highlight or validate certain limited aspects of a system (e.g., the user interface) to *evolving, full- scale, and persistent repositories* of acquired knowledge, development solutions and decisions [BKKZ92]. Sophisticated prototyping tools and reuse of (object-oriented) system specifications and components support a rapid construction of systems.

The full set of requirements and possible solutions are often not known or can't be overseen before implementation begins. With a prototype that is constructed early and evolves during the whole development, software concepts can be worked out and implemented in subsequent cycles of realization, testing, and feedback from (early) users to the developers. Concentrating on core functionality and on the new, experimental parts to explore, the current prototype represents at any time the ideas and visions with respect to conceptual solutions and the system architecture.

This new role for prototyping puts additional requirements and constraints on the prototype and on the activity of "prototyping", and hence makes it a principle target for *configuration management* and *software quality assurance*. Important prerequisites of evolutionary prototyping are:

1. *Sophisticated software engineering methodologies* to facilitate a comprehensible and thorough requirements specification (e.g., Use Cases [JCJO92]), and to assist the definition of a solid, but extensible system design (e.g., OMT [RBP+91]).

2. *Object-orientation* - not only for the development methodology which captures the intuitive organization of the embedded knowledge, but also for the implementation. Object-orientation offers the concepts and the components crucial for the construction of the evolutionary prototype, and hence also for the final system. Modularization, reusability, and encapsulation support focusing on the relevant parts of the specification and the system at any time, and to filter out the unimportant parts. Clear implementation of the interfaces between objects is also well supported by object-orientation. Through reuse of existing components, an easier construction of full-scale prototypes is fostered.

3. The evolutionary prototype incorporates much more than the actual version of the implementation. To be able to trace back decisions and to reconstruct previous stages of implementation, *version control* and *careful documentation of the code* has to be supported to bridge the gap in time and understanding between successive development cycles.

4. Users have to be more tightly committed to the development of the system, as regular *evaluation and testing* of the prototype versions are necessary to steer system evolution into the right direction.

As benefit, costs and implications of decisions taken become more visible, risk decreases *earlier* and *faster*. Another important benefit of this form of prototyping is a smoother transition from the design phase to the implementation. The prototype is constructed, kept and evolved throughout the whole design (now: evolution) phase. Thus, the available code on which the implementation is based, is already rather complete and sound, i.e., fulfills required software standards.

4 Experiences With the Evolutionary Approach

In recent years, TNO Institute of Applied Geoscience has developed a number of software systems to manage exploration & production (E&P) data. Many of these applications are unique regarding their innovative, scientific nature, and the knowledge TNO has gathered in this field is reflected by the numerous cooperations with international petroleum and software companies, universities, and standardization organizations.

TNO's activities fall into three categories: research & development projects, consultancy & (pure) production projects, and knowledge transfer. While the latter two pose no, or only low risk of not succeeding, research & development projects have a higher risk of failure, due to their innovative nature of realizing in a software system the results of geoscientific research. These kind of projects lead to so-called *experimental software* (because the success of the resulting system can't be predicted), and are therefore often contracted to TNO. Our institute combines the application specific, i.e., geophysical, knowledge with the information technological skills to cope with the high risks of these projects. It was the particularity of the research & development projects, combining research with the production of operative software system, that triggered the elaboration of a different life cycle plan.

This section will compare the development efforts for two "pure production" projects, realized with a traditional waterfall life cycle plan, with two experimental software projects, developed using the evolutionary approach introduced in the previous sections. Since not only technical data about the products, but also confidential development information is being presented, the real names of the projects have simply been replaced with capital letters.

Project A and B: Information Systems for Reservoir Characterization

The projects presented in this subsection are both information systems that aim at managing information needed for the process of oil & gas reservoir characterization. Both systems run on workstations and offer graphical display functionality on top of a relational database. In both cases, information analysis was carried out with semantic data modeling techniques (cf. table 1).

The fundamental difference between the two systems lies in the specific application area. Unlike the conventional systems for well information, hydrocarbon production and geology (the category of *System A*), *System B* focusses on outcrop information and includes viewing functions in the form of e.g. maps, cross sections and scanned images. As such a system didn't exist in the past, there was no reference system to rely on, and therefore System B's technical risks were much higher than for System A.

Figure 4 gives an overview of the distribution of man-hours for the various development activities of the two projects.

Characteristics:	Project A:	Project B:
project type	pure production	research & development
total duration	18 months	18 months
total man-hours	3100	3600
team size	6	4
platform/ tools	workstation, Unix, OSF/Motif, Uniface, Oracle DBMS, Lotus 1-2-3	workstation, Unix, OSF/Motif, Uniface, Oracle DBMS, XV (image display)
languages	Uniface 4GL	Uniface 4GL
development tools	(ER tool), Uniface	X-NIAM, Uniface
product size	40 entities 249 forms 2.9 mb executable + 19.7 mb Uniface 4GL code	180 entities 197 forms 3.5 mb executable + 14.4 mb Uniface 4GL code

Table 1: Technical data for Project A and Project B.

Management activity hours are not explicitly stated, but attributed to the respective development phases. Although both projects are about the same size (duration, man-hours, platform), it is difficult to compare the absolute numbers directly. Developers, who participated at both projects, subjectively categorized Project B as "2–3 times more complex" than Project A. This appraisal is supported by the complexity of the data model (cf. table 1, no. of entities). So instead of the absolute numbers, the amount of man-hours is given in percentage of the total man-hours of the respective project, which reveals a very interesting picture, too.

In Project A, with the sequential life cycle, a first version of the system was available only in the implementation phase, after more than 70% of the total project time, with the first complete version being ready only after more than 90% of the total project time. This means that feedback about adequacy and performance of the system could occur only at a very late project stage, while the risk of the project was still relatively high. At the same (relative) point of time, Project B was still in the finalization phase, but feedback by the users had already occurred (at least) *three times* (at 17.2%, 50.8%, and 77.8%), not taking into account the α-release finished shortly after (at 94.6%).

Note also, how the percentage of activities during the three evolution cycles of Project B shifts from mainly requirements analysis (`R`) to mainly prototyping (`P`). The small box on the right side of figure 4 gives the total percentage of requirements analysis, design and prototyping for Project B. Design activities in Project B have a considerably bigger share of the total project time than in Project A (15.8%, comp. to 8.1%). This is not only an indication for a more complex system, but is also due to the fact that evolutionary prototyping requires a better *thought-out system architecture* which does not only aim at an optimal final product, but also facilitates reconstruction and change of its subcomponents. Furthermore, Project B already incorporates the principle of reusability, i.e., the aim of constructing, components that would be easier to reuse in future projects - a characteristics that requires good components design.

Interesting are also the numbers for the requirements analysis, which was shorter for Project B (27.5%, comp. to 35.5%). In Project B, requirements analysis was done by a project partner who had little or no computer-science specification or abstraction background. As consequence,

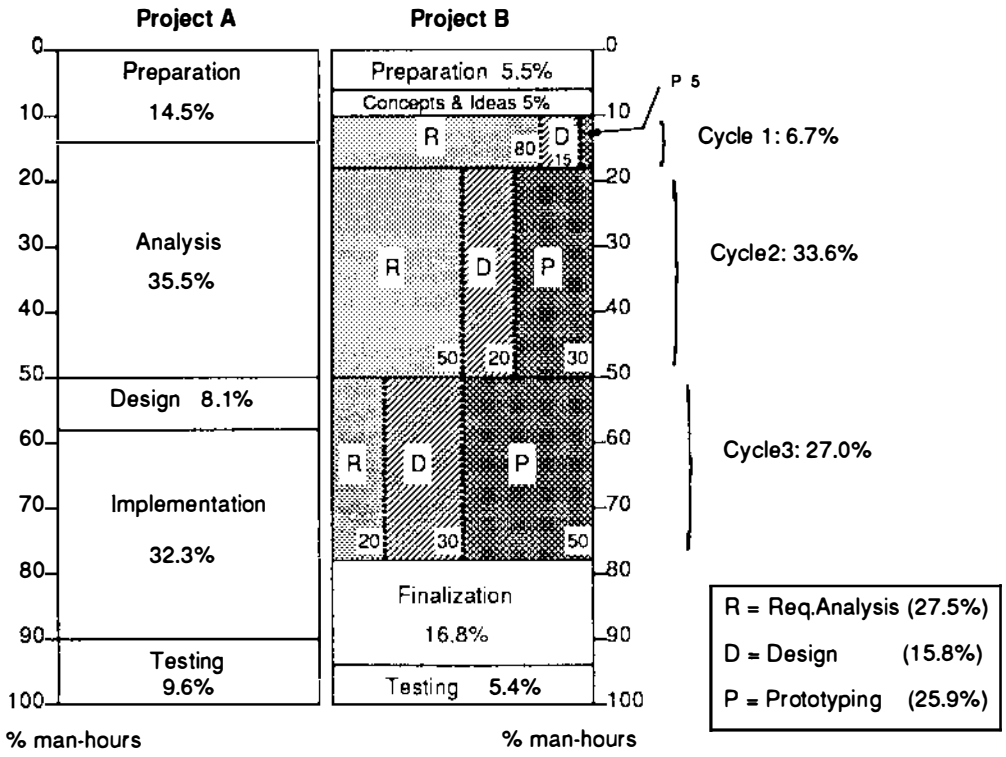


Figure 4: Overview of development activities for Project A and B

the specifications were not so clear and complete, and had to be adapted and revised by showing their implications to the users via the prototypes.

As expected, the finalization phase of Project B is much shorter than the implementation phase of Project A (16.8%, comp. to 32.3%). Finalization consisted mainly of "cleaning up the code". Nevertheless, the total amount of coding activities is higher for Project B, as the prototyping has to be considered as coding, too. This high percentage of coding activities (P + finalization: 40.7%, comp. to 32,3% for Project A), but even more the fact that the α-release of the system was preceded by (at least) three versions of a prototype, required a good version and code management tool. In the case of Project B, this task was sufficiently realized by *Uniface* itself, which was used also as development environment.

Project C and D: Reservoir Characterization by Seismic Data Interpretation

Both projects aim at developing a software system which enables the geoscientist to extract structural and reservoir characteristics from seismic response around an interpreted (artificial) seismic event. The goal is to transform the seismic data into an accurate depth image of the subsurface including the greatest possible detail on stratigraphic, structural features, and above all, rock and pore parameters. While both systems incorporate innovative parts and are comparable in terms of project size (cf. table 2), an innovative combination of stochastic modeling and artificial intelligence, i.e., artificial neural networks, is applied in *System D* to achieve its goal. The approach of using trained neural networks is new and made it necessary to prototype extensively with real data in proprietary case studies, in order to feedback the results of these studies to optimize the software product.

Characteristics:	Project C:	Project D:
project type	production	research & development
total duration	30 months	26 months
costs (overall)	US $ 1.25 M	US $ 1.30 M
total man-hours	7400	9600
team size	5	15
platform/ tools	workstation (UI) + Cray Unix, OSF/Motif, XFaceMaker SU (Seismic Unix)	workstation (UI) + Cray Unix, OSF/Motif, XFaceMaker, *System C* (reuse), SU, WingZ (spreadsheet), Aspirin/Migraine (PD neural network)
languages	C	C, C++, WingZ-Hyperscript
development tools	StP[5], XFaceMaker	StP, XFaceMaker, RCS/CVS, proprietary tools
product size: subsystems (subject of research)	2 (1) 48 mb executable (116 kloc)	5 (4) 38 mb executable (110 kloc) + 2.44 mb worksheets (WingZ) (35 kloc)

Table 2: Technical data for Project C and Project D.

A situation similar to Project A and B holds here: while both projects had about the same duration and costs, and similar platforms, the complexity of Project D, indicated by the number of subsystems that were subject to research, classifies it as *experimental development*.

Figure 5 shows an overview of the development activities for both projects. The same technique as for the other two projects is applied to compare development efforts.

In *Project C*, with sequential life cycle, implementation was available for α-testing after 83.8% of the total project time. The functional tests at this relatively late stage revealed serious performance problems related to the access and processing of the large amount of seismic data. These problems explain the extensive α-test phase (13.5%), in which it was tried to locate the pitfalls that threatened the success of the whole project. After β-testing soon confirmed the shortcomings of the system, it was decided to totally redesign and re-implement System A (this is not indicated in figure 5). With the experiences of Project A in mind, management and developers chose for *extensive prototyping* of System B (44.3% of the total project hours), concentrating especially on the technical part, the performance of the "unknown" neural network. At a comparable point of time (at 82.3% of the total project hours), three prototypes had been tested and evaluated. Thus, finalization and α-testing, like for System B, took much less time than in the sequential reference projects (8.3% finalization, 3.1% α-testing).

The same *shift in activities* from requirements specification (R) to prototyping (P) during the evolution phase is observable here for Project D. The percentage of design activities is again higher than for System C, and the total amount of coding activities, i.e., prototyping + finalization, for System D matches the implementation activities of System C. The difference is, of course, that in System D these coding activities are more "distributed" and intertwined with the requirements analysis, design and feedback by the user. Initially, in Project D, the familiar SCCS-tool for

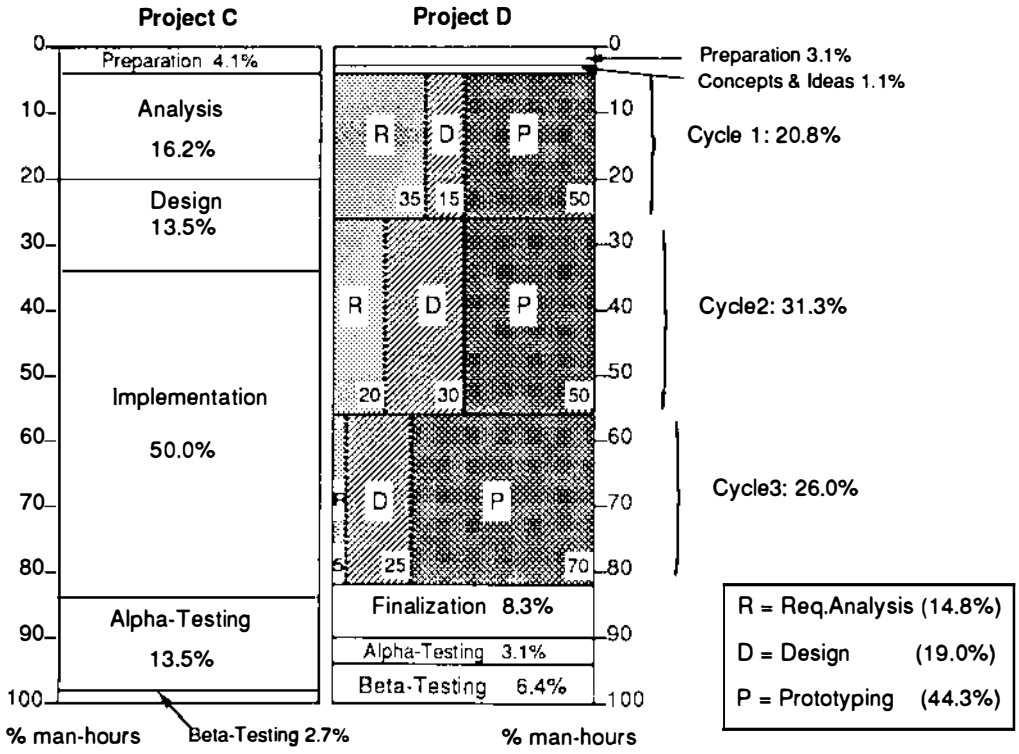


Figure 5: Overview of development activities for Project C and D

configuration management & control was used. Later, it was switched to the use of *RCS/CVS*, as this tool offers a richer set of features and more appropriately allows a team to do concurrent software development. Apart from this tool, a proprietary tool, implemented for a former TNO-project, was used for documentation, extraction of headers, annotating, etc..

5 Assets, Constraints and Conclusions

It truly is difficult to compare the absolute numbers of the projects to come to a conclusion like "we saved *this* amount of time and *that* amount of money with our evolutionary approach." After all, the new life cycle plan was established because we had to cope with new, experimental software systems, for which no reference systems exist.

Assets

What comes closest to an absolute estimation is a cautious statement that with the new approach "we were able to develop more complex systems with the same order of magnitude of resources." Much more important though, is that the evolutionary life cycle plan helped us to decrease technical risks and the risk of not meeting the users' requirements and wishes earlier and faster.

Each version of the prototype and the feedback from testing and users translates directly in a decrease of risk, as sketched in figure 6, where the risk curves for traditional, sequentially phased projects and for the evolutionary development are compared. As consequence, this approach to

system development can increase the confidence of users and developers in the quality of the future product.

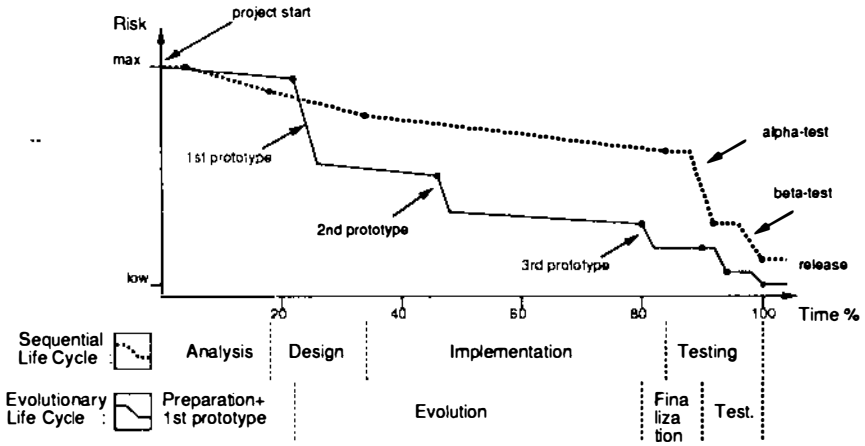


Figure 6: The risk curves for sequential and evolutionary development compared.

Apart from the general, more advantageous risk curve for the evolutionary approach, two details concerning this curve are noticeable. In the beginning, risk decreases slower because analysis and design activities are not as detailed as for the sequential approach. But this changes with every prototype that is evaluated. At the end, the hazard of failure generally is lower for systems developed via the evolutionary approach, simply because, although spread over the whole life cycle, the total of feedback and evaluation of the current development efforts is higher than in traditional projects.

Constraints

Some considerations with respect to the presented evolutionary approach deserve particular attention, as they are vital for the success of the "hybrid" life cycle plan:

- Project management has to pay constant attention to measure precisely enough the *progress of development* during the cyclic evolution phase.
- Deliverables are not necessarily produced in the order presented in figure 1, and specification documents like the SRS and SDD may no longer be anchor points for controlling development prior to the start of prototyping. But concepts, overall requirements, and overall design have to be *sufficiently complete and concise* when coding starts, in order not to hamper the evolution of the system at later stages. To decide about this is again a crucial task for project management and (software) quality assurance.
- The prototype has an additional role as central repository of knowledge concerning concepts, design decisions and their implementation. As consequence, the code has to be *documented and annotated carefully*, and a sophisticated development environment has to *support the different versions and stages* the different parts of the prototype may be in.
- Due to early prototyping, a (global) *test plan* needs to be available in a relatively early stage. Users and non-computer scientists, as e.g., application domain experts have to be *involved in the activities of the evolution phase*. This might not always be so easy to achieve for

usual software development sites. At TNO, which is a large organization[6], it is possible and supported that software constructors and domain experts (geophysicists) work together in one team.

Conclusions

The benefits of our approach to integrate the developer's and the management's perspectives of the software life cycle and of evolutionary prototyping have been confirmed by the outcome of the project described above. This holds for positive results as the sophisticated capturing of requirements and wishes of the users, as well as for the early detection of risks and shortcomings. Still, it is difficult to *measure precisely* the impact of our life cycle plan. The two reasons for this point to the areas of future work concerning our approach:

- *Metrics* have to be provided to describe more concretely the decrease of risk and increase of confidence, which up to know can only be estimated, as sketched in figure 6. The time diagrams (figure 4 and 5) give a good picture of the different stages of the project, but another significant indication would be comparing the number of *change proposals* and their impact on the projects.
- The experimental nature of the systems likely to be developed via the evolutionary approach makes it difficult to *find reference systems*. Uncapturable factors as the experience of the development team and the knowledge of the users, and the complexity of the research part always play an important, individualizing role that can only be excluded by developing *more projects* according to the evolutionary approach.

Furthermore, work is done to formalize and to integrate the approach presented here into a more general methodology which integrates also other aspects of software engineering, such as system architecture and specification techniques [Zam94].

References

[BKKZ92] Reinhard Budde, Karlheinz Kautz, Karin Kuhlenkamp, and Heinz Züllighoven, editors. *Prototyping: An Apporach to Evolutionary System Development.* Springer-Verlag, 1992.

[Boe76] B. W. Boehm. Software Engineering. *IEEE Transactions on Computers*, C25(12):1226–1241, 1976.

[Boe88] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 12(5):61–72, 1988.

[Bud84] R. Budde, editor. *Apporaches to Prototyping.* Springer-Verlag, 1984.

[Ger93] Bart Gerritsen. Software Quality Assurance Plan. Technical Report OS 93-52-C, TNO Institute for Applied Geoscience, 1993.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall, Inc., 1991.

[6] about 4500 employees in 15 institutes

[GS93] David Garlan and Mary Shaw. Architectures for Software Systems. Tutorial Notes, 15th International Conference on Software Engineering, Baltimore, June 1993.

[IEE84] IEEE. *ANSI/IEEE Standard for Software QUality Assurance Plans*, 1984. ANSI/IEEE Std 730-1984.

[IEE86] IEEE. *ANSI/IEEE Guide for Software QUality Assurance Planning*, 1986. ANSI/IEEE Std 983-1986.

[IEE89] *IEEE: The Software Engineering Standards - Third Edition*, 1989.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[LRR93] Perdita Löhr-Richter and Georg Reichwein. Object-Oriented Life Cycle Models. Technical Report 93-05, Technical University of Braunschweig, 1993.

[RBP+91] James Rumbaugh, Michael Blaha, William Premberlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., 1991.

[Zam94] Andreas Zamperoni. Integration of the Different Elements of Object-Oriented Software Engineering into a Conceptual Framework: The 3D-model. Technical Report 94-18, Leiden University, Dept. of Comp. Science, 1994. (also available by *anonymous ftp* from `ftp.wi.leidenuniv.nl` in `/pub/cs-techreports` as `tr94-18.ps.gz`.

Acknowledgements:

We would like to thank the project members and leaders Paul de Groot, Simon Pen and Ipo Ritsema for providing us the useful statistics about the four projects mentioned. Their meticulous man-hour counting won't be forgotten.

Cleanroom Software Engineering: Quality Improvement and Cost Reduction

Michael Deck
Cleanroom Software Engineering, Inc.

Abstract

Cleanroom software engineering has been proven effective in improving software development quality while at the same time improving productivity and reducing cost. This paper compares the practices and results of several recent case studies to show how Cleanroom can be tailored to fit the unique needs of any development project. Current trends in Cleanroom are surveyed, including its use in re-engineering and object-oriented development.

Michael Deck (*e-mail deckm@csn.org*) is a consultant with Cleanroom Software Engineering, Inc., providing assessment, consultation, and training in the use of advanced software engineering techniques. From 1982 to 1993 he was employed by IBM as a Cleanroom team member, team leader, project manager, and technology consultant. He is an ACM and IEEE member and chairs the Boulder, CO, chapter of the ACM. He has published several articles on the theory and practice of Cleanroom software engineering. He holds a BA in Mathematics from Kalamazoo College and an MS in Computer Science from the University of Maryland, and has taught at both institutions. He may be reached by postal mail at 6894 Flagstaff Rd., Boulder, CO 80302, by telephone at (303) 447-0844, and by fax at (303) 447-0538.

1.0 Introduction

Cleanroom software engineering [1][2][3] is a set of principles and practices for software management, specification, design, and testing that have been proven effective in improving software development quality while at the same time improving productivity and reducing cost. The name "Cleanroom software engineering" (or "Cleanroom engineering") comes from the clean rooms used in high-precision manufacturing [4]. There, an environment is created and maintained, at considerable expense, that virtually eliminates production defects. This is done because defect prevention is vastly more cost-effective than defect removal.

Although Cleanroom software engineering has been in use for some time, recent improvements have made it more accessible to a wider variety of projects and companies. This paper describes Cleanroom techniques in three software-development activity areas:

- Cleanroom **specification and design** techniques are based on formal methods. Their use results in reliable systems, components, and parts that are easy to maintain, enhance, and reuse. Though defects are largely prevented, the few errors that do remain can be quickly identified and corrected, without introducing new errors. Cleanroom design techniques are a natural complement to object-orientation. Improved requirements understanding and adaptability to change are also demonstrated benefits of the Cleanroom techniques.

- Cleanroom **testing** techniques focus on certifying the quality of the system, rather than on testing-in quality and defect removal. Cleanroom testing emphasizes the user's view of the system, permitting quality improvement efforts (when necessary) to be targeted at those areas most noticed by the user. A side benefit of Cleanroom testing is the development of an operational profile, which can also be used by designers and managers.

- Cleanroom **management** techniques coordinate the design and testing activities in an incremental, team-based process. Incremental development permits process feedback for quality improvement, and provides end-to-end executable "mini-releases" for requirements understanding and prototyping, early user feedback, and accurate quality prediction. The team focus incorporates the idea of "ego-less" programming, so that teams work together to solve problems that are too large for any individual to solve alone.

This paper examines Cleanroom practices in three time frames, using case studies and experience reports as illustration. It begins by describing the historical foundation of Cleanroom, including early Cleanroom projects and their results. During this time frame, an initial set of Cleanroom practices were used on a small number of projects. Next, the "growth phase" of Cleanroom is described, during which new projects contributed to evolutionary changes in the method. Four recent projects are examined in detail. A third section describes the current status and direction of Cleanroom.

This paper focuses on process issues, and is thus primarily directed at managers and technical leaders. The formal methods that are the foundation of Cleanroom, and detailed discussion of their specific application, are given only limited coverage. Although the focus of this paper is on process and management issues, many of the design, review, and testing practices it presents will be of interest to technical staff members as well.

2.0 Cleanroom Historical Basis: 1981 - 1986

2.1 The Origin of the Name

There are many similarities between Cleanroom software engineering and the hardware clean rooms after which it is named. For example, there is a clear distinction between design and testing. The purpose of design is to create a nearly error-free product; the purpose of testing is to certify the quality and reliability of the product. The line is quite clear – there is no finger-pointing, no assignment of "blame" for failures, and the criteria used by the testers are objectively defined in a formal specification document.

The demonstrated success of the large-scale users of clean rooms, especially the makers of integrated circuits, prompted Dr. Harlan Mills of IBM's Federal Systems Division (FSD) to suggest that the same principles – defect prevention during design, separation of design and test, and objective testing criteria – be applied to software development.

2.2 Foundations

A confluence of theory and practice contributed to the timing of the earliest Cleanroom projects.

- Design techniques. In the late 1970's, IBM FSD had completed large-scale training of managers and developers in the use of top-down, structured design and correctness-verification techniques based on functional verification and state machines [5]. In the federal sector, the importance of defect prevention is perhaps more obvious than in other sectors, because live operational testing is so difficult. The use of modules and abstract data types was also becoming widespread at that time. Functional verification continues to figure prominently in Cleanroom practice, though state machines have largely been replaced by box structures [6][7] and objects.

- Testing techniques. In this same time frame, several researchers were looking into models of reliability that would permit statistical quality control (SQC) techniques to be applied to hardware. The IBM team of Currit, Dyer, and Mills focused their attention on the use of Mean Time To Failure (MTTF) as a key measure of software quality and proposed a testing paradigm that used operational profiles[1], objective testing against specifications, an incremental process, and a reliability growth model to estimate the MTTF, and to use it for in-process feedback and improvement [8]. This testing paradigm is still widely used in Cleanroom today.

- Management techniques. In a process with heavy emphasis on specification and design, the problems already inherent in a "waterfall" life-cycle model are aggravated: If a pure waterfall were to be used in Cleanroom, a year-long project might not see any testing until the 9th or 10th month! The solution is a process consisting of a series of increments, each of which is itself a mini-waterfall [5][8]. Improvements over the organizational structures then prevalent in industry – large, undifferentiated programming staffs – were also made, particularly influenced by the success of Chief Programmer Team techniques [9]. Finally, in line with manufacturing SQC, the design/development and testing functions were separated as much as possible, linked only by an objective, precise, formal specification of the black-box system behavior. The incremental process, organization into small, coherent teams, and separate design and test teams are still important components of Cleanroom management.

It is important to observe that few of the individual ideas, techniques, or tools in Cleanroom are or were revolutionary. Cleanroom has always incorporated many of the best ideas in Computer Science. It does,

[1] The terms "usage specification", "usage distribution", and "operational profile" are used more or less interchangeably in the Cleanroom literature. This paper uses "operational profile".

however, emphasize principles and practices, and not specific tools or languages. If anything is new about Cleanroom, it is the attitude that near-zero-defect software is possible, and that defect prevention is more cost-effective than defect removal. Today, an assemblage of practice and experience exists to support this belief and to demonstrate how to repeat the success of projects that have used Cleanroom.

2.3 Early Applications and Studies

Although the Cleanroom name, and some of its practices, were first published in 1981 [10], it was not described in major journals until 1986 (Currit, et al. [8]) and 1987 (Selby, et al. [11].) Together, these papers describe Cleanroom practices at that time:

- Incremental development process. The development process is organized around accumulating increments of function, each of which is executable in a user-like environment.

- Formal methods of specification and design. Structured specifications, based on the formal models of functions and state machines, restate the requirements precisely yet understandably. Top-down, structured programming is used to implement these specifications through a design that consists of a hierarchy of subspecifications linked by certain programming-language constructs (e.g., WhileDo.)

- Review replaces designer debugging. Execution of programs by designers is virtually abolished. Several studies (including that of Selby) enforced this tenet by preventing even compilation by the designers. Defects are located and removed through review – including verification-based inspection – rather than through execution. This point has proved to be the most remembered aspect of Cleanroom: Designers don't execute their code. Or, more succinctly, "no unit debugging."

- Independent, statistical testing. All, or nearly all, program execution is done by an independent team and is based on statistical models of usage, recorded in operational profiles. The focus of testing is reliability assessment, not error detection, though errors are corrected if found. The primary metric is MTTF.

Selby, et al. conducted an extensive empirical study, in which 15 teams of upper-class and graduate students were assigned the same project. Ten teams used Cleanroom, while five used a "Traditional" approach. The results are striking:

- Most of the developers were able to apply the techniques of Cleanroom effectively, though many of them "missed the satisfaction of program execution."

- The Cleanroom teams' products met system requirements more completely and successfully executed more test cases.

- The source code developed using Cleanroom had more comments and was less complex.

- All of the Cleanroom teams met all of their scheduled intermediate product deliverables, while only two of the five non-Cleanroom teams did.

- Eighty-one percent of the Cleanroom developers said they would use the approach again.

While the Selby study was taking place, IBM was using Cleanroom to develop the COBOL Structuring Facility (COBOL/SF) product.[2]

---

[2] The present author was a member of the COBOL/SF development team.

The COBOL/SF project was the first application of Cleanroom to a real-world, non-DoD project, and the published results [12] are impressive in every respect. Approximately 52,000 lines were modified or added to a base product of approximately 40,000 lines, resulting in about 80 KLOC of PL/I code. Five increments were built, ranging in size from 4 to 19 KLOC. All in-house testing found only 179 errors, of which 111 were in the one largest increment. Although inspection statistics were not kept, reasonable estimates based on then-prevalent industry standards suggest that between 1500 and 3000 errors would have been detected and removed during team verification reviews. More importantly, field testing, during which a half-million lines of customer COBOL code were restructured, found only 10 errors, all of which were simple to find and fix. Only one field-test error was considered "highest severity" by the customer. At the same time, productivity (in lines of code per labor month) was measured at 740 LOC/LM while industry averages were approximately 150 LOC/LM. Although results from the initial base product were never published, the quality experience was similar.

These experiences were enough to convince IBM and several other organizations to begin wider use of Cleanroom techniques.

3.0 Cleanroom Growth Phase: 1987 - present

Beginning in 1987, a number of other organizations, in addition to IBM, began to apply Cleanroom techniques. A satellite control project was developed at NASA Goddard Space Flight Center, reporting low testing failure rates and large improvements in quality and productivity [4]. Martin Marietta developed an automated documentation system (APCODOC) in which no errors were ever reported [13]. IBM projects applied all or part of the methodology in areas such as compilers, image products, database transaction processors, and tape drive microcode [14]. In a large-scale use of the approach, IBM's AOEXPERT/MVS product, comprising 107 KLOC, was developed using Cleanroom principles [15]. In all of these cases, Cleanroom techniques were tailored to fit the project and its environment. These projects have uniformly reported low error rates. When reported, they also indicate productivity at or above expected levels.

Cleanroom evolved during this time to keep up with the changing world of software. For example, the rapid pace of change in language and compiler technology has meant programmers may be learning the implementation language during the first development increment. In such an environment, it may be unrealistic, even imprudent, to strictly enforce a ban on designers compiling or executing code. Teams have sometimes chosen to use Cleanroom design techniques with traditional testing. The design paradigms have grown from strict top-down structured programming to include object-orientation and even rule-based reasoning languages such as TIRS. Users of Cleanroom have adapted it to coexist with a variety of tools and techniques. The ranks of the Cleanroom researchers have grown: The 25th Hawaii International Conference on System Sciences (HICSS-25) in January, 1992, included a minitrack on Cleanroom, and the second annual European Industrial Symposium on Cleanroom Software Engineering will take place in Berlin in October, 1994. Several consulting organizations have been formed to help teams use Cleanroom, and PhD dissertations have addressed special issues with the methodology. Cleanroom is entering the mainstream: An article in *Computer Language* highlighted Cleanroom in a survey of formal methods [16], and the topic has appeared several times on the agenda at major user groups including GUIDE and SHARE.

Space does not permit a close examination of all the projects that have published results, now totalling close to one million lines of code, but an examination of four recently published studies will give some idea of the range of Cleanroom usage.

3.1 NASA Satellite-Control Projects

In 1987, the Software Engineering Laboratory (SEL) of the NASA Goddard Space Flight Center embarked on a study in which Cleanroom software engineering would be applied on a moderate-sized (40 KLOC) but representative project [4][17]. The process typified the Cleanroom of that day:

- Requirements analysis. Produce and review "informal specifications."
- High-level design. Determine build schedule, convert requirements into state machines and functions. Refine and review before proceeding.
- Detailed design. Refine subroutines and module prologues, common blocks, and subroutines. Review before proceeding.
- Coding by increment. Write code from prologues, conduct individual code readings of other developers' code (but not formal verification). Compilation and unit test are prohibited.
- Pretest by increment. Determine operational profile and generate test cases (statistical emphasis, non-statistical if necessary.)
- Statistical testing by increment. Receive code and place under configuration control. Compile, link, and test. Validate correct execution by hand calculation.

This initial Cleanroom effort published several measurements, including:

- The cost of training the team was calculated at 4% of project hours.
- The activity distribution was calculated at 33% design, 18% coding, (of which 45% writing and 55% reading), 27% testing, and 22% other (meetings and other overhead.)

Overall, the use of Cleanroom led to 69% higher productivity, a 45% reduction in error rate, and a large (60-80%) decrease in resource usage as compared to typical projects in the SEL. Other conclusions from the study include:

- "The fundamental reason why developers read the code so thoroughly was that they could not test, so code had to be read."
- "Redundant sequential code reading, although it was time consuming and somewhat tedious, from the developer's point of view, resulted in significant numbers of faults being found and corrected before the code was actually tested. The readers found and corrected an average of 30 faults/KSLOC while code reading. For this reason, the developers did not exercise the option of unit testing their code, which was a part of the fall back plan."
- "Surprisingly, of the 30 executable faults/KSLOC found by the readers, only 28.5% were found by both readers..."
- "Lesson: Cleanroom forces development to be a team effort. Developers cannot depend on the computer (since they cannot test), so they must depend on each other."
- "Lesson: The decision whether to design a system completely and then begin coding, or to design and code by increments must be considered carefully."

A later report [18] adds:

- "...the developers felt that the methodology provided a clearer understanding of the requirements and design, thereby allowing changes to be better analyzed and more easily accommodated."

In 1990, the SEL embarked on a second experiment [18] designed to assess the reproducibility and extendibility of the Cleanroom approach. They also applied the box structures technique, which had not been used in the first experiment. The second experiment involved two projects, one small (20 KLOC) and one large (150 KLOC.) The small project involved SEL employees, while the larger one was subcontracted.

A preliminary report from the second experiment, published in 1991, shows that the initial success was not an aberration. The overall error rate in the second experiment was 3.2 errors/KLOC. Although productivity was not as high as in the first experiment, it was still higher than the previous baseline. One change from the initial experiment, namely the use of box structures, was considered to have "minimal benefit" though other projects (see below) have identified it as a key success factor. In addition, the design teams found the prohibition against compiling to be cumbersome. The SEL is continuing to tailor and adapt the methodology to find the right combination of practices for its unique mission.

3.2 IBM "Comet" Printer Application

The "Comet" project presented two challenges for Cleanroom: Write a new Cleanroom component to be tightly integrated into an existing non-Cleanroom system, and adapt the box structures approach to a multitasking GUI environment (OS/2 Presentation Manager.) These two challenges had a thread of commonality: the new code would be very dense in calls to application programming interfaces (APIs) that were either undocumented, poorly documented, or incorrectly documented.[3]

To apply functional verification, the effect of every statement on the program state must be understood. In a program like COBOL/SF, where there were comparatively few API calls, this is not difficult to accomplish. But in Comet, more than two-thirds of the code statements were API calls to either base system functions or to OS/2 functions. A newly-published extension of box structures [19] was applied, permitting the team to understand and document the API only to the extent needed for correctness verification.

The use of box structures also permitted a user-view black box specification to be written purely in terms of the history of system usage. Although it took considerably longer than planned, this document saved countless hours during the design, implementation, and testing phases. One team member commented, "The level of detail, clarity, and consistency in a Cleanroom specification leaves one wondering how we did our jobs under a more traditional process."[20] The user view black box specification had two distinct components. The "informal part" was a collection of pictures and prose, describing the basic requirements of the entire planned system. The "formal part" was an incremental restatement of the informal part using box structures. Because of the need for process prototyping, and the amount of time required to complete it, the formal specification was done on an incremental basis, but the overall architectural evolution was guided by the informal first component. The user-view black box specification was reviewed by the entire team, including testers and human-factors specialists. All reviews were based on consensus. This meant a bit more up-front time but less wrangling later on.

Design and code reviews were done by the developers, sometimes working in small subteams of two to three. The operational profile was also developed incrementally, and was reviewed by the test team plus a design-team representative.

---

[3] The present author acted as methodology consultant on the Comet project.

Developer testing was strictly limited, though some amount of code execution was needed to discover the function of certain library calls. Generally this was done by constructing and executing small experimental programs. Compilation was done by the developers. Each increment was tested using statistical certification techniques. Because the operational profile was still evolving, MTTF was supplemented with other metrics including errors/KLOC and subjective estimates of quality.

Team enthusiasm about the process was low in the beginning, but ended up extremely high. Distribution of knowledge was such that a team member was able to take a vacation during the coding phase, and another member could complete his section of the code. Reliance on indispensable subject-matter or component "gurus" was substantially reduced.

Although the project was not completed for business reasons, the published results [14] of the first increment are encouraging:

- Size: 6.7 KLOC of C code
- Certification failure rate: 5.1 errors/KLOC
- All but 1.9 errors/KLOC were attributed to the underlying base code and to the unfamiliarity of the team with the environment.

3.3 STARS AMCCOM LCSEC Project (Picatinny Arsenal)

This project was one of several conducted by the US Department of Defense (DoD) Software Technology for Adaptable, Reliable Systems (STARS) program. Cleanroom software engineering is one of the major STARS focus areas.

The AMCCOM LCSEC project at Picatinny Arsenal was chosen to examine both Cleanroom software engineering and process-driven project management. The final evaluation report [21] describes the environment as a "SEI CMM level 1 software support site within the DoD that performs software enhancement tasks for a set of Army weapons systems." Project tasks are carried out by a combination of contractors and government employees. This particular study involved two projects; the published results are from the first increment only.

The process used was a fairly "strict" Cleanroom. It included a very thorough implementation of box structures, elimination of unit debugging, verification in team reviews, and statistical testing based on operational profiles. The final evaluation does not identify specific verification procedures or whether compilation by the designers was permitted.

The preliminary project findings include several items of interest. The following are from the final evaluation report.

- "Picatinny is no exception in that when an organization replaces craft-based practices with engineering-based practices, morale improves."
- "The team-oriented approach of CSE [Cleanroom Software Engineering] saw immediate acceptance and realized both tangible and intangible benefits."
- "It is possible to transfer CSE practices to project teams operating within a SEI CMM level 1 organization."
- "SEI CMM level 1 organizations can realize important benefits from the application of CSE."

- "Cleanroom specification, most notably black box documentation, was cited as being responsible for gains in productivity."

- "Box structured design is credited with focusing the code generation process and with making team reviews more effective. The team enjoyed the orderly process of developing software."

- "Since the process relies a great deal on logical thinking as opposed to programming skill, less experienced programmers are able to take a bigger share of the development burden."

The final evaluation report includes a preliminary estimate of their return on investment (including all training), based on increment 1 data for one of the two projects.

- Overall productivity improvement: 41%

- Return on Investment: 2.43:1

These represent the most conservative estimates. For example, the ROI when excluding training (i.e., the expected ROI for subsequent increments) and when using only government-staffed projects as a baseline (i.e., when used on non-subcontracted work) was projected to be 6.09:1.

3.3 Ellemtel OS32 Operating System

The OS32 project is unique primarily for its size – 350 KLOC of PLEX and C – and for the fact that it is the first major application of Cleanroom to telecommunications switching[22][23].

OS32 is an operating system for new family of Ericsson telecommunications switches. The project took 33 months to complete with a staff of 73. As is common with firmware and microcode projects, the compiler and the processor were developed in parallel with the operating system. In OS32, a "vertical" increment plan meant that each "design object" went through its own incremental life cycle. Then, collections of incremental design objects were organized into "function test" increments. This permitted the design groups a measure of scheduling freedom. Incrementation was cited as giving an "early possibility to monitor and rectify (if problems) on the way, early deliverables, and 'instant reward.'"[22] Because of the size of the team, and the need for many groups to work closely together, a formal review schedule was instituted. Two days per week were devoted to review, the first being for review preparation and the second for actual document and design review. The design was based on SDL, a design language for telecom switches.

All of the design was subjected to "extensive review" but functional verification was not used. Unit testing was prohibited. The bulk of the testing was usage-based (either in simulators or on actual hardware) but statistical certification was not employed.

Team work was encouraged but not enforced – some designers continued to work alone. However, the observed benefits of team work were such that future projects will probably require everyone to work in teams[22].

The project manager's report is emphatic on the benefit of review: "We can e.g. spend 120 days reviewing instead of 120 days debugging without losing calendar time." A review required 5% as much time to find each fault as did function testing. Statistics showed that 99.99% of faults were corrected in review.

The project was three months late in 3 years with no removal of decided functionality, compared to several months delay per year in previous projects.

Compared to previous Ellemtel projects, OS32 saw:

- 114% testing productivity increase (in hours per test)
- 70% design productivity increase
- 157% increase in quality (in faults per sub test)
- 50% improvement in failure rate (in failures per KLOC)
- 0.6% of project resource spent in rework, compared to 5% typical.

4.0 Cleanroom Today

The next section summarizes Cleanroom as it is applied today, and describes some of the new directions in Cleanroom practice. Examples from published results are used to highlight these items.

4.1 The Cleanroom Methodology

Although some of the specific practices have changed over time, the fundamental principles of Cleanroom have remained very much the same [24]:

- **Design Principle**. Programming teams *can* and *should* strive to produce systems that are nearly error-free upon entry to testing.

- **Testing Principle**. The purpose of testing is to certify the reliability of the developed software product, and not to "test quality in."

- **Management Principle**. Incremental, team-based management practices allow in-process feedback for continuous improvement, and limit the scope of human fallibility.

These fundamental principles lead to a number of specific practices, but project-specific tailoring *within the bounds of the above principles* is the rule, rather than the exception.

- Incremental development. An increment is a functional subset of the system that is executable in a user-like environment. This "end-to-end executability" requirement ensures that no effort is wasted in building artificial scaffolding and drivers, and that failure-rate results can be assessed accurately. Incrementation can begin from specification (as in Comet and COBOL/SF, where the early increments were used for prototyping) or from coding (as in case of NASA and OS32.) Making tradeoffs of increment content and size requires judgement and experience.

- Team organization. Cleanroom projects have ranged in size from a few people to dozens. The smallest effective team size is usually cited as 3 persons, but the maximum is partly determined by logistics – the larger the team, the harder it may be to coordinate review schedules and find meeting rooms. Usually a "project team" will act as a unit for specification activities, since both the designers and the testers must understand the specification. Thereafter, the design team and the test team act independently except for brief feedback, specification, and planning sessions between increments. On a large project (e.g., OS32) a structure involving many teams will be required to coordinate the efforts of many developers. This large structure is likely to be organized along architectural lines, perhaps with a small team owning each subsystem, and a team of team-leaders coordinating the overall project.

- Formal methods of specification and design. Although the original Cleanroom practice required formal methods for specification and design, practical usage has suggested that the level of formalism can vary widely from project to project and even within a single project. OS32, for example, used the design language SDL to capture most of its design. The first NASA project

used informal specifications together with state machines and functions. Comet and Picatinny used a more rigorous box structures approach. Each team must discover for itself, guided by experience when possible, what level of formality is required. A good rule of thumb is to continue to add formality while there is still difference of opinion about what is meant in a particular area or module.

- Intensive review. Intensive team review has become the consensus of Cleanroom usage. The variable in practice is the level of formality, which to some extent is tied to the choice of formal methods for specification and design. A team that does not use functions to specify programs and program parts cannot effectively use functional verification to assess correctness. However, inspections (ranging again in formality from the "thorough inspections" of OS32 to moderated inspections in the style of Fagan [25]) can be applied in any case. The most frequent choice is to use some level of functional specification plus a "verification-based inspection" as documented by Dyer [3]. No particular standard of review practice predominates among Cleanroom teams. Some teams have used techniques borrowed from Fagan, with roles like "moderator" and "reader." More often, however, the designer of a particular piece acts as reader and note-taker, while the team leader acts as moderator or facilitator. The focus, too, is different from that of Fagan. Instead of looking for bugs, the review is intended to convince the entire team that the code (or specification) adequately meets all the important design and correctness criteria.

- Testing. Testing is the area where the widest difference in practice exists. The consensus among nearly all Cleanroom teams is that the testing function and the design function should be as independent as is practical. However, the choice of boundary between the testing function and the design function is significant. A few teams (e.g., NASA and the Martin Marietta APCODOC project) have considered first-compilation to lie on the testing side. Most other projects, however, have placed the burden of clean compilation on the designers. In most Cleanroom applications, design change control begins when an increment is handed over to the test team. Prior to that, change can be made for any valid reason as long as the team has reviewed and agreed to the change. Unit testing by developers is almost universally ruled out, though experimentation for the purpose of understanding a language or interface element is usually permitted, and the distinction between the two can be a narrow one. The normal progression is for programmers to feel a "sense of loss" in the first increment, but to be convinced by the other benefits afterwards. Fault isolation and correction during the testing process may be done through traditional debugging techniques or purely through re-review. The review of any proposed fix is a must, to prevent introduction of new failures and to apply defect prevention techniques including root-cause analysis. About half of the teams that have published results have used profile-based testing; none have published MTTF data. The remaining teams have applied traditional black box testing including system testing and claims testing. The majority of teams that do certification have used the Currit, et al. "Certification Model" but newer models are entering use [26].

In sum, there is enormous variability of Cleanroom practice, within the boundaries defined by the fundamental principles. In the next sections, we look more closely at a few areas where Cleanroom is changing most rapidly.

4.2 Phased Approach to Technology Introduction

New recommendations [27][28] for phasing-in the use of Cleanroom techniques will make adoption easier for most projects. One of the primary concerns about introducing any new technology is the possible adverse impact of that introduction. It is frequently feared that the time between the investment and the payoff is too long, and the up-front investment is too large. By phasing in the Cleanroom approach, each incremental investment will be smaller, and will be recouped earlier.

Every development team operates in a different environment. Many will have prior commitments that make adopting a new approach difficult. Others will be learning a new language, or adapting to new colleagues or managers. Most teams will want to phase-in a Cleanroom approach, so as not to disturb prior commitments and not to cause too much upheaval. Trammell [29] has suggested one path, based on the IBM AOEXPERT/MVS experience; what follows is a slightly different approach.

The phase-in begins with an assessment of a team's (or company's) current processes and technologies. This is focused on determining the major software development costs and risks, and the areas in which Cleanroom techniques are most required.

Technology introduction is done in three areas: management; specification, design, and review; and testing. The phase-in progression can proceed at different rates in the different areas.

In the management area, the focus is on forming and encouraging teams, and on incremental development planning. As technology introduction progresses, both of these (especially incrementation) will become more refined. All of the case studies have used team organization and incrementation, though the incrementation tactics have depended on the risk profile of each project.

Specification and design phase-in begins with the basics and grows into formal methods. The first step is to use hierarchical design structures and to express intended behavior. Formal notations are not necessary at first; precise natural language will suffice. This level supports review and inspection but not functional verification. All of the case studies have incorporated at least this level.

The second development step is to apply techniques of process and data abstractions, including state machines and intended functions. This level is required before functional verification can be used. The COBOL/SF project and the first NASA project were at this level.

The next development level uses more formal notions of data and function abstraction. This level supports proofs of correctness to be applied where necessary. Box structures, either with or without object-orientation, were the choice of several projects, including the second NASA projects and Comet. There have been studies involving the use of Z, but to date there have been no published project studies.

In testing, the first step is to abolish all, or nearly all, testing by the designers. Such "debugging" (sometimes institutionalized as "unit testing") runs counter to the principle that testing is for reliability certification, not defect removal. By prohibiting developer testing while at the same time insisting (as a management standard) on near-zero-defect designs, the designers are forced to actively review all code. All of the case studies did this; NASA and APCODOC went as far as to prohibit compilation by designers as well. However, some projects permitted experimentation especially when trying to understand languages or interfaces.

The next testing step introduces incremental testing based on usage. A general operational profile is developed to guide the testing process, and testcase success or failure is judged against the specification. All of the projects did this.

The third testing level adds statistical estimation of MTTF. This requires an operational profile that is close to actual usage, the collection of interfail data, and the use of reliability models. COBOL/SF and Comet were at this level; the testing protocols of the Picatinny project are not documented. In COBOL/SF, the operational profile was developed incrementally. The first increment's profile was a rough approximation, and detail was added in successive increments, with beta-test acting as the final step in that process.

4.3 Object-orientation and Cleanroom

Perhaps the fastest-growing segment of software engineering involves the use of object-oriented techniques. Recent articles [30][31][32] have proposed ways to incorporate the object paradigm into Cleanroom projects (or, vice versa, to apply Cleanroom practices to object-oriented development.) The use of a box-structures hierarchy (to complement the object/class hierarchy) provides direction and organization to the development of objects.

One major difference between object-orientation and its predecessors is the importance of data abstractions and implementations. In early "structured" methodologies, there were a few, large, high-level data abstractions such as "file" or "database" and the bulk of the effort in design went into implementing the processes (e.g., update the master database using the transaction files.) In object-oriented approaches, a much greater emphasis is made to define hierarchies of objects (including inheritance and containment hierarchies.) So one would define the database as an indexed collection of tables, each table containing rows, each row containing cells, and all of these objects – database, table, row, cell – would have an abstract specification and a concrete design. Languages have also played a crucial role, making it much easier to implement such object hierarchies.

The data-abstraction and verification practices of box structures have of late become sufficiently streamlined to permit practical verification of the large number of data abstractions found in object-oriented development. A project at IBM's Santa Teresa Lab undertook the concurrent introduction of Cleanroom and object-orientation into a team that had previously used neither. A pilot "first increment" demonstrated a natural synergy between Cleanroom and object-oriented techniques.[4]

4.4 Reliability modeling

Another area where Cleanroom is advancing is in testing technologies. Earlier papers (e.g., [2]) primarily focused on black-box testing driven by operational profiles. However, Poore et. al [26] have suggested that reliability testing at the component level is compatible with Cleanroom. They have proposed a technique based on Markov chains, and have suggested a "sampling model" to complement the certification model traditionally used in Cleanroom. To date, published studies do not provide enough data to compare the usefulness of these two approaches. The OS32 project used a modified Markov approach, and Wohlin [33] reports considerable satisfaction with its use.

4.5 Re-engineering support

Historically, Cleanroom has struggled with the mis-perception that Cleanroom is only useful for new code. While it is true that the focus of most Cleanroom papers has been on new development, the idea that function abstraction could be used to capture the essential behavior of structured programs is not new [34][35]. Indeed, one of the commercial purposes for COBOL/SF was to convert unstructured COBOL into structured form for subsequent function abstraction. The Comet project and an IBM tape microcode project (described in Hausler, et al. [28]) show how function abstraction can be used to understand both software and hardware interfaces. Unfortunately, the lack of tools support for true function abstraction (as opposed to data-flow modeling) has hindered this effort, since it is otherwise a tedious and time-consuming process. Nevertheless, in many instances it may be cost-effective when applied to a strategic component or interface.

It is also important to note that "new code" does not necessarily mean "new system." In the NASA, Comet, and Picatinny projects, the Cleanroom development produced a new part that would be integrated into a mass of non-Cleanroom code. Comet, in particular, demonstrated that this can be done effectively even when the interface between new and old is quite wide.

---

[4] The present author acted as methodology consultant on this project.

5.0 Conclusion

Although much work remains, recent developments show that Cleanroom software engineering is successfully improving quality and reducing development cost in a variety of environments and on a wide spectrum of projects. What they also show is the need for judgement and experience to be used in tailoring and phasing-in the use of Cleanroom projects. Cleanroom has moved into the mainstream of software development; its benefits are now within the reach of nearly every software project.

6.0 References

1. Mills, H.D., M. Dyer, and R.C. Linger, "Cleanroom Software Engineering," *IEEE Software*, September, 1987, pp.19-25.

2. Cobb, R.H., and H.D. Mills, "Engineering software under Statistical Quality Control," *IEEE Software*, November, 1990, pp. 44-54.

3. Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley, 1992.

4. Kouchakdjian, A., *Lessons Learned Using Cleanroom Software Engineering in the Software Engineering Laboratory*, MS Thesis, University of Maryland, College Park, 1990.

5. Mills, H.D., D. O'Neill, R.C. Linger, M. Dyer, and R.E. Quinnan, "The Management of Software Engineering," *IBM Systems Journal*, vol. 19, no. 4, 1980.

6. Mills, H.D., R.C. Linger, and A.R. Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, 1985.

7. Mills, H.D., "Stepwise Refinement and Verification in Box-Structured Systems," *IEEE Computer*, June, 1988, pp. 23-36.

8. Currit, P.A., M. Dyer, and H.D. Mills, "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, (January, 1986), pp. 3-11.

9. Baker, F.T., and H.D. Mills, "Chief Programmer Teams," *Datamation*, December, 1973.

10. Dyer, M., and H.D. Mills, "The Cleanroom Approach to Reliable Software Development," in *Proc. Validation Methods Research for Fault-Tolerant Avionics and Control Systems Sub-Working-Group Meeting: Production of Reliable Flight-Crucial Software*, Research Triangle Institute, NC, Nov. 2-4, 1981.

11. Selby, R.W., Basili, V.R., and Baker, F.T., "Cleanroom Software Development: An Empirical Evaluation," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 9, September, 1987, pp. 1027-1037.

12. Linger, R.C., and H.D. Mills, "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility," *Proc. 12th International Computer Science and Applications Conference*, October, 1988.

13. Trammel, C.J., Binder, L.H., and Snyder, C.E., "The Automated Production Control Documentation System: A Case Study in Cleanroom Software Engineering," *ACM Trans. on Software Engineering and Methodology*, vol. 1, no. 1, January, 1992, pp. 81-94.

14. Linger, Richard C., "Cleanroom Software Engineering for Zero-Defect Software", *Proc. 15th International Conference on Software Engineering*, May, 1993.

15. Hausler, P.A., "A Recent Cleanroom Success Story: The Redwing Project," *Proc. 17th Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, December, 1992.

16. Keuffel, W., "Clean Your Room: Formal Methods for the '90s," *Computer Language*, July, 1992, pp. 39-46.

17. Green, S.E., A. Kouchakdjian, and V.R. Basili, "Evaluation of the Cleanroom Methodology in the Software Engineering Laboratory," *Proceedings of Fourteenth Annual Software Engineering Workshop*, NASA, Goddard Space Flight Center, Greenbelt, MD 20771 (November 1989), pp. 1-22.

18. Green, S.E., and R. Pajersky, "Cleanroom Process Evolution in the SEL," *Proceedings of Sixteenth Annual Software Engineering Workshop*, NASA, Goddard Space Flight Center, Greenbelt, MD 20771 (December 1991), pp. 47-63.

19. Deck, M.D., M.G. Pleszkoch, R.C. Linger, and H.D. Mills, "Extended Semantics for Box Structures," *Proc. 25th Hawaii International Conference on System Sciences*, 1992, pp. 382-393.

20. Boldt, G.D., "Cleanroom Software Engineering, and How it is Used," unpublished manuscript.

21. STARS ID52 Final Evaluation Report, *STARS CDRL 005503-004* (July, 1993).

22. Tann, L-G., "OS32 and Cleanroom," *Proceedings of 1st Annual European Industrial Symposium on Cleanroom Software Engineering* Copenhagen, Denmark, 1993, pp. 1-40.

23. Linger, R.C., "Cleanroom Process Model," *IEEE Software*, March, 1994, pp. 50-58.

24. Deck, M.D., and P.A. Hausler, "Cleanroom Software Engineering: Theory and Practice," *Proc. Software Engineering and Knowledge Engineering '90*, June, 1990, pp.71-77.

25. Fagan, M., "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, vol. 15, no. 3 (1976) pp. 219-249.

26. Poore, J.H., Mills, H.D., and D. Mutchler, "Planning and Certifying Software System Reliability," *IEEE Software*, January, 1993, pp. 88-99.

27. Trammell, C.J., P.A. Hausler, and C.E. Galbraith, "Incremental Implementation of Cleanroom Practices," *Proc. 25th Hawaii International Conference on System Sciences*, January, 1992, pp. 437-448.

28. Hausler, P.A., Linger, R.C., and Trammell, C.J., "Adopting Cleanroom Software Engineering with a Phased Approach," *IBM Systems Journal*, March, 1994.

29. Trammell, C.J., P.A. Hausler, and C.E. Galbraith, "Incremental Implementation of Cleanroom Practices," *Proc. 25th Hawaii International Conference on System Sciences*, January, 1992, pp. 437-448.

30. Hevner, A.R., "Object-Oriented System Development Methods," *Advances in Computers*, vol. 35, Academic Press, 1992, pp. 135-198.

31. Hevner, A.R., and H.D. Mills, "Box-structured methods for systems development with objects," *IBM Systems Journal*, vol. 32, no. 2, 1993, p. 232-251.

32. Deck, M.D., "Using Box Structures to Link Cleanroom and Object-Oriented Software Engineering," *Technical Report 94.01b*, Cleanroom Software Engineering Associates, Boulder, CO, 1994.

33. Wohlin, C., "Engineering Reliable Software," *Proceedings of Fourth International Symposium on Software Reliability Engineering*, November, 1993, pp. 36-44.

34. Linger, R.C., H.D. Mills, and B.I. Witt, *Structured Programming: Theory and Practice*, Reading, MA: Addison-Wesley, 1979.

35. Basili, V.I., and H.D. Mills, "Understanding and Documenting Programs," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 3, (May, 1982),pp. 270-283.

Software Process Security and ISO 9001: A Proposed Integration

Edward G. Amoroso
W.E. Kleppinger
Philip K. Sikora

AT&T Bell Laboratories
67 Whippany Rd., Rm. 15A-345
Whippany, NJ 07981
[e.amoroso, w.kleppinger, p.sikora]@att.com

Abstract

A security analysis of the software process is performed to identify a collection of threats that exist during software design, development, and maintenance. A security coverage analysis is used to determine the degree to which these threats are addressed in the existing ISO 9001 clauses. A collection of new security-oriented requirements is proposed to mitigate identified security risks. The potential implication of this research investigation on future ISO 9001 cerifications are summarized.

Keywords and Phrases: ISO 9001, software process, software security, software quality, software threats.

Biographical:The authors are members of the Secure Systems Engineering Department of AT&T Bell Laboratories. The SSE Department offers a variety of security-related services and products to assist in the prevention, detection, and recovery from security threats, vulnerabilities, and attacks.

Ed Amoroso is a Distinguished Member of Technical Staff. He is currently performing system security engineering for the U.S. Army on its Sustaining Base Information System (SBIS) project. He has also been involved with the Department of Defense Trusted Software Methodology (TSM) and the AT&T System V/MLS operating system. Dr. Amoroso joined AT&T in 1985 and holds a B.S. in physics from Dickinson College, Carlisle, PA, and an M.S. and Ph.D. in computer science from the Stevens Institute of Technology, Hoboken, NJ.

W. E. Kleppinger is a Member of Technical Staff. He is currently performing system security engineering on a variety of different government and international projects. Dr. Kleppinger joined AT&T in 1985 and holds an A.B. in physics from Princeton University, Princeton, NJ, and an M.S. and Ph.D. in physics from Stanford University, Stamford, California.

Phil Sikora is a Member of Technical Staff. He is currently performing system security engineering on a variety of different projects. Mr. Sikora joined AT&T in 1986 and holds a B.A. in education from Kean College of NJ, and an M.S. in computer science from Polytechnic University.

Software Process Security and ISO 9001: A Proposed Integration

Edward G. Amoroso, W.E. Kleppinger, and Philip K. Sikora
Secure Systems Engineering Department, AT&T Bell Laboratories
Whippany, N.J. 07981 - USA
e.amoroso@att.com
w.kleppinger@att.com
p.sikora@att.com

Abstract

A security analysis of the software process is performed to identify a collection of threats that exist during software design, development, and maintenance. A security coverage analysis is used to determine the degree to which these threats are addressed in the existing ISO 9001 clauses. A collection of new security-oriented requirements is proposed to mitigate identified security risks. The potential implications of this research investigation on future ISO 9001 certifications are summarized.

1 Introduction

A recent trend in the software community has been towards greater attention to process issues during the software design, development, and maintenance lifecycles. This is evident in the various software process models that have emerged to guide organizations toward process improvement. Examples of such process standards include the Software Engineering Institute's Capability Maturity Model [1] and the International Standards Organization's ISO 9001 requirements [2]. Both of these standards promote enhanced lifecycle activities toward higher software quality. An implicit assumption in the application of these models, however, is that software quality deficiencies originate as non-malicious, inadvertent errors. The work reported here removes this assumption and investigates security issues in the software process.

In particular, considerable evidence is becoming available that suggests that the software lifecycle is vulnerable to a wide range of different malicious attacks. (Readers interested in a complete taxonomy of software security risks and their associated attributes are referred to the *Risks to the General Public* forum moderated by Peter Neumann [3]). As a crude generalization, one might suggest that defense software environments must contend primarily with attacks by intruders who possess nationalistic or political agendas, whereas commercial software environments must contend primarily with attacks by disgruntled employees and competitors. One must be careful with this type of generalization,

however, since disgruntled employees in defense environments have also successfully initiated damage, as have politically motivated intruders in commercial environments.

A recent process standard that focuses on security has emerged in the U.S. Department of Defense Ballistic Missile Defense Organization (BMDO) as part of the Global Protection Against Limited Strike (GPALS) project. The process standard is known as the Trusted Software Methodology (TSM) [4] and is based largely on the unique attributes and concerns associated with military software development. The TSM offers guidance to software organizations by presenting a collection of *trust principles* that are organized into a hierarchy of trust classes. These classes provide a means for specifying process security and quality requirements and for comparing the trustworthiness of software that results from different development processes.

The TSM requirements were written using DoD-STD-2167A [5] as an assumed underlying software process model, since it was developed specifically for use in military environments. In fact, various TSM trust principles refer liberally to specific concepts and terminologies used in DoD-STD-2167A (e.g., "*. . . all Critical Design Reviews (CDRs) and Preliminary Design Reviews (PDRs) shall . . .*"). While this emphasis on the defense model has eased the transition for military software organizations from their existing processes to more trustworthy approaches, it has also made it more difficult for commercially oriented software organizations to utilize the trust principles and classes. As the authors of the TSM, we recognized that the integration of security issues into a commercial process would require more than just suggesting that the TSM be used in general software process settings.

Since the recent ISO 9001 process requirements and the companion ISO 9000-3, Part 3 [6] interpretation for software have become de-facto international standards for the software process, we conjectured that perhaps the same procedure used to integrate security into DoD-STD-2167A could be used for ISO 9001. Thus, the major goal of our research emerged: namely, that **the basic integration principles identified during the creation of the TSM for defense processes could be used to integrate security technology into ISO 9001 for commercial and other types of software development and maintenance**. These integration principles included:

- A *separation principle* that suggested that the new security enhancements for ISO 9001 should maintain its original quality standard as a well-defined subset;
- A *justification principle* that suggested that any new enhancements be justified by a specific software process malicious threat;
- A *minimization principle* that suggested introducing enhancements that only addressed new requirements for security, rather than include redundant requirements that might be used to mitigate threats.

The purpose of this paper is to report on our experience and results integrating security into ISO 9001. It details the security analysis that was performed on the software process to determine a baseline set of threats in commercial software design, development, and maintenance. The paper describes, with respect to the existing ISO 9001 clauses, the manner in which the threats that resulted from this analysis were used as the basis for a security coverage analysis. The intent was to determine the degree to which the existing

clauses provided security safeguard and countermeasure protection. It was found that the existing standard was lacking in several important security areas. To address these deficiencies, a collection of new security oriented requirements is proposed in this paper, using the terminology and style used in the ISO 9001 document and its companion ISO 9000-3, Part 3. The manner in which the steps of our analysis fit together is shown in Figure 1.

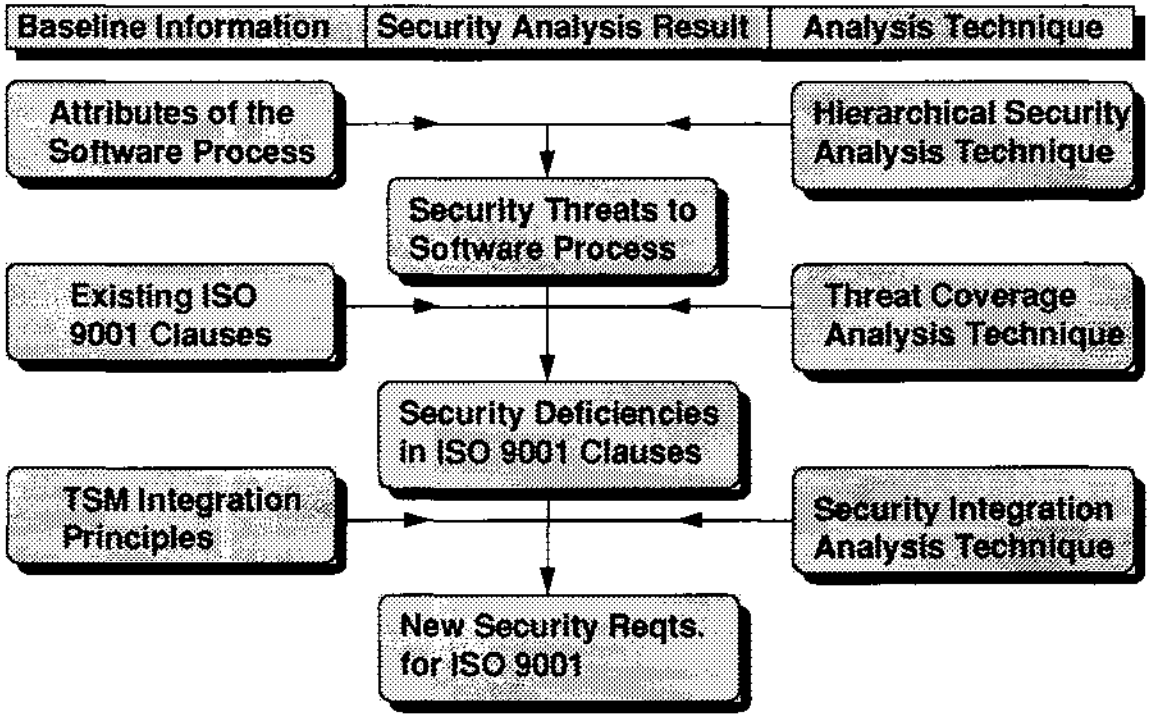


Figure 1. Summary of Analysis Steps

The resulting collection of new requirements clauses can be used by any software organization that is attempting to achieve ISO 9001 compliance, and or also wishes to install suitable measures for countering malicious attacks to software during design, development, and maintenance.

2 Summary of ISO 9001

Since the primary focus of this work is to examine and enhance the degree to which the ISO 9001 standard addresses security in the software process, it is important to provide a baseline description of the ISO 9001 requirements. ISO 9001 was originally intended as a general quality standard for organizations designing, developing, and maintaining products (ISO 9002 and 9003 exclude design and development issues [7, 8]). The standard is expressed as a collection of twenty process requirement clauses that offer desired process attributes for quality. Thus, an organization that wishes to be formally certified and registered by a delegated authority as ISO 9001-compliant must adjust its software process to meet the requirements in these clauses.

Unfortunately, since ISO 9001 was not written with software specifically in mind, a degree of interpretation is required to determine whether a given requirement clause is being met during the software process. To manage this problem and promote more uniform interpretations, a companion document known as ISO 9000-3, Part 3 was created to interpret the requirements' clauses in terms that are more familiar to the software practitioner. For example, a clause is included in ISO 9001 for "Product Identification and Traceability" that may not be immediately familiar in the context of software. The ISO 9000-3, Part 3 interpretation, however, explains that this requirement should be considered in terms of configuration management, which is certainly a familiar concept for software.

Readers interested in ISO 9001 and ISO 9000-3, Part 3 are urged to examine the standards directly. For convenience here, a brief summary (in the authors' words) of the twenty clauses is offered in Appendix A.

3 Software Process Threats

We begin by defining a *threat* as any occurrence, malicious or inadvertent, that can cause harm to the resources and assets of the software organization during the software process. The types of threats that we address include:

- Disclosure of code or algorithms to an adversary;
- Changes or destruction to any aspect of the software process by an intruder or malicious developer;
- Or denial of software resources to authorized personnel.

Traditionally, such threats are identified by brainstorming sessions, but we prefer to follow a more engineering oriented approach here.

That is, in order to identify the possible threats to the software process, we will create a hierarchical threat tree structure as described in previous works [9, 10]. The goal of a threat tree is to characterize threats by partitioning the class of all threats into demonstrably complete subclasses. By *demonstrably complete*, we imply that proof may be offered that the partition does not leave any threat out of an associated collection of subthreat classes. This is easy to obtain in practice by including a default subclass for a given threat. However, an engineering goal in the application of threat trees to a problem is to reduce dependence upon such non-productive steps.

We will represent threat trees in terms of class levels that will be referred to by a simple indexing convention where C is the first class level; C[1], C[2], and so on, comprise the second; C[1][1], C[1][2], and so on, comprise one refinement branch of the third level, etc. Thus, since each class in the hierarchy represents a set of threats, we are obliged to define the contents of each threat in a given class. Relations on different levels and threats will be identified as we proceed with the exercise. We begin by identifying the highest level, C, as comprising the class of all threats to the software process:

C: Class of threats to the software process

This first threat class is offered more as a *starting point* than as a useful engineering step. A second class of threats can now be identified and associated with a subset relation with respect to the first class level. That is, the union of all classes in the second level equals the class represented in the first level. In fact, the subset relation between successive levels will be implicitly assumed in all subsequent refinements in this exercise.

In addition, another relation on the various classes within a particular level will also be implicitly assumed. Specifically, the set of classes within a level will be associated with an equivalence relation so that any subclass of threats is viewed as an equally valid instantiation of the higher class. This is not to imply that we will not add prioritizations to the subclasses based on environmental attributes.

The second class level will correspond to the traditional three types of threats: namely, *disclosure* of information related to the software process (e.g., source code, documentation), *integrity* of resources associated with the software process (e.g., software tools, developed code, tests), and *denial of service* to software process resources (e.g., availability of documentation, tools). We use a simple indexing convention to refer to the elements of this class.

C[1]: Disclosure threats to software process
C[2]: Integrity threats to software process
C[3]: Denial of service threats to software process

The third class level of threats will correspond to *the various phases of the software process*. This is a demonstrably complete refinement because threats would have to occur during at least one phase of the process. Threats that occur during multiple phases can be identified with any of the phases in the set of classes (e.g., by convention, threats in multiple phases can be associated with the phase in which the threat is initiated). The specific software process phases to be included will be the (1) requirements analysis phase, (2) design phase, (3) development phase, (4) test and evaluation phase, (5) integration phase, and (6) maintenance phase.

C[1][1]: Disclosure threats to software process during requirements analysis phase
C[1][2]: Disclosure threats to software process during design phase
C[1][3]: Disclosure threats to software process during development phase
C[1][4]: Disclosure threats to software process during test and evaluation phase

.
.
.

C[3][5]: Denial of service threats to software process during integration phase
C[3][6]: Denial of service threats to software process during maintenance phase

The fourth class level is designed to isolate the *intent* associated with a particular threat class. This is usually associated with attack taxonomies, rather than threat trees because it is focused on the approach to enacting a threat, rather than just the effect of the threat. Nevertheless, we have generally found this to be a useful refinement in this type of

analysis. The subclasses at this level correspond to the threats initiated by innocent software development errors and those initiated by malicious software development attacks.

C[1][1][1]: Disclosure threats to software process during requirements analysis phase caused by innocent errors
C[1][1][2]: Disclosure threats to software process during requirements analysis phase caused by malicious attacks
C[1][2][1]: Disclosure threats to software process during design phase caused by innocent errors
C[1][2][2]: Disclosure threats to software process during design phase caused by malicious attacks

.
.
.

C[3][6][1]: Denial of service threats to software process during maintenance phase caused by innocent errors
C[3][6][2]: Denial of service threats to software process during maintenance phase caused by malicious attacks

Now that we have refined the initial set into a hierarchical structure with three subclass levels, we have identified a total of 36 different leaf nodes that offer useful guidance in identifying potential security vulnerabilities and possible attacks. For example, the C[2][6][2] leaf node (i.e., integrity threats during maintenance by malicious attack) represents the set of security concerns that involve attackers changing or deleting maintained software. Similarly, the C[2][2][1] (i.e., integrity threats during design by innocent error) represents the set of security concerns that involve developers inadvertently changing or deleting design information.

The above hierarchy is often referred to as a threat model for the given environment. As such, it offers a suitable framework for various types of security and risk coverage analyses. For example, one might consider prioritizing the leaf nodes so that different subclasses are viewed as requiring more urgent security attention than others. For this analysis, we choose to provide a baseline prioritization to the subclasses in the second level. That is, we choose to differentiate among the three types of threats represented in the first refinement.

Specifically, we will identify the following priority order for threat types in the second class level:

- *Integrity Threats*: This will be viewed as the highest priority threat for the typical software process since control of change is the salient feature of the typical software development process.
- *Disclosure Threats*: This will be viewed as the second highest priority threat for the typical software process.
- *Denial of Service Threats*: This will be viewed as the lowest priority threat for the typical software process since it involves a potential occurrence that can generally be rectified without serious consequences.

It is important to note that the above prioritization is created with respect to certain assumptions about the typical software process environment. Specific environments with unique attributes (e.g., highly confidential algorithms) might involve different prioritizations. Nevertheless, because our goal in this work is to use the threat model as a basis for integration into a general process standard, we are obliged to make some decisions about what the security priorities will be. Otherwise, all security threats become identified as critical and the proposed protections run the risk of becoming excessive and costly. In cases where the prioritization is different, local decisions must be made on how to tailor available security safeguards to the software process.

4 Threat Coverage Analysis

In order to properly explain and justify the degree to which the twenty clauses included in ISO 9001 cover the 36 threat classes represented in the threat model in Section 3, we must first provide additional detail on the notion of coverage. Certainly, different process requirements will offer different degrees of security risk mitigation for a given threat. One requirement might, for instance, completely remove the potential for a given threat, whereas another might not offer any mitigation at all for that particular threat. This becomes a difficult problem because any number of different degrees of mitigation thus become possible.

In order to make this exercise tractable, we choose to describe a given ISO 9001 process requirement clause as addressing a particular threat in the software process threat model in one of the following ways: (1) Not Applicable, (2) No Mitigation, or (3) Partial Mitigation. This decision follows the observation often made in the security community that no protection ever offers complete mitigation against any security threat. Instead, all protections reduce risk in some manner against select security threats.

Our methodology for determining coverage of the threat model by ISO 9001 clauses follows a simple brute force approach in which we begin with the first threat and continue on to the last, examining the degree to which the full set of clauses mitigates the target threat. Since this is a tedious process to present, we include the results of the analysis in Appendix B. These results are presented in a table in which all threats in the software process threat model are represented and cross-referenced to the twenty requirements clauses in ISO 9001. The resulting areas of noncompliance provide a roadmap for the insertion of new requirements clauses to ISO 9001.

In particular, the following general results were obtained via the threat coverage analysis as direction for our study:

- The threat area best addressed by the ISO 9001 clauses is integrity, which is a welcome result, since integrity is judged to be the most critical concern in a typical software process environment.
- ISO 9001 clauses 15 (Handling storage, packaging and delivery) and 18 (Training) can partially mitigate all of the threats in the model, as would be expected.
- The ISO 9001 clauses provide some measure of security mitigation for all broad areas of security threat. In none of the cases, however, do the

clauses provide sufficient security protection against typical attacks and vulnerabilities.

- ISO clauses tend to be descriptive, rather than prescriptive in their presentation. In addition, the ISO clauses tend to avoid assurance related issues in lieu of more function and feature oriented requirements.
- Heavy reliance for threat mitigation was placed in a few general ISO 9001 clause areas, which introduces risk if the implementation approach for those clauses by a particular organization only satisfies these clauses minimally.

5 Process Security Integration

The manner in which new security process requirements could be offered for organizations interested in both ISO 9001 certification and threat mitigation was considered thoroughly during our study.

One approach considered was to create actual addendums to the existing ISO 9001 clauses in the terminology and style of the existing document. The advantages of this would include ease of integration, should our recommendations meet with successful adoption across the community, as well as ease of understanding by organizations that understood ISO 9001. However, the problem with this approach was that it might mislead readers and users into believing that the proposed additions were for design and development efforts of non-software products (recall that ISO 9001 was not written specifically for software). Since our threat model and analysis were specific to software, this would clearly not be an appropriate perception of our results.

We also considered the potential for incorporating our recommendations into the ISO 9000-3, Part 3 interpretation for software. The advantage to this approach was that it solved the problem of readers and users misinterpreting our results in terms of non-software efforts. A problem, however, arose from the fact that the sections in the interpretation seemed to represent an incomplete framework for integrating our suggestions. Early drafts of such an incorporation required that security be inserted into certain existing sections, and that new sections be added. Ultimately, this approach was rejected in favor of a simpler, more modular solution.

Since the users of the results and recommendations here were likely to be expending significant time, resources, and effort toward their ISO 9001 registrations, we decided to simply describe our security enhancements as a logically separate document, using terms that would be familiar to software specialists. This would minimize the impact to user organizations and would allow for similar types of documents to be constructed for hardware security, system security, physical security, or any other type of security discipline that might be of interest. This integration approach is depicted in Figure 2.

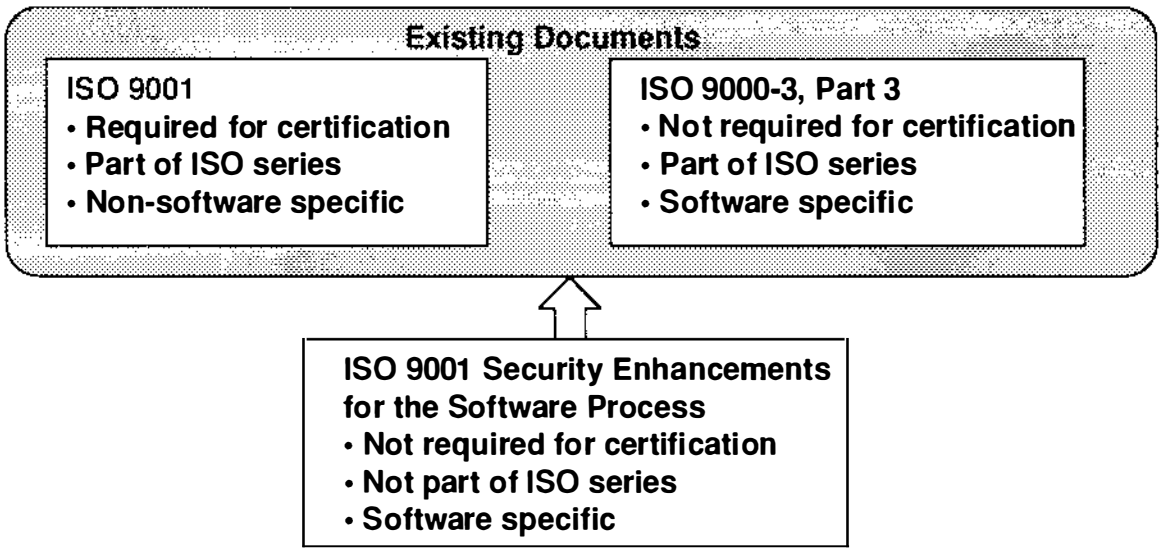


Figure 2. Security Enhancement Integration Approach

6 ISO 9001 Software Security Enhancements

The security enhancements to the software process that are proposed for organizations attempting ISO 9001 certification and registration must be presented in the context of the following important factors:

- They must address the security threats identified earlier to the generic software process. By *address*, we imply that a degree of security risk mitigation (i.e., partial mitigation as defined above) is offered by the proposed enhancement. It is considered beyond the scope of this paper to quantify the precise degree of risk mitigation offered by each security enhancement.
- The proposed security enhancements must be adaptable to the unique needs of a particular software process. This implies that proposed mitigation approaches that rely on specific tools or specific methods are to be avoided since certain processes might not include these as components.
- The proposed security enhancements must not invalidate—in any way whatsoever—the compliance of any software process with the ISO 9001 requirements. This does not imply that the proposed security enhancements should not assist in the ISO 9001 compliance. Instead, it requires that an existing or planned process that is ISO 9001-compliant and that is adjusted to meet the security enhancements proposed here, should not become non-compliant as a result of integrating security.
- The proposed enhancements should be generally-accepted technologies that will not introduce any significant impact to the manner in which particular software development organizations currently do business.

The proposed security enhancements are presented here as a collection of general security technologies that have specific implementations using either adjusted software process approaches or enhancements to the software development environment. The technologies addressed are as follows:

- Authentication
- Access Control
- Auditing and Intrusion Detection
- Key-Managed Encryption Protocols
- Separation of Duty
- User Security Awareness
- Security Administration, Management, and Policy

These technologies are discussed below by providing a brief *description* of the proposed security enhancement, an *identification* of the threat being countered, and a *justification* for the proposal in terms of its impact on a typical ISO 9001 software process. This is followed by a specific statement of the proposed requirement in each area.

It is important to note that the proposed requirements are stated in terms that are more specific than the corresponding ISO 9001 requirements. We followed this approach because more general descriptions would have been less-effective in mitigating the associated security risk. We also recognized that the users of our proposed enhancements are likely to be organizations that desire more detailed guidance in how to mitigate their security risks than a general identification of a technology area.

Authentication

Description. The technology of authentication deals with protecting entry points to computer and network systems via the use of validation processes. In the familiar context of a client/sever architecture, authentication is used to demonstrate to a server that the reported identity of a client is valid. This is accomplished by the use of special knowledge held by the client (e.g., a password), special equipment held by a client (e.g., a smart card), or special attributes of the client (e.g., fingerprint).

Threats. The threats dealt with via authentication are broad. In fact, nearly all vulnerabilities and attack methods associated with every threat identified in Section 3 require access to the target system in some manner. Authentication thus provides a first level of security protection.

Impact. Since most organizations are familiar with the use of passwords for users, implementing user authentication generally does not introduce great problems for software organizations. The most common problems emerge in settings that employ computer or network applications or infrastructure components that do not support such methods. This may imply that new infrastructure elements must be introduced to the software development environment.

Proposed Requirement. "Any software development personnel attempting to gain any type of access to the software development environment should be required to authenticate themselves based on a unique identification token."

Access Control

Description. The technology of access control deals with the use of procedures and mechanisms to restrict access to specified resources and information (e.g., files, directories, network elements) based on a defined security policy. In recent years, operating systems and networks have begun to exhibit access control functionality as a common feature.

Threats. Like authentication, the threats dealt with via access control are broad. Combined with authentication (which provides valid identities on which to base access control decisions) this safeguard provides an important level of security protection.

Impact. The impact of access controls depends to a large extent on the attributes of existing or proposed process environment components. Operating systems or networks without mandatory controls, for example, require considerable enhancement to integrate such controls.

Proposed Requirement. "Any software resources (including documentation and tools) that are considered critical to the software process and its resultant software should be stored in a repository that offers basic access control partitioning protection using methods such as secure network firewalls, access control lists, and commensurate technologies."

Auditing and Intrusion Detection

Description. The technology of auditing consists of on-line mechanisms that allow for capture of information about security-critical events. Typical operating system security-critical events include login attempts and file opens. Typical network security-critical events include message passing or remote login sessions. Intrusion detection consists of techniques for interpreting stored audit data to determine whether an attack has occurred.

Threats. The threats to the software process that are countered by these safeguards include all threats associated with intruders who do not fear reprisal if caught. Such attacks, often referred to as kamikaze attacks, proceed unimpeded in the presence of auditing.

Impact. As with access control, the impact of auditing and intrusion detection depends to a large extent on the attributes of existing or proposed process environment components. Operating systems or networks without auditing, for example, require considerable enhancement to add such functionality.

Proposed Requirement. "The software development environment should be associated with routines and procedures for on-line auditing of software process activity for intrusion detection processing."

Key-Managed Encryption Protocols

Description. This technology offers the capability for remote authentication, disclosure protection, and integrity in message or data transmission. This is accomplished via the use of cryptographic keys (public or private) that offer a means for ensuring that certain conditions hold during transmission. For instance, if the sender and receiver agree on a private key for transmission, then attackers cannot tap the communications media to obtain information.

Threats. This technique is useful for any network disclosure threats, and to a degree, network integrity threats.

Impact. The impact of such integration can be considerable to software organizations that have never considered the use of such protocols before. Certainly, this safeguard is only applicable in environments that include the passing of critical information between locations connected via unprotected communication media.

Proposed Requirement. "The software development environment should include provision for cryptographic key-managed protocols if sensitive, critical information is being transmitted remotely."

Separation of Duty

Description. Separation of duty is a principle that provides guidance for the management of the software process. It is based on the notion that a particular duty in the software process (e.g., software tester, software configuration management) must be staffed with multiple individuals so as to separate the duty.

Threats. This principle reduces the risk of threats associated with single individuals attacking a system or environment using their assigned authority. Thus, threats initiated by disgruntled administrators or developers are mitigated by this type of safeguard.

Impact. The impact of adjusting software processes to comply with separation of duty in certain operations (e.g., system administration) should not introduce great impact to a typical software process environment.

Proposed Requirement. "Job roles and responsibilities for all software process personnel should be allocated in a manner that promotes basic separation of duties in software management, administration, design, development, maintenance, and other activities."

User Security Awareness

Description. User security awareness involves ensuring that any aspect of security for the software process that involves user discretion is dealt with suitably. This is only possible if user discretion is exercised in an educated manner. Hence, security awareness seminars, training, courses, and documentation become important.

Threats. The risk associated with any threat to the software process for which human intervention or operations could alleviate risks is mitigated by enhanced user security awareness.

Impact. The impact of this safeguard in typical ISO 9001 software organizations is likely to be minimal to negligible.

Proposed Requirement. "All software process personnel should be trained in those areas of security for which their discretion plays a critical role (e.g., password selection)."

Security Administration, Management, and Policy

Description. This involves the identification and documentation of suitable security policy rules for administrators and managers to follow.

Threats. The risk associated with any threat for which administration and maintenance could offer mitigation will be affected by this type of safeguard approach.

Impact. The only impact that should be expected for this type of safeguard is that administrators and managers might be forced to follow new types of procedures.

Proposed Requirement. "A specific security policy for the software process should be identified and procedures for enforcing this policy should be integrated into the software process administration and management."

7 Concluding Remarks

We believe that the basic approach taken and results obtained in this study offer confirmation of several issues related to software process quality and security, and the manner in which the two can be integrated. Three of the more critical issues are highlighted below.

- *Software quality controls—in accordance with ISO 9001—can be undermined in practice by the absence of basic security controls.* This result should be taken seriously by organizations who develop software in environments for which security has not been considered. This is especially true in networked environments that may involve remote software development sites. The safeguards recommended provide a first step toward suitable mitigation of the most common security risks.
- *Integration of security controls requires knowledge about the specific attributes of the software process.* This is evident in the many caveats offered in the presentation of security safeguards. For instance, key-managed encryption is only an issue for software processes that involve remote transmission of critical software information. Thus, one should not expect a cookbook approach to work without the corresponding environment-specific analysis.
- *Specification of security safeguards for the software process requires a more prescriptive approach than is demonstrated in ISO 9001.* This may be because the technology of security requires more attention to detail. It may also, however, suggest that the ISO requirements might have been expressed in a less descriptive manner.

References

[1] Weber, C., et al., Key Practices of the Capability Maturity Model, Software Engineering Institute, 1993.

[2] International Standards Organization, *ISO 9001, Quality Systems - Model for Quality Assurance in Design/Development, Production, Installation and Servicing*, 1987.

[3] Neumann, Peter, Risks to the General Public, *ACM Sigsoft Engineering Notes*.

[4] Amoroso, E. et al., Toward an Approach to Measuring Software Trust, *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, Ca., May, 1991.

[5] Department of Defense, *Military Standard for Software Development*, DoD-STD-2167A, 1983.

[6] International Standards Organization, *ISO 9000-3, Quality Management and Quality Assurance standards* - Part 3: *Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software*, 1991.

[7] International Standards Organization, *ISO 9002, Quality Systems - Model for Quality Assurance in Production and Installation*, 1987.

[8] International Standards Organization, *ISO 9003, Quality Systems - Model for Quality Assurance in Final Inspection and Test*, 1987.

[9] Weiss, J., A System Security Engineering Process, Proceedings of the 11th National Computer Security Conference, Baltimore, Md., October, 1991.

[10] Amoroso, E. et al., An Engineering Approach to Secure System Design, Analysis, and Integration, *AT&T Technical Journal*, September/October, 1994.

Appendix A: ISO 9001 Clause Summaries

[Note that the requirements summaries in this appendix are expressed in the authors' words. Exact wording and associated interpretations can be found in the referenced ISO documents.]

Clause 1: Management Responsibility. This requirement clause is grouped into three sections: Quality Policy, Organization, and Management Review. The Quality Policy section requires that the supplier's management define and document its quality policy. The Organization section lists requirements for responsibility and authority, verification resources and personnel, and management representation as it relates to the Quality Policy. The Management Review section requires periodic reviews of the appropriateness of the Quality Policy.

Clause 2: Quality System. This area requires the establishment of a quality system including documented procedures and instructions, and effective implementation of the quality system.

Clause 3: Contract Review. This area requires contract review to ensure that requirements are defined and documented, requirement differences are resolved, and the supplier is capable of meeting the requirements.

Clause 4: Design Control. This area requires the establishment of general procedures, design and development planning, design input, design output, design verification, and design changes.

Clause 5: Document Control. This area requires procedures to control documents and requirements data including documentation availability and the removal of obsolete documents.

Clause 6: Purchasing. This area requires assessment of sub-contractors, purchasing data documentation, and the verification of the purchased product with respect to specified requirements.

Clause 7: Purchaser-Supplied Product. This area requires the establishment of procedures for the verification, storage, and maintenance of products and tools supplied by the customer.

Clause 8: Product Identification and Traceability. This area requires the establishment of procedures for identifying the product from available documents and the application of a unique identifier when traceability is specified.

Clause 9: Process Control. This area requires: instructions for production and installation, monitoring and control of process/product during production and installation, approval of process/product, criteria for workmanship, and monitoring of the product in use when subsequent inspection is inappropriate.

Clause 10: Inspection and Testing. This area requires receiving inspection and testing, in-process inspection and testing, final inspection and testing, and inspection and test records.

Clause 11: Inspection, Measuring and Test Equipment This area requires the supplier to identify the measurements to be made, identify, calibrate and adjust all equipment, document applicable procedures, ensure accuracy and precision of equipment, identify inspection to show the calibration status, maintain equipment calibration records, assess previous inspection when equipment is found to be out of calibration, ensure suitable environmental calibration conditions, ensure handling of equipment to maintain accuracy, and safeguard inspection, measuring and test facilities to maintain a valid calibration setting.

Clause 12: Inspection and Test Status. This area requires the inspection and test status of a product be physically identified and be maintained through production and installation.

Clause 13: Control of Nonconforming Product. This area requires the establishment of procedures concerning non-conforming products including responsibility for review and authority for the disposition of the product.

Clause 14: Corrective Action. This area requires the establishment of procedures for investigating the cause of nonconforming product, analyzing all processes to detect and eliminate nonconforming product, initiating preventative action, applying controls to implement corrective action, and implementing and recording procedural changes.

Clause 15: Handling, Storage, Packaging, and Delivery. This requires the establishment of procedures: to prevent damage of the product through improper handling and storage, to control packaging to ensure conformance, and to protect the quality of the product through delivery.

Clause 16: Quality Records. This area requires the establishment of procedures to maintain quality records that demonstrate the achievement of the quality system.

Clause 17: Internal Quality Audits This area requires the supplier to audit the effectiveness of the quality system on a scheduled basis including follow-up actions and corrective actions on any deficiencies found.

Clause 18: Training. This area requires the establishment of procedures to identify training needs, provide necessary training, select qualified trainers and maintain training records.

Clause 19: Servicing. This area requires the establishment of procedures for performing and verifying that servicing meets requirements.

Clause 20: Statistical Techniques. This area requires the establishment of procedures for identifying statistical techniques for verifying the acceptability of process capability and product characteristics.

Appendix B: Security Coverage Analysis

Threat	ISO 9001 Clause																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
C(1)(1)(1)			P		P										P			P		
C(1)(1)(2)			P		P										P			P		
C(1)(2)(1)			P	P	P										P			P		
C(1)(2)(2)			P	P	P										P			P		
C(1)(3)(1)			P	P	P										P			P		
C(1)(3)(2)			P	P	P										P			P		
C(1)(4)(1)			P		P										P			P		
C(1)(4)(2)			P		P										P			P		
C(1)(5)(1)			P		P					P					P			P		
C(1)(5)(2)			P		P					P					P			P		
C(1)(6)(1)			P		P										P			P	P	
C(1)(6)(2)			P		P										P			P	P	
C(2)(1)(1)	P	P	P		P	P	P								P			P		P
C(2)(1)(2)	P	P	P		P	P	P								P			P		P
C(2)(2)(1)	P	P	P	P	P	P	P								P			P		P
C(2)(2)(2)	P	P	P	P	P	P	P								P			P		P
C(2)(3)(1)	P	P	P	P	P	P	P			P	P				P			P		P
C(2)(3)(2)	P	P	P	P	P	P	P			P	P				P			P		P
C(2)(4)(1)	P	P	P		P	P	P			P	P				P			P		P
C(2)(4)(2)	P	P	P		P	P	P			P	P				P			P		P
C(2)(5)(1)	P	P	P		P	P	P			P	P				P			P		P
C(2)(5)(2)	P	P	P		P	P	P			P	P				P			P		P
C(2)(6)(1)	P	P	P		P	P	P								P			P	P	P
C(2)(6)(2)	P	P	P		P	P	P								P			P	P	P
C(3)(1)(1)															P			P		
C(3)(1)(2)															P			P		
C(3)(2)(1)															P			P		
C(3)(2)(2)															P			P		
C(3)(3)(1)															P			P		
C(3)(3)(2)															P			P		
C(3)(4)(1)															P			P		
C(3)(4)(2)															P			P		
C(3)(5)(1)															P			P		
C(3)(5)(2)															P			P		
C(3)(6)(1)															P			P		
C(3)(6)(2)															P			P		

P	Partial Mitigation
Blank	No Mitigation
Shaded	Not Applicable

Faith and Hope: Methodologies for Building Trusted Systems

John McHugh
Associate Professor / Tektronix Professorship
Computer Science Department
Portland State University

Premise

While much of the software that we build is "pretty good", we lack the link between process and product that would permit us to predict accurately the quality of the resulting product.

As a result:

- We cannot give prescriptive advice to developers.
- We cannot provide meaningful warranties
- We cannot predict the safety or reliability of systems that make extensive and critical use of software

Is this really the case?

I believe that it is so.

I invite you, *implore you*, to convince me otherwise.

Let us consider some cases that matter.

- Flight control systems
- Medical applications
- Safety critical nuclear powerplant controls
- Weapons systems

The evidence is disheartening

Flight Control Systems

As a frequent flyer, I'm concerned.

John Rushby has some wonderful case studies from an experimental military aircraft.

There is a lingering suspicion that most of the A320 crashes have had software involvement.

I have collected enough anecdotes from pilots and flight crews to be leery of full authority systems.

- Piedmont localizer
- UPS MD-11 fuel system
- F-100 Glass cockpit

Medical Applications

The Therac-25 is probably the best known example. It killed a number of people due to software errors.

My medical colleagues all have their own stories. These are frequent enough to make me uneasy.

Devices are not the only risk. We have seen instances of deliberate tampering with medical records, putting patients at risk.

Physicians are far from perfect, but the trend towards relying on automation as a cost control mechanism will probably not improve the quality of care.

Nuclear Power Plants

The traditional nuclear reactor control system separates operational from safety controls. This allows a very conservative safety approach, but it leads to an excessive number of false alarms as operators try to squeeze out extra kwh.

This is leading towards increasingly complex safety systems and growing doubts about their safety.

Faith and Hope

Weapons Systems

We have been lucky with the bombs but not so with the decision making and control systems.

There are well documented software faults in the Patriot. Aegis design flaws led to the destruction of a civilian airliner with the loss of all on board.

Most large weapons systems today have a substantial software component. Most of the procurements attribute a large part of their cost overruns and delivery slips to software problems.

Faith and Hope

Is This Necessary?

We have seen similar phenomena in other areas. The histories of Navel architecture, bridge building, and steam engineering are replete with similar examples.

As we move into new areas of development, we slowly learn that prior techniques are fraught with pitfalls.

In early endeavors, the field is full of charlatans. These are brought under control, by societal pressure, either through market pressures, internal governance of the profession, or government regulation.

Some Nostrums

Lots of things have been suggested.
Some may or may not work. We don't know.

- Structured programming
- Formal Methods
- TDM
- etc.

Some we know don't work.

- N- Modular redundancy

Is This Necessary?

AS LONG AS WE DON'T KNOW WHAT WE ARE DOING, A CERTAIN NUMBER OF ACCIDENTS ARE INEVITABLE!

Failures can be a driving force for research and discovery. Shortly after we learned about centers of mass, we stopped designing inherently unstable ships. Failures in cast iron bridges and steam boilers led to fundamental work in materials science and structural engineering.

The current situation in software engineering has

- Similarities and
- Differences

Similarities

On the whole, we do pretty well.

- **The average individual is not often affected by software failures.**
- **Failures that affect lots of people are widely publicized, sometimes.**
- **We are in a state of denial about the inevitability of the problems, though we seldom attribute them to a deity.**

Differences

- **Software is much more malleable**
- **We have many more small failures and are willing to tolerate them more often.**
- **We tend to anthropomorphise many failures and avoid looking for the real causes.**
- **We have not yet reached the threshold at which public pressure forces a change.**
  - **We may never. Commonalty is not recognized.**
  - **We aren't ready by a long shot.**

Three Problems

- Collectively and individually, we fail to learn from our past mistakes.
- We rush to impose standards without any evidence that the standard will improve the state of the practice.
- We don't understand what goes wrong.

  This applies equally to product, process, and to the relationship between product and product.

History

Those who cannot remember the past are condemned to repeat it.

Santayana

Other engineering disciplines have overcome failures by collecting failure data and analyzing failures for commonalty that could lead to avoidance of that kind of failure in the future.

Failure data in software is generally considered proprietary.

With few exceptions, failure data from product developments is not available for open research

Failure Data

At an NRC workshop last year, Win Royce of TRW urged the sharing of failure data. When pressed, he admitted that he was not in a position to offer TRW data.

This is typical. Very little has appeared on attempts to use detailed failure data for product and process improvement. IBM has a program for some product lines.

In regulated areas, the availability of product and process failure data for public scrutiny should be a matter of policy, possibly in trade for reduced liability exposure.

Standards

There is a rush towards premature standardization. MoD 0055 and 0056 are perhaps examples, though they were intended as driving forces.

Standards such as MilStd 2167A, DO 178, etc. tend to be interpreted in a prescriptive fashion, even though there is no evidence that the prescribed activities are either effective or cost effective in meeting the goals of the standard.

Traditional standards are codifications of long accepted practices. Software engineering standards tend to be more arbitrary.

Standards Agenda

Standards that are not codifications of practices that have proven effective should have a validation component.

This would require the collection of data that, used as case studies, would validate or refute the assumptions on which the standard was based.

Where do errors come from?

The process of building software is a human activity.

The first generation of computer programmers came primarily from the physical sciences, engineering, and math.

This kind of background is usually ill suited to careful consideration of the human factor. The social scientists who have come into computer science have generally looked at how people use computers, not how computers and programs are developed.

We can only speculate about the latter.

Speculation

1) Poor communications.
   Customers and users don't speak the same language.
2) Overwhelming complexity.
   Programs offer complexity at many levels. Details get lost and resurface.
3) Incompetence and over confidence.
   Many programmers are amateurs and don't understand what they are up against.
4) Individual differences
   Why do different people write different programs from the same specifications? Why do they get it wrong differently but in the same place?

What can we do?

It is tempting to say, "I've identified the problem, the solution is up to you."

After all, you are the folks in the trenches who have to live with it. But that wouldn't be right.

- It may be that there is little or nothing we can do. What if the development process is chaotic?
- If we are going to solve the problems, we need information. This means systematically collecting and sharing failure data.
- Given enough data, we may be able to figure out what goes wrong and perhaps devise ways to fix or prevent the problems.

COMPONENTS OF AN EFFECTIVE PROCESS IMPROVEMENT PROGRAM

ABSTRACT

This paper describes a workshop that addresses many of the key components of successful process improvement programs. This workshop is highly tailorable; it may be presented as a single day overview to individuals and groups, or it may be used in a multi-day, "teach a little, work a little" format with intact work groups and teams. This paper covers:

1. BACKGROUND AND MOTIVATION - why this workshop was developed
2. PRACTICE WHAT WE PREACH; WALK OUR TALK - how this workshop is presented
3. WORKSHOP TOPICS - what is presented in one tailoring of this workshop
4. SUMMARY OF OUR EXPERIENCE - what the reaction of workshop participants was
5. BIBLIOGRAPHY - where we derived our information
6. ACKNOWLEDGEMENTS - who were the initial reviewers

BIOGRAPHY: JUDY BAMBERGER

Judy Bamberger is a Senior Quality Assurance Software Engineer at Sequent Computer Systems Inc. She is currently coordinating the software quality improvement activities across the company and is leading the effort for ISO 9001 registration for the software development groups under the TickIt scheme. In addition, she is working with local government and higher education to improve the capabilities of software engineers who pass through universities, colleges, and continuing education programs.

Previously, Ms Bamberger was a Senior Software Engineering Specialist at Loral Western Development Labs and a member of the Software Engineering Process Group. In this role, Ms Bamberger developed a "Health & Status Check" method to help projects focus on specific areas requiring improvement, and worked with the training and metrics programs. She developed and offered training in leadership skills, communication, and managing change.

Prior to joining Loral, Ms Bamberger was a Member of the Technical Staff at the Software Engineering Institute (SEI). She was a member of the Software Process Assessment Project, where she co-authored the update of the Capability Maturity Model for Software and contributed to the assessment method and teaching materials. She also was project leader for the Distributed Ada Real-Time Kernel (DARK) Transition Project and a member of the DARK development team.

Prior to joining the SEI, Ms Bamberger spent seven years at TRW Defense Systems Group, where she was a Section Head and Member of the Technical Staff. In those positions, she provided expertise to several proposal efforts in the areas of databases, Ada, software engineering and design, and she was an activity leader working in databases and their use in C3I systems and secure communication systems.

Judy Bamberger received her BS in Mathematics, French, and Education from the University of Wisconsin - Milwaukee in 1974, her MEd from the University of Northern Colorado in 1979, and her MS in Computer Science from UCLA in 1985. She has authored several papers and has been invited to give numerous presentations and workshops on the Capability Maturity Model, Managing Change, Process Improvement, Software Process Assessment, and Ada, nationally and internationally

Judy Bamberger served two terms as Chairperson of the Ada-Joint Users Group (AdaJUG) and was an active member of ACM SIGAda, most recently serving as Conference Chairperson for TRI-Ada '91. She is actively involved in the Pacific Northwest Software Quality Conference organization.

Ms Bamberger is authorized by the SEI as a Lead Assessor in the new, CMM-Based Appraisal method for Internal Process Improvement.

Judy Bamberger	Voice: +1-503-578-3028
Senior Quality Assurance Software Engineering	Fax: +1-503-578-7562
Sequent Computer Systems Inc	Internet: bamberg@sequent.com
15450 SW Koll Parkway COL2-860	
Beaverton OR 97006-6063	

1. BACKGROUND AND MOTIVATION

Process improvement is messy; it takes time; it takes patience; it takes creativity and perseverance. And guess what - there is no one right way!

This simple yet powerful statement is something that comes to most of us as an "aha!" during our careers as process improvers. In fact, when I joined my first formal process improvement activity, I was told - or should I say "warned!" - that "it will change your life." It did; it continues to do so. And in talking with my colleagues and gleaning information from readings and courses, that is the general conclusion to which we have come, together and as individuals.

As a result, when it was time for me to create a "process improvement workshop," I went back to those same colleagues and references to identify the toughest issues that need to be addressed. Understanding the Malcolm Baldridge criteria, interpreting ISO 9000 for software, applying the Capability Maturity Model for Software, etc - these were *not* among the tough issues. The areas that the process improvers found the most intractable and insurmountable were "the people issues." Individual strengths and weaknesses, effective team behaviors, obtaining and maintaining sponsorship, conflict management, problem solving, ... - these proved to be the toughest issues to address before and throughout process improvement activities.

The key to effective and continuing process improvement, as echoed clearly by Kearns and Nadler in, *Xerox: Prophets in the Dark - How Xerox Reinvented Itself and Beat Back the Japanese*, is recognizing that a *culture change* is needed. Words of wisdom from the quality gurus (e.g., Deming, Crosby, etc), the quality tools (e.g., quality maturity grids, statistical process control) - none of this in and of itself will ensure that the process changes are institutionalized. What must be changed, in a positive manner, is the *culture* within which the work occurs.

A culture involves people, and the degree to which the people issues are addressed effectively is linked tightly to the degree of successful improvement activities. Improvement, by its very nature, implies something must change, and the people who are involved with that something-that-is-changing - they, too, must change - how they do their jobs, with whom they work, when the task is done, or the whole task itself. So in talking about effective process improvement, we need to recognize that we are talking about people, ultimately, and changing the way people work.

This tailorable workshop focuses on the people issues that appear to be key contributors to successful process improvement programs. The workshop introduces several concepts and skills that are applicable to process improvement in any domain - software development, procurement, business management, etc.

Specific behavioral objectives to be achieved are defined for each workshop section; more in-depth readings are provided as "homework" - to introduce and reinforce the concepts and skills, to foster discussion among workshop participants, and to demonstrate the relationship between experiences in other domains to those of the participants. A unified case study is woven throughout the workshop, and time is provided for participants to apply the newly introduced techniques to their own organizations and domains. Following effective adult learning paradigms, the workshop uses a highly participatory approach in teaching these process improvement techniques.

If you are expecting "here is how you get to Level 2," this paper, as well as the workshop from which it is derived, is not for you; if your focus is on leveraging the strengths of individuals and teams, on problem solving and conflict management techniques, and on strategies for building sponsorship and leadership, keep reading!

This paper summarizes the specific topics covered in the workshop, and how we teach them.

This workshop is intended primarily for those who are leading process improvement efforts or participating as members of a process improvement team, and the paper reflects that focus.

2. PRACTICE WHAT WE PREACH; WALK OUR TALK

The workshop itself is structured to demonstrate the behaviors we are asking from those who lead and participate in process improvement activities. In-workshop exercises require team interaction - understanding and exploiting individual strengths and weaknesses and team dynamics are one key to success. We teach these; we try them out; and we discuss them in isolation ... and they are woven throughout the exercises and reviewed during session post mortems.

Effective meeting management techniques are used throughout team exercises. There are clear goals to achieve, clear time limits within which to accomplish the tasks, and clear roles and responsibilities needed to complete the exercises effectively. We teach meeting management; workshop participants are encouraged to practice it; and we review the effectiveness at the end of each session.

We teach brainstorming, consensus, problem solving techniques and more ... we provide exercises to reinforce the basic concepts and skills; and we use them throughout the workshop and application of the techniques to real-world issues.

3. WORKSHOP TOPICS

By design, the process improvement workshop may be tailored to the specific needs of the group participating in it. When given to intact work groups, we use workplace examples in place of the case study, or, if the case study is the most effective way to demonstrate cleanly some foundational concept or skill, we follow the case study exercises with in-workshop time to focus on the real problems facing the teams. Table 1 is the high-level, general outline for the process improvement workshop, including the section number in this paper where the workshop topic is described further.

Table 1. General Outline for Process Improvement Workshop.

- Introductions (section 3.1)
- Introducing the Big Picture (section 3.2)
  - Management Support
    - Barriers Facing Organizations
      - Brainstorming
      - Consensus
      - *NEAT* (Nature, Expectations, Agenda, Time)
    - Organizations as Systems
    - Management and Stakeholder Support
      - Management Sponsorship Planning
    - Process Improvement Models
  - Process Improvement Framework
- Starting Point - Focus on Self (section 3.3)
  - A Personality Model
    - Strategizing by Exploiting Preferences
  - A Communication Model
    - Practicing Skills for Effective Communications
- Inter-dependency - Focus on Team (section 3.4)
  - A Team Growth Model
    - Preferences and Team Growth
  - A Problem Solving Model
    - Useful Tools at Each Phase
    - Decision Making Styles and Tools
  - A Conflict Resolution Model
    - Identifying, Managing, and Resolving Conflict
- Revisiting the Big Picture (section 3.5)
  - Managing Technological Change
  - Close and Evaluation

3.1 INTRODUCTIONS

A question always arises - why bother introducing workshop participants to each other, especially if they work together already? Introductions play an important part for setting context and expectations - how participants are going to work with each other, the role of the workshop facilitator, what each of the participants brings into the workshop and expects out of it. We have found that no matter how well people know each other before coming into the workshop, there is always something new they can learn about other participants - especially their teammates - that they can exploit to everybody's benefit both in the workshop and back in the work environment.

In team growth terms, the time spent in this section focuses the teams on "forming" - orienting themselves to the task and to the relationships with each other, and the facilitator does most of the directing as the team begins to work together.

3.2 INTRODUCING THE BIG PICTURE

The objectives of this section are for workshop participants to:

- *Learn and use three fundamental techniques.*

  The basic techniques we teach, review, and use here and throughout the remainder of this workshop are: brainstorming, consensus, and meeting management (*NEAT* [described a bit further in a following subsection] and roles and responsibilities). While many of us have learned these processes formally at one time or another, periodic refreshers are always helpful.

- *Discuss key barriers facing their organizations.*

  We use the tools described just above to elicit information from the participants - about the barriers to process improvement that they have encountered in the workplace. This information is discussed here and used repeatedly throughout the workshop. Barriers encountered by other organizations are shared ... and regardless of the varying maturity or formality of the organizational processes. the teams come to consensus fairly rapidly on the key barriers.

- *Begin to gain insight into organizations as systems, management sponsorship issues, and why process improvement is "messy."*

  Through these discussions, the team members identify similarities and common views among them and among teams that are sometimes scattered across different organizational entities. We discuss organizations as systems - examining how changes in one part of the organization affect many other parts, and discussing the importance of shared responsibility in process improvement activities (too many efforts have failed because the organization adopted the attitude that "it's just software's problem: they have to fix themselves once and for all!"). We review the important role of management in effective process improvement, using references from Watts Humphrey, the Process Enhancement Partnership, and the Juran Institute.

- *Use some tools to begin structuring improvement efforts.*

  like any project would be. Two tools are introduced at this point to help process improvement teams structure their efforts and treat them like real projects.

  The first is an organizational climate appraisal tool. It is modeled on the dimensions of the organizational climate model described in articles by Forehand and Gilmer, and by Pareek. This information is used to help improvement teams identify cultural strengths to leverage and barriers to address.

  The second tool is a sponsorship planning tool. Since building and maintaining effective sponsorship is key to successful process improvement programs, and since it is the area that has the most potential to enable or destroy process improvement success, we teach a structured approach that is based on a model of commitment over time and effective communication techniques, among others. The outcome of these exercises are to provide workshop participants with a structured list of organizational

strengths and barriers, for reference in later steps of process improvement, and a prioritized set of contact and commitment plans for sponsors.

Both tools show workshop participants that even some of the most "people-oriented" portions of process improvement can be structured in a way to enhance management of the process improvement activities, thus allowing process improvers to maximize effective creativity in areas that require creative analysis and activities.

- *Plant seeds of appreciation for why process improvement must be continuous.*

  This section of the workshop introduces four process improvement models, including the basic "plan, do, check, act" (PDCA) model. Each of the other three - FADE from Organizational Dynamics Inc, the process improvement model from the Institute for Software Process Improvement, and the model from Being First - has components of the PDCA cycle, and each has been extended in different ways to emphasize key points, techniques, or activities. Through discussion, workshop participants discuss why all the issues they have identified early on cannot be solved in just one cycle through any of these models.

In team growth terms, the time spent in this section focuses the teams on "forming" and "storming" - resistance to the task at hand, trying to shape it in their own image, and hostility in relationships. During this part of the workshop, the workshop participants, in teams, are faced with some problem solving activities, and the facilitator is operating more in a coaching role than a directing role. During this phase, the team members jostle for leadership and legitimacy, and the opportunity to contribute and be recognized for it.

3.2.1 *NEAT* (Nature, Expectations, Agenda, Time)

A "neat" and elegant meeting management technique is woven throughout the workshop, and is used to structure the exercises. The technique, called *NEAT*, is useful for structuring a meeting beforehand - e.g., in the meeting announcement, or for recouping some degree of focus if/when a meeting gets out of hand. Table 2 summarizes the *NEAT* technique.

Table 2. *NEAT*.

Nature	Purpose of the meeting; *"why are we here?"*
Expectations	Outcome from meeting; *"what is our product. our goal?"*
Agenda	Issues to discuss, actions to take, decisions to make; *"how are we getting there?"*
Time	For each agenda item and/or for adjournment; *"when is all this happening?"*

Along with *NEAT*, we present the importance of assigning specific roles and responsibilities - e.g., recorder, timekeeper, moderator, presenter - and again use and reinforce this concept throughout the workshop. We have observed that both techniques are used frequently, correctly, and expeditiously in the workplace.

3.3 STARTING POINT - FOCUS ON SELF

In addition to reinforcing and building on skills and concepts introduced in the previous section, the objectives of this section are for workshop participants to:

- *See different preferences for how people gather information and make decisions.*

  This section is based largely on Carl Jung's psychological preferences. (If time allows, we explore other models as well.) Some readers may have been exposed to these through the Myers-Briggs Type Indicator (MBTI) and/or by reading about it in books such as *Please Understand Me - Character & Temperament Types*, by Keirsey and Bates. Jung, and work that followed him, demonstrated that people have natural preferences along four dimensions: energizing, attending, deciding, and living. People tend to have aspects of all characteristics, yet people prefer to use those that appear most natural and comfortable. The workshop explores these in some detail, and the participants are led through a series of exercises to identify their relative strengths.

- *Discuss how these preferences both aid and confound process improvement activities.*

  The participants, again through a series of team exercises, discover how their preferences help them exploit their strengths most effectively in process improvement activities, and how they can most effectively leverage the strengths of others when appropriate. Using the information from the sponsorship planning exercises, the participants integrate the personality preferences concepts to hone the messages and methods of communication into contacts that have increased probability of successful and accurate message delivery and receipt.

- *Discuss and practice components of effective communication.*

  This component presents a general model of communication, and leads the workshop participants through a structured set of exercises to explore effective communication techniques. This is often not the first time many of the participants have been exposed to these skills; that the skills are clearly defined, build on each other, and practiced many times makes this module effective. Once again, these skills are tied back into work done so far by role playing and discussion.

In team growth terms, the time spent in this section focuses the teams on "storming" and "norming" - building effective communication about the task and establishing cohesion among the team members. During this part of the workshop, the workshop participants, in teams, are faced with continuing problem solving activities, and the facilitator is operating more in a coaching and supporting role. During this phase, the team members begin to focus more on the task at hand and the relationships they need to build to address the problems ... and less on personalities and hidden agendas.

3.4 INTER-DEPENDENCY - FOCUS ON TEAM

In addition to reinforcing and building on skills and concepts introduced in the previous sections, the objectives of this section are for workshop participants to:

- *Discuss, formally, the characteristics of effective teams.*

  During this portion of the workshop, the five-stage team growth model (forming, storming, norming, performing, and adjourning) is presented and discussed. Characteristics of high-performing teams and effective team leadership are also discussed. Each of the teams looks at where it is on a grid relating the task behaviors to the relationship behaviors, and focuses on how to move forward - through "storming" and "norming" and into "performing." The importance of characteristics of effective teams - and the need to revisit and discuss these ideas in the workplace on a regular basis - is discussed.

- *Identify strengths and weaknesses that different personality preferences contribute to each phase of team growth.*

  Each individual brings strengths and weaknesses at any phase of team growth. Using personality preferences as one basis for discussion, the team focuses on the strengths each preference brings to effective team operation at each stage of growth. The barriers are identified and discussed as well. This is tied back to the sponsorship messages and lists of barriers to begin understanding how the strengths of each preference can be synergized as a team in effective communication, strategic planning, and execution of process improvement programs.

- *Examine and usea process for problem solving and decision making.*

  By this point in the workshop, the participants have been doing some problem solving, coming to consensus, making some decisions. Within the framework of work so far, the team is introduced to a structured problem solving process. There are many; they all have the same basic characteristics, which are not dissimilar to the PDCA process improvement model introduced earlier. Each step of a generic problem solving model is reviewed - what is accomplished in each step, what tools are often effective in each step, what roles different people can play during each step. When this module is used with intact work groups, the teams focus on specific problems to be solved, identify specific tools to use and receive training on them as needed, and then execute the problem solving process with real world problems and information. We review the effectiveness of the problem solving process as part of team adjourning.

  Making decisions is a part of solving problems. There are times when the decision may be delegated - "you decide"; when consensus is appropriate - "let's decide"; when a more consultative style is necessary - "let's discuss and I'll recommend a solution"; or when an autocratic style is needed - "I decide." There is no one right style for all contexts; decision making is truly situational. The workshop explores the benefits and problems associated with each. When working with intact work groups, the teams identify which decision-making style is appropriate for the problem at hand, and they document it as part of the team groundrules. The relationship between personality

preferences and the key contributions each makes to effective decision making and problem solving is demonstrated.

- *Practice conflict management techniques.*

  When we begin the discussion of conflict, many participants are uncomfortable, as they have been taught carefully that conflict in and of itself is "bad." We take the opposite point of view - that conflict is to be expected when solving problems, and the only "bad" thing about it is when it is not managed and resolved. Often, conflicts appear as manifestations of different personality preferences - this is illustrated and discussed. An exercise is provided to help make the content of conflict visible - examining the facts, methods, goals, and values from the different points of view of those in conflict. This paradigm has proven effective is focusing the conflict on the issues, and away from the persons and personalities. Five different styles of conflict management are described - based on the cooperativeness/assertiveness scales described by Thomas-Klimann. Each style: avoiding, accommodating, competing, collaborating, and compromising, is appropriate to use in certain situations ... and is very detrimental to effective conflict resolution if used inappropriately, as an exercise involving each style demonstrates. The workshop explores conflict resolution further with a deceptively simple exercise that can be solved quickly if the content of the conflict is identified clearly.

In team growth terms - the time spent in this section focuses the teams increasingly less on "storming" and more on "norming" - building effective communication about the task and establishing cohesion among the team members. When this workshop is offered to intact work groups, some teams progress to "performing" - focusing on problem solving for the task at hand, exploiting the team interdependence and shared strengths. This takes time: teams do not reach this stage over night, even if they have been working together and are now solving a new problem. Teams do not remain at this stage automatically or without attention to basic team needs on a regular basis. The workshop is created so that the facilitator structures the sessions, activities, discussion, and homework to expedite moving the team forward as a team, modeling effective team behaviors so that the teams can become self-sufficient and self-tending after the workshop.

3.5 REVISITING THE BIG PICTURE

This section of the workshop provides some time for the participants to weave together earlier themes, ideas, tools, and methods. The objectives of this section are for workshop participants to:

- *See and discuss typical responses to change.*

  Tying up the opening themes - process improvement implies change. Recognizing this and addressing it effectively is key to the successful implementation of any improvement. The workshop examines the challenges of dealing with change using William Bridges' *Managing Transitions - Making the Most of Change* as the basis of exploration. Using simple and fun exercises, the basic barriers to change are demonstrated and discussed.

- *Demonstrate some challenges faced within the workplace.*

  The teams apply that information to identifying and discussing the barriers facing them in the workplace - issues that have been identified and accumulated throughout the workshop, and begin to formulate action plans to address those barriers.

- *Generate ideas and strategies to position workshop participants for highly effective process improvement.*

  Based on the shared knowledge of workshop participants, exercises, discussions, readings, and in-workshop activities, characteristics of successful process improvement programs are identified.

In team growth terms, the time spent in this section focuses the teams on both "norming" and "performing." At this point in the workshop, the participants are more involved in review and synthesis than in problem solving and decision making. This "relaxation" period at the end of the workshop provides participants with an opportunity to reflect, internalize, and share what they have learned.

4. SUMMARY OF OUR EXPERIENCE

This workshop was designed to meet the needs of those leading and participating in process improvement programs. I scavenged the best references and examples, listened to those front-line pioneers of process improvement in their organizations, talked and worked with those involved in process improvement activities, and identified and created tools and techniques to address the key problem areas. The workshop components are focused on the basic concepts and skills, and specific applications of tools and techniques are provided when the need arises - "just in time" - to enable solving specific problems when appropriate in the life of the process improvement projects themselves.

The workshop participants themselves unanimously have rated the preparation provided by the workshop as "very good" (4) to "excellent" (5). For example, from a typical offering:

- 4.36 for "The course content is relevant to my job assignment."
- 4.55 for "The course was structured and organized to facilitate learning."
- 4.64 for "The materials were accurate and easy to understand."
- 4.36 for "The labs/exercises were appropriate for the course."

Specific written comments on the workshop evaluation forms included:

- "I wish it had been longer."
- "Excellent; it was very worthwhile!"
- "It was good to practice or 'see' the ideas that are talked about."
- "I've just started as an SEPG member. I haven't done anything yet, but I think I have some ideas how I can be effective as a member of the team."

In addition, the participants have been able to take the raw workshop materials and use them in their own work environments, extending them and tailoring them to meet their specific needs. I have not done any follow up to determine long-term effectiveness, nor have I done any empirical, before-and-after studies.

The process improvement workshop was designed originally to fit in the context of an aerospace contractor, Loral Western Development Labs. As a testament to its sound, conceptual basis, key components have been reused, intact, within the commercial environment at Sequent Computer Systems Inc, and have been presented at conferences at other business locations. One initial goal of this workshop was to ensure its usability across domains, across companies, across projects. This goal has been demonstrated successfully. In addition, many of the process improvement tools and techniques already in use at Sequent are, not surprisingly, not dissimilar to those used in the workshop - and in many cases, our sources for the raw information have been the same.

The workshop "travels well," in its entirety or as separate modules. It has been presented in pieces at various Loral offices, at conferences, and with smaller intact work groups

5. BIBLIOGRAPHY

An extended list of references and some of the author's "favorite things" appears in the slide package.

6. ACKNOWLEDGEMENTS

The work on which this paper is based was originally created at Loral Western Development Labs. My thanks to them for their intelligent and ruthless review of the workshop materials; the improvement from the original to the current content and form is tremendous - Patty Ehlers, Dave Gobuty, Karen Humber, Karl Kelley, Joe Leporati, Steve Lazarus, Gayle Towers. My thanks to Mike Forster of ASK; and also to Neal Brenner, then of SunSoft, now of Oracle, for his intellectually-stimulating comments, debates, discussions, and insights before, during, and after dry run hours. My gratitude also to my new colleagues at Sequent Computer Systems Inc for their capable and insightful review of this paper, and for providing me the opportunity to practice these skills in a new environment. And to Steve and Neal, who cheerfully admit that "I have ruined their lives with this OD stuff" - thank you for your support through all the challenges and opportunities.

SOFTWARE QUALITY IN 1994:
WHAT WORKS AND WHAT DOESN'T?

Draft 1: March 17, 1993
Draft 4: June 30, 1994

Capers Jones, Chairman
Software Productivity Research, Inc.
1 New England Executive Park
Burlington, MA 01803

Phone	617 273-0140
FAX	617 273-5176
EMail	capers@spr.com
Compuserve	75430,231

Abstract

In late 1992, Software Productivity Research (SPR) was commissioned to explore aspects of software quality. This study was continued in 1993 and 1994. Both traditional software methods such as formal inspections, and modern software methods such as ISO certification, were investigated. Some of the results were encouraging. The production of very high-quality software is both technically achievable and becoming more common. Other results were less optimistic. The gap between quality leaders and quality laggards is still extremely wide, and those who are far behind may even be moving in the wrong direction. An important topic with essentially zero available information is that of the quality levels of data used by software applications. There is also a shortage of good empirical data on the quality levels of client-server software and object-oriented software.

Introduction

Software quality is now a major issue for the success of companies, industries, and even countries. Software itself has become one of the most pervasive technologies of the 20th century. Within the past half century, software has spread from a small number of comparatively specialized applications to become a critical factor in almost all engineered products.

Software has also become a major factor in consumer goods, and in corporate operations, military strategy, tactics, and weapons systems, and in government operations.

Twenty to thirty years ago, poor software quality was often annoying, but today poor software quality can literally shut down a phone system, a weapons system, and even a company. Any reasonable prognosis makes software even more critical in the future, and hence software quality will become more critical than today as well.

In late 1992 Software Productivity Research was commissioned to explore several different aspects of software quality internationally. This survey has become a continuing practice, and was also carried out during 1993 and 1994. The total volume of data on software quality now available is much too large for a journal article to cover it completely. Following are short excerpts from the major findings, listed in alphabetic order:

Baldrige Awards

As a class, Baldrige Award winners are much better in a number of quality control respects than most companies. It is interesting that even though the Baldrige Awards were not won for software quality, the Baldrige-winning corporations tend to be much better than most organizations in software quality too.

The most striking difference between Baldrige-class corporations and others is the use of software quality metrics and sophisticated quality measurement systems. Baldrige Award winners start quality measurements early in the life cycle, and continue all the way through development and out into the field. Baldrige winners measure user satisfaction, defect quantities, defect severity levels, defect origins, and defect removal efficiencies.

More than any other group of enterprises surveyed, Baldrige winners tend to measure "defect removal efficiency" or the total percentage of bugs or errors removed prior to delivery of software. From a knowledge of this key metric, Baldrige winners are able to plan an optimal series of reviews, inspections, and test steps that in some cases removes more than 99% of software errors.

The Baldrige Awards have recently been questioned for various reasons, such as possible political maneuvering or the fact that several Baldrige winners have experienced poor financial performance within a year or two after winning the award. (The incidence of financial problems among Baldrige winners appears roughly equivalent to other companies within similar industries.) Nonetheless, the Baldrige Award seems to do more good for the culture of quality, and lead to better results, than almost anything else attempted.

A number of companies such as IBM, Motorola, U.S. Sprint, and the like have internal awards that are derived from the Baldrige Award and utilize the same criteria. These internal awards are also beneficial, and are an excellent precursor to applying for the actual Baldrige Award. However, these internal awards do not have the same status or prestige with clients and the outside world.

Barriers to Software Quality Measurement

Historically, software quality was measured in terms of "defects found per 1000 source code statements" (normally abbreviated to KLOC, where K stands for 1000 and LOC stands for lines of code).

Unfortunately, the KLOC metric contains a built-in paradox which causes it to give erroneous results when used with newer and more powerful programming languages, such as Ada, object-oriented languages, or program generators.

The main problem with the KLOC metric is that this metric conceals, rather than reveals, important quality data. For example, suppose a company has been measuring quality in terms of "Defects per KLOC" which has been a common practice. A project coded in FORTRAN might require 10,000 lines of code, and might contain 200 bugs, for a total of 20 defects per KLOC.

Now suppose the same project could be created using a more powerful language such as C ++, which required only 2,000 lines of code and contained only 40 bugs. Here too there are 20 defects per KLOC, but the total number of bugs is actually reduced by 80%.

In the FORTRAN and C ++ examples shown above, both versions provided the same functions to end users, and so both contain the same number of Function Points. Assume both versions contain 100 Function Points. When the newer metric "Defects per Function Point" is used, the FORTRAN version contains 2.00 defects per Function Point, but the C++ version contains only 0.40 defects per Function Point. With the Function Point metric, the substantial quality gains associated with more powerful high-level languages can now be made clearly visible.

In 1986, Function Point users formed a non-profit association, the International Function Point Users Group, commonly known as IFPUG. (For additional information, IFPUG headquarters is located in Blendonview Office Park, 5008-28 Pine Creek Drive, Westerville, Ohio 43081-4899; telephone 614 895-7130.) Note that in 1994 the IFPUG

organization voted to change its name, as a result of broadening its scope from only function points to a full range of metrics and measurement topics. The new name is the North American Software Metrics Association (NASMA).

Best in Class Quality Results

The best software projects within the organizations that constitute roughly the upper 10% of the groups that SPR has assessed have achieved remarkably good quality levels. Here are quality targets derived from "best in class" software projects and organizations:

Defect potentials of less than 1 defect per Function Point.
(Sum of defects found in requirements, design, code, user documents, and bad fixes.)

Cumulative defect removal efficiency averages higher than 95%.
(All defects found during development, compared to first year's customer-reported defects.)

Average less than 0.025 user-reported defects per Function Point per year.
(Measured against valid, unique defects.)

Achieve 90% "excellent" ratings from user-satisfaction surveys.
(Measured on topics of product quality and service.)

Allow zero error-prone modules in released software.
(Modules receiving more than 0.8 defects per Function Point per year.)

Improve software quality via defect prevention and defect removal at more than 40% per year. (Baseline is the current year's volume of customer-reported defects.)

Best Industries for Software Quality Control

There are over 250 industries operational in the United States. Software Productivity Research has only collected data from about 50 of these industries, so there are plainly major gaps in this research. Nonetheless, several U.S. industries stand out as being much better than average in terms of software quality control.

In general, the best industries are those that meet these three criteria: 1) The software produced is an integral part of the equipment the industry markets; 2) The equipment will not operate successfully unless the software quality levels are high; 3) Either human life or major business functions or both depend upon the reliability of the marketed products and equipment.

Following are the industries with much better than average quality levels:

1) Computer manufacturers
2) Defense and military software manufacturers
3) Medical instrument manufacturers
4) Telecommunication equipment manufacturers
5) Avionics and aircraft equipment manufacturers

Within these industries, it is the systems and embedded software that have the best overall quality levels. (Note: systems software is defined as programs or systems which control physical devices such as a telephone switch or a computer or a radar set. Embedded software is that which is resident within a physical device, such as on board a cruise missile.)

The software produced by these industries typically utilize a combination of defect prevention methods, pre-test defect removal methods such as formal inspections, and multi-stage testing by testing specialists. The number of explicit defect removal stages within these industries can encompass four to six kinds of review and inspections, and up to 10 different test stages such as unit test, new function test, component test, regression test, stress or performance test, quality assurance test, integration test, system test, field test, and acceptance test. These industries typically have well staffed and adequately funded software quality assurance departments.

These same five industries are most likely to have good measurements of defect potentials and defect removal efficiency. They are also the most likely to use techniques such as formal design and code inspections. This explains why the average defect removal efficiency within these industries is about 95%, or roughly 10% higher than U.S. norms.

Clean Room Development

The concept of "clean room development" is based on the idea of total prevention of software defects, so that no problems are passed on to the next step in a development cycle. (The concept originated in the special rooms developed for VLSI manufacturing, which used filtered air, special garments, and other approaches to keep contaminants out.)

Dr. Vic Linger and Dr. Harlan Mills of IBM's Gaithersburg facility were the pioneers of this approach, and other industry analysts such as Dr. Gerald Weinberg have also espoused clean-room development methodology.

The data on clean-room development is favorable, but somewhat ambiguous. The projects which have been developed using clean-room methods and protocols do achieve very high quality levels. However, it is not yet certain if the results would be equally favorable for all kinds of software, or just selected systems and scientific applications.

It is not yet clear how the clean-room idea will mesh with approaches such as object-oriented analysis and design or client-server applications. It is also unclear how the clean-room method would work with the many MIS software projects that have highly volatile

requirements, where the rate of change in the fundamental requirements for the project exceed 2% per month.

Client-Server Quality

As client-server software applications become more pervasive, there is an urgent need to explore the quality levels, inspection approaches, testing methods, and other topics of significance.

In the original survey performed in 1992, there was insufficient data available to discuss client-server quality levels at all. There is still a shortage of data in 1994, but it is no longer a total shortage.

Unfortunately, the majority of client-server applications reviewed by the author and his staff have been somewhat careless of quality control approaches. From very preliminary and partial data, the defect levels of client-server applications do not appear to be any lower than the U.S. norm for mainframe software of about 5 bugs or errors per function point.

The defect removal efficiency levels against client-server applications appear to lag those of mainframe software, and none of the preliminary results indicate defect removal efficiencies higher than 90% and many are below 80%. Client-server projects seldom use inspections or much in the way of rigorous development approaches. These findings, if they continue without improvement, can mean three things in the future: 1) Dissatisfied client-server users; 3) Low reliability levels of client-server applications; 3) High maintenance costs when client-server applications themselves become legacy systems.

In the context of exploring client-server software quality, it has been noted that there are no metrics nor any current data available on an emerging and important topic: data quality. An unexpected byproduct of the explosive growth of client-server software has been an apparent reduction in data integrity since many client-server applications are outside the scope of authority of corporate data administration groups.

Cost of Quality

A number of companies utilize the "cost of quality" concept which was made popular by Phil Crosby's well-known book, Quality is Free. The cost of quality concept utilizes three cost buckets for exploring software quality economics: 1) Prevention costs; 2) Appraisal costs; 3) Failure costs.

The cost of quality concept originated in the manufacturing sector, and is not necessarily an optimal concept for software quality. From both an economic and a psychological point of view, the phrase "cost of quality" seems a poor choice since it is the lack of quality that is expensive for software.

A cost structure more suited to the nature of software work would expand upon the three cost buckets of the original cost of quality concept, and resemble the following:

1) Prevention costs
2) Non-test defect removal costs (reviews, inspections, walkthroughs, etc.)
4) Testing defect removal costs
5) Post-release defect removal costs
    Costs associated with explicit warranties
    Costs associated with good will
6) Post-release customer support costs
7) Product recall costs
8) Litigation and damages costs

The purpose of this expanded set of quality cost buckets is to allow accurate economic measurement of the impact of various levels and severities of software defect rates.

Cost per Defect

The widely used "cost per defect" metric is highly unreliable, and appears to be economically invalid under a number of conditions. The popular software anecdote that "it costs 100 times to fix a bug in the field as in development" and its many variations has become a commonplace but may not be accurate in many cases.

One of the problems with the cost per defect metric is that it tends to ignore fixed costs, and hence can fluctuate in counter-intuitive ways. Here are two simplified examples to illustrate the problem:

Case 1: Suppose a poorly developed Ada program of 25,000 LOC is being tested, and a total of 500 bugs are found during test. Assume that test case preparation cost $5,000; executing the tests cost $20,000; and fixing the bugs cost $25,000. The entire testing cycle cost $50,000 so the cost per defect of the 500 bugs is exactly $100.

Case 2: Assume that the application used modern defect prevention approaches, so that the number of bugs found during testing was only 50 instead of 500, which is an order of magnitude improvement. In this scenario, test case preparation cost $5,000; executing the test cases cost $17,500; and fixing the bugs cost only $2,500. Now the total testing cost has declined to $25,000. However, the cost per defect for the 50 bugs has risen to $500. Obviously, test case preparation is a fixed cost, and test case execution is comparatively inelastic and only partly dependent upon the number of defects encountered.

As can be seen, the cost per defect metric tends to escalate as quality improves and does not capture the real economic advantages of higher quality. Fortunately the function point metric is now being used for economic studies of quality control. Since Ada applications average about 70 statements per function point, let us assume that both examples are 350 function points in size. The testing cost per function point in Case 1 was $142.87 for the

poor quality example. The testing cost per function point in Case 2 was $71.42 for the high quality example. As can be seen, the function point metric correlates exactly with the real economic gains while the cost per defect metric would be invalid in this example.

Current U.S. Averages for Software Quality

Based on a study published in one of the author's books Applied Software Measurement (McGraw-Hill, 1991), the average number of software errors is about five per function point. This data has been comparatively stable for the United States as a whole between the mid 1980's and 1994.

Specific companies, however, have been able to make notable reductions in their potential defects and to do so rapidly. In fact, the best results have been quality improvements that approximate 40% per year, compounded for several years in a row. Both between and within industries the range of quality results is alarmingly large. Note that software defects are not found only in code, but originate in all of the major software deliverables, in the following approximate quantities:

Defect Origins	Defects per Function Point
Requirements	1.00
Design	1.25
Coding	1.75
Document	0.60
Bad Fixes	0.40
Total	5.00

These numbers represent the total numbers of defects that are found and measured from early software requirements throughout the remainder of the life cycle of the software. The defects are discovered via requirement reviews, design reviews, code inspections, all forms of testing, and user-reported problem reports.

Data Quality and Data Metrics

The topic of data quality was not included in the original 1992 survey, but has catapulted into significance in the world of client-server applications. Unfortunately, as of 1994 there are no known normalizing metrics for expressing the volume of data a company uses, the quality levels of the data, and the costs of creating data, migrating data from platform to platform, correcting errors, or destroying obsolete data.

There is so little quantified information on data quality that this topic is included primarily to serve as a warning flag that an important domain is emerging, and substantial research is urgently needed.

Several commercial companies and Dr. Richard Wang of MIT have begun to address the topic of data quality, but to date there are no known published statistical studies that include the volumes of data errors, their severity levels, or the costs of removing data errors.

There is not even any accepted standard method for exploring the "cost of quality" for data errors in the absence of any normalizing metrics. (Note that the author's company, Software Productivity Research, is now exploring the concept of a "data point" metric that can be mathematically related to the function point metric. This is a research project and all technical contributions would be welcome.)

Defect Prevention Methods

A host of technologies have some effect in defect prevention, or reducing the probable number of errors which might occur in software projects. There are too many to discuss all of them, but a few of the most successful defect prevention methods include: Joint Application Design (JAD), prototyping, and various flavors of reusability such as reusable designs and reusable source code.

It is an interesting fact that formal design and code inspections, which are currently the most effective defect removal technique, also have a major role in defect prevention. Programmers and designers who participate in reviews and inspections tend to avoid making the mistakes which were noted during the inspection sessions.

It is obvious on theoretical grounds that object-oriented analysis and design should be an effective defect prevention mechanism. Unfortunately, as of 1994 there is no empirical evidence to support the theory, and indeed there is some evidence that the steep learning curve associated with OO analysis and design may result in higher than normal front-end defect levels for at least the initial projects that use OO approaches.

Defect Removal Efficiency

Complementing the Function Point metric are measurements of defect removal efficiency, or the percentages of software defects removed prior to delivery of the software to clients.

The U.S. average for defect removal efficiency, unfortunately, is currently only about 85% although top-ranked projects in leading companies such as AT&T, IBM, Motorola, Raytheon, and Hewlett Packard achieve defect removal efficiency levels well in excess of 99% on their best projects.

All software defects are not equally easy to remove. Requirements errors, design problems, and "bad fixes" tend to be the most difficult. Thus, on the day when software is actually put into production, the average quantity of latent errors or defects still present tends to be about 0.75 per Function Point, with the following distribution:

(Data Expressed in Terms of Defects per Function Point)

Defect Origins	Defect Potentials	Removal Efficiency	Delivered Defects
Requirements	1.00	77%	0.23
Design	1.25	85%	0.19
Coding	1.75	95%	0.09
Document	0.60	80%	0.12
Bad Fixes	0.40	70%	0.12
Total	5.00	85%	0.75

Note that at the time of delivery, defects originating in requirements and design tend to far outnumber coding defects. Data such as this can be used to improve the upstream defect prevention and defect removal processes of software development. The best companies are using state-of-the art methods to lower their defect potentials, and coupling that with state-of-the-art methods for removing defects with high efficiency. The results can be quite impressive: Defect potentials of less than 2.0 defects per Function Point coupled with removal efficiencies in excess of 99%.

Following are the approximate ranges of defect removal efficiency levels for selected kinds of defect removal activities:

Defect Removal Activity	Ranges of Defect Removal Efficiency
Informal design reviews	25% to 40%
Formal design inspections	45% to 65%
Informal code reviews	20% to 35%
Formal code inspections	45% to 70%
Unit test	15% to 50%
New function test	20% to 35%
Integration test	25% to 40%
System test	25% to 55%
Low-volume Beta test (< 10 clients)	25% to 40%
High-volume Beta test (> 1000 clients)	60% to 85%

It is obvious that no single defect removal operation is adequate by itself. This explains why "best in class" quality results can only be achieved from synergistic combinations of defect prevention, reviews or inspections, and various kinds of test activities.

Figure 1 illustrates the ranges of defect potentials and defect removal efficiency levels that are associated with three levels of software quality control: malpractice, U.S. averages, and best-in-class performance.

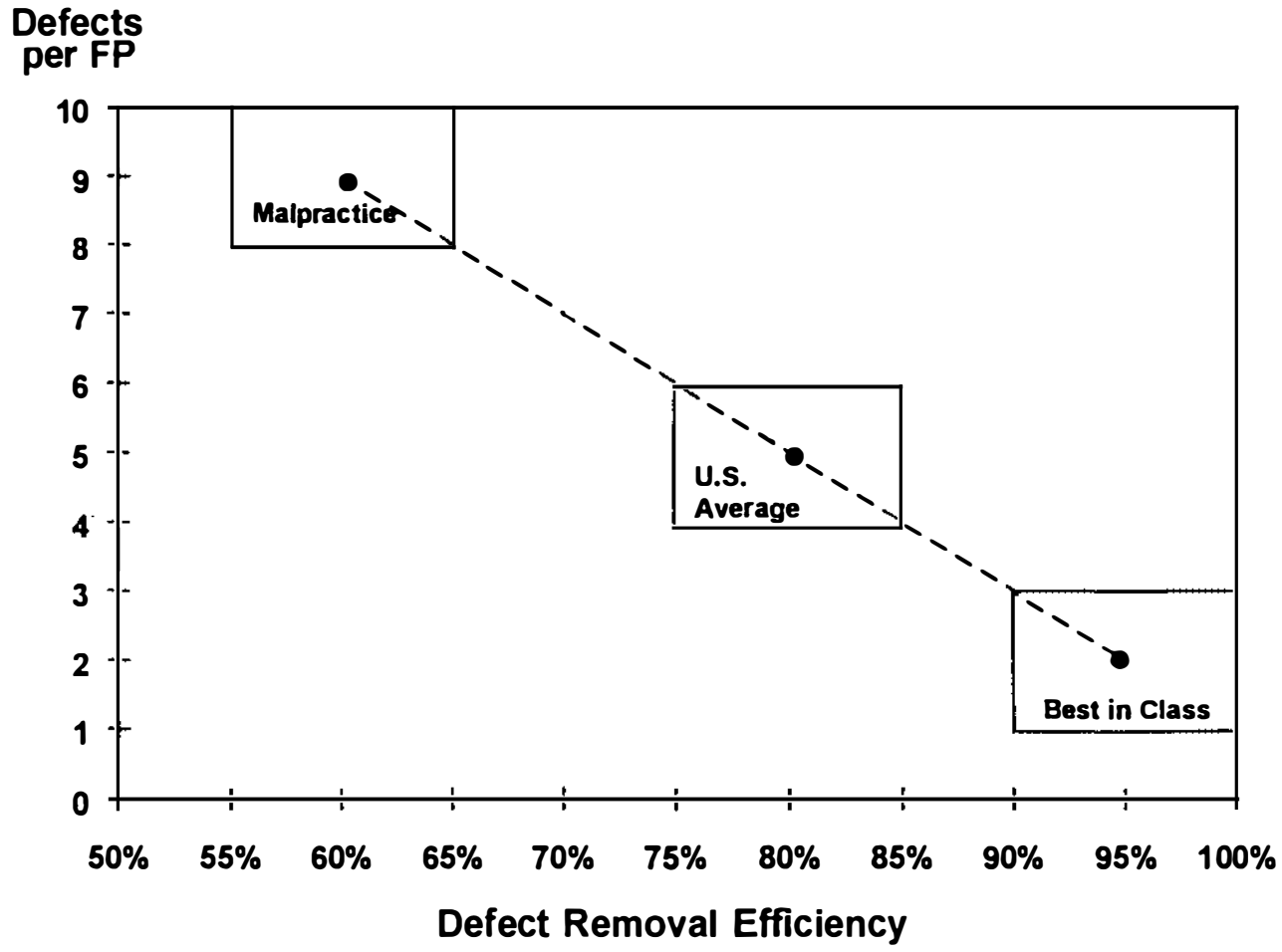


Figure 1: U.S. Ranges of Defect Potentials and Defect Removal Efficiency Levels

DoD Standard 2167A

Since military software ranks among the best in terms of quality levels, it must be concluded that the software quality methods included in various military and Department of Defense standards have a positive and beneficial impact.

However, there is another aspect which should also be addressed. Of the five industries which typically create software with the highest quality levels (aerospace manufacturers, computer manufacturers, defense, telecommunication manufacturers, and medical instruments) the military software producers create twice as much paperwork as the second-place industry (telecommunications). Further, the costs of military software average more than 100% higher than most civilian projects, and more than 50% higher than the other industries which consistently create high-quality software.

Formal Design and Code Inspections

Although formal design and code inspections originated more than 20 years ago, they still are the top-ranked methodologies in terms of defect removal efficiency. (Michael Fagan, formerly of IBM Kingston, first published the inspection method.) Further, inspections have a synergistic relationship with other forms of defect removal such as testing, and also are quite successful as defect prevention methods.

Most forms of testing are less than 30% efficient in finding errors or bugs. The measured defect removal efficiency of both formal design inspections and formal code inspections is sometimes more than 60% efficient, or twice as efficient as most forms of testing.

A combination of formal inspections and formal testing by test specialists are the methods which are most often associated with projects achieving a cumulative defect removal efficiency higher than 99%.

High Volume Beta Testing

Commercial software producers with large numbers of clients (more than 10,000 for example) sometimes approach software quality in a fashion that is not possible for custom software with low usage.

The number of bugs found during testing is roughly proportional to the number of test personnel that are executing the software. The high-volume commercial software producers sometimes have more than 1,000 customers simultaneously participating in external Beta tests. Any software product with more than 1000 concurrent users testing it at once is likely to have a rather good defect removal efficiency: 75% or higher. However, this method of testing is only available to companies such as Computer Associates or Microsoft who have many thousands of customers. Even within this restricted set, not every product has enough customers to allow the kind of high-volume testing that is extremely efficient.

ISO 9000 Certification

It should be noted that ISO 9000 certification only became mandatory in 1992, so there is still very little data yet available even in 1994. In the course of the SPR survey, a number of companies were contacted that were in the process of completing their ISO certifications or which had recently been certified. Similar companies were also contacted which were not applying for certification. So far as can be determined, there is not yet any tangible or perceptible difference in the quality levels of companies which have been certified and similar groups which have not. Of course, there was less than a year of history by which to judge when this report was first published.

As of 1994, there is still a remarkable lack of quantified results associated with the ISO 9000-9004 standard set. To date it does not seem to have happened that with several years of history, the ISO-certified groups have pulled ahead of the non-certified groups in some tangible way. The only tangible aspects visible remain the higher costs of the ISO certification process and the time required to achieve certification.

After several years, it is obvious that the ISO standards have had a polarizing effect on software producers, as well as on hardware manufacturers. Some companies such as Motorola have challenged the validity of the ISO standards, while others such as ViewLogic have embraced the ISO standards fully.

The reaction to ISO certification among software groups to date has been so negative that several industries were contacted (fiber optics, electronics manufacturing). Here the reaction is often even more negative or polarized, and the ISO certification process has been asserted to be ineffective for quality control as currently performed.

Some ISO adherents have observed that it is not the place of the ISO 9000-9004 standards to actually improve quality. Their purpose is simply to ensure that effective practices are followed, and are not in and of themselves a compilation of best current practices.

Indeed, some companies have pointed out that compliance is fairly easy if you already have a good quality control approach in place, and that ISO certification result tends to raise the confidence of clients even if quantification is elusive.

On the whole, the data available from the ISO domain as of mid 1994 remains insufficient to judge the impact of the ISO standards on these topics: 1) Defect potentials; 2) Defect removal efficiency levels; 3) Defect prevention effectiveness; 4) Reliability; 5) Customer satisfaction levels; 6) Data quality.

Object-Oriented Quality Levels

The object-oriented world has been on an explosive growth path. The usage of OO methods has now reached a sufficient critical mass to have created dozens of books, several OO magazines, and frequent conferences such as OOPSLA and Object World.

Unfortunately, this explosion of interest in OO methodologies has not carried over into the domain of software quality. There are a number of citations on software testing, but little or no empirical data on actual software quality levels associated with the OO paradigm.

From very preliminary and provisional results, it appears that object-oriented languages such as C++ and Smalltalk may have reduced defect levels compared to normal procedural languages such as C or COBOL or FORTRAN. Defect potentials for OO languages are perhaps in the range of 1 per function point, as opposed to closer to 1.75 per function point for normal procedural languages.

However, the defect potentials for object-oriented analysis and design have not yet showed any reduction, and indeed may exceed the 1.25 defects per function point associated with the older standard analysis and design approaches such as Yourdon, Warnier-Orr, and Gane & Sarson.

Unfortunately, OO analysis and design tends to have a high rate of abandonment, as well as projects which started with OO analysis and design but switched to something else in mid stream. These phenomena make it difficult to explore OO quality.

There is obviously a steep learning curve associated with most of the current flavors of OO analysis and design, and steep learning curves are usually associated with high levels of errors and defects.

There is not yet enough data available in 1994 to reach a definite conclusion, but there is no convincing published evidence that OO quality levels are dramatically better than non-OO quality levels for OO analysis and design. The data on OO languages is somewhat more convincing, but still needs more exploration. On theoretical grounds it can be claimed that OO quality levels should be better, due to inheritance and class libraries. However, that which is inherited and the class libraries themselves need to be validated carefully before theory and practice coincide.

Quality Assurance Departments

The empirical results of having formal quality assurance departments ranges from very good to negligible. The very good results come from well staffed and well funded software quality assurance groups that are able to take a proactive role in quality matters.

The most effective quality assurance groups typically average about 5% of the total software staff in size. These groups also start work on quality early, and assist software teams with special studies such as defect estimation modeling. They also measure quality and defect removal efficiency levels, serve as moderators or recorders in formal inspections, and are very active in their participation.

The marginal to poor results from quality assurance departments are often associated with understaffing and underfunding. QA groups with, for example, where staffing is less than 2% of the software development staff can't keep up with the work.

Note that quality assurance is not just testing. Indeed, testing should normally be carried out by either the development teams themselves, or by testing specialists.

Quality assurance groups should have the authority and clout to resist excessive schedule pressure or unwise attempts to deliver shoddy applications. This means a separate organization that has its own chain of command, and considerable authority.

Active quality assurance organizations are often associated with better than average quality results. They are almost always found in the "Best in Class" organizations with low defect potentials and high defect removal efficiency levels.

Quality Definitions

Few terms used in modern business are as controversial or ambiguous as the word "quality." In performing the SPR surveys, many different concepts of what quality means were encountered. Unfortunately, some of the more common definitions for quality are unmeasurable, illogical, or both. A few examples can illustrate some of the problems encountered:

The definition of quality that it means "conformance to requirements" was encountered in a majority of MIS software organizations, but seldom occurs among commercial, military, or systems software producers even though it is included in several IEEE standards. The obvious problem with this definition is that requirements themselves contribute about 15% of the total volume of errors associated with software projects. A definition of quality that means conformance to a key source of errors prevents the direct measurement of requirements defects themselves and leaves a major gap in quality data record keeping.

Also, the observed rate at which new requirements are created during software development projects is about 1% per month during the design and coding phases! This phenomenon raises serious logical questions about exactly when conformance to requirements starts and stops.

Among the definitions of quality contained in various IEEE and ISO standards are a number of aspects under the general topic of quality: Portability, reliability, efficiency, accuracy, error, robustness, and correctness. Each of these terms, in turn, is defined more fully. (Refer to ANSI/IEEE Std 729, IEE83, IEEE Std 610, ISO 9000-9004, for more extended discussions of quality terms.)

These definitions were originally created in the 1970's, long before software warranties were common and very long before the specter of litigation and consequential damages was on the software horizon.

Several of these nominal quality factors seem to be irrelevant to quality in any normal usage of the word, and should probably be removed in the future. For example, the concept of "portability" is an important business topic, but irrelevant to quality. In a similar fashion, "efficiency" is a significant topic, but has nothing to do with quality.

Many of the companies contacted utilized pragmatic definitions of quality that included these three aspects: 1) Low defect rates after delivery; 2) High user satisfaction levels after delivery; 3) Rapid response to customer service requests.

Conformance to requirements is often dealt with in a pragmatic manner. If the requirements are rational, technically achievable, and defined early in the development cycle then they should be met. If the requirements occur too late or are technically impossible, then they cannot be dealt with in the context of the current release and the client should be apprised of the situation promptly.

Quality Function Deployment (QFD)

Quality Function Deployment is a kind of formal, structured group activity involving clients and development personnel. In the course of the QFD sessions, the users' quality criteria are enumerated and defined, and then the product is built from day one to ensure that the quality criteria are implemented.

For the kinds of software where clients and developers can meet and have serious discussions, QFD appears to work very well. This technique has been used in Japan for a number of years for hardware projects, and since about 1990 for software projects.

The initial results are favorable enough to recommend at least substantial experiments with the QFD approach. The only caveat is that for certain kinds of software, such as spreadsheets or word processors with thousands of users, it is not possible to gather all of the users together or explore their quality needs.

Quality Laggards

Although the survey concentrated on "best in class" enterprises, the data collected was also capable of highlighting enterprises at the bottom of the quality pyramid. These enterprises have the following characteristics: 1) No software quality measurement program of any kind; 2) No usage of formal inspections; 3) No knowledge of the concepts of defect potentials and defect removal efficiency by either managers or software technical personnel; 4) Either no quality assurance group, or a group that is severely understaffed (i.e. ratios such as having only one QA member for more than 50 software development personnel); 5) No trained testing specialists available; 6) From a low of one to a high of perhaps four distinct testing stages; 7) Defect potentials averaging more than six defects per Function Point; 8) Defect removal efficiency averaging less than 75%.

Schedule Pressure

The interaction of software schedules and software quality is poorly understood by many software managers and technical personnel as well. Software projects where high quality levels are the initial target, and where effective defect prevention is used in concert with formal inspections and formal testing have the highest probability of achieving on-time or early delivery dates. Such projects also have lower costs and higher user satisfaction levels, as well as the lowest volumes of delivered defects when compared to projects that were rushed to achieve arbitrary or irrational schedules.

Projects where the schedules are so irrational that there is no possibility of achieving them, and where managers or clients insist on various short cuts have the highest probability of being canceled, or running late, and of over running their budgets. When finally delivered, these projects also have the highest maintenance costs and the largest volumes of customer reported defects, as well as very low levels of user satisfaction.

The profiles of "healthy" and "pathological" software projects have been studied for more than 30 years, and are clearly understood by best in class software producers. If you skimp or compress the front end of a software development cycle, you will stretch out the testing to an inordinate degree. Unfortunately, the pathological projects give the false appearance of looking like they are ahead of schedule until testing begins. Then it is discovered that the project does not work, and a frantic cycle of testing, debugging, and late night repairs begins. Figure 2 illustrates the profiles of healthy and pathological software projects.

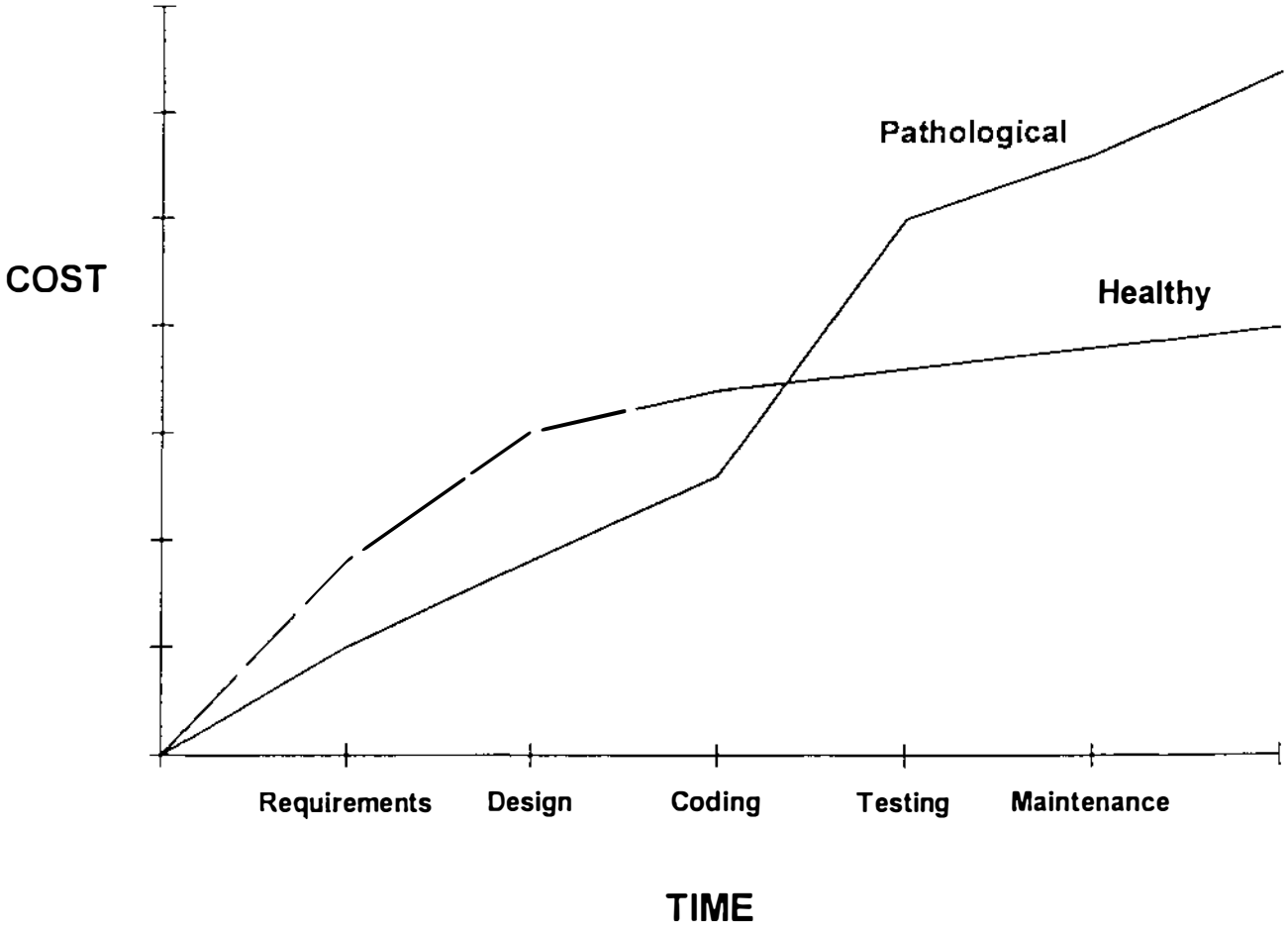


Figure 2: Development Profiles of Healthy and Pathological Software Projects

SEI Maturity Levels

The Software Engineering Institute (SEI) maturity level concept is one of the most widely discussed topics in the software literature. SEI has claimed that software quality and productivity levels correlate exactly with maturity level. As of 1993, there was not yet a lot of empirical data which either supports or challenges these assertions. SEI itself was only just beginning to start data collection, so up until 1993 the assertions were purely theoretical.

In terms of quality, the very preliminary data available to date indicates that for maturity levels 1, 2, and 3 average quality may rise with maturity level scores. However, one of the military services reported a counter example, and has stated that at least some software produced by a level 3 organization was observed to be deficient.

Further, some of the software projects created by organizations at SEI level 2 are just as good in terms of quality as those created by SEI level 3. Indeed, there are even good to excellent quality projects created by some SEI level 1 organizations.

Finally, there has been no quantitative comparison between software produced by the few hundred enterprises that have used the SEI maturity level concept and the more than 30,000 U.S. software producers that have not adopted the maturity concept at all.

The topic of the exact costs and the value of the SEI maturity levels needs a great deal more research and quantification before definite conclusions can be reached. In 1994, the U.S. Air Force is commencing a study to demonstrate the costs and quality levels of the higher SEI maturity levels of 3 and above. This study should generate additional information before the end of 1994.

Six-Sigma Quality Levels

The Motorola corporation has received substantial interest in its famous "six sigma" quality program for hardware and manufactured components. The phrase "six sigma" refers to defect densities of one part in a million. Another way of looking at it would be to achieve a cumulative defect removal efficiency rate of 99.999999% prior to delivery of a product.

Since the current U.S. average for software defect removal efficiency is only about 85%, and quite a few software products are even below 60%, it may be seen that software producers have some serious catching up to do.

As it happens, there are a few software products which appear to have achieved six-sigma quality levels. In the past 25 years, the author has observed perhaps half a dozen such projects out of a total of several thousand.

Surprisingly, the prognosis for achieving six-sigma quality levels for software is fairly good. The best of the available technologies today can approach this level if a full suite of defect prevention and defect removal operations are carried out synergistically.

Testing

Testing has been the basic form of defect removal since the software industry began. For perhaps a majority of software projects, it is the *only* form of defect removal utilized. Considering the importance of testing, it is surprising that so little quantitative data has been published on this topic.

Most forms of testing, such as unit test by individual programmers, are less than 30% efficient in finding bugs. That is, less than one bug out of three will be detected during the test period. Sometimes a whole string of test steps (unit test, function test, integration test, and system test) will find less than 55% of the bugs in a software product. By itself, testing has never sufficient to ensure really high quality levels.

Testing should be part of a synergistic and integrated suite of defect prevention and defect removal operations that may include prototyping, quality assurance reviews, pre-test inspections, formal test planning, multi-stage testing, and measurement of defect levels and severities.

It should be noted that inspections or testing of client-server applications and testing of object-oriented applications are still emerging topics. The number of articles and books on these subjects is increasing, as are commercial tools. However, as of 1994 there is not yet a body of proven techniques for ensuring high quality in either the client-server or the OO domain.

Total Quality Management (TQM) for Software

About half of the Total Quality Management (TQM) experiments in the U.S. are successful and improve quality, and the other half are failures. The successful use of TQM correlates strongly with the seriousness of the commitment and the depth of understanding by executives and management.

TQM only works if it is really used. Giving the TQM concepts lip service but not really implementing the philosophy only leads to frustration.

The TQM concept is not a replacement for more traditional software quality approaches. Indeed, TQM works best for organizations that are also leaders in traditional quality approaches. Specifically, some of the attributes of companies which are successful in their TQM programs include the following: 1) They also have effective software quality measurement programs which identify defect origins, severities, and removal efficiency rates; 2) They utilize formal reviews and inspections before testing begins; 3) Their software quality control was good or excellent even before the TQM approach was begun.

Conversely, enterprises which lack quality metrics, which fail to utilize pre-test defect prevention and removal operations, and which lag their competitors in current software quality control approaches tend to gain only marginal benefits, if any, from the adoption of the TQM method. Such enterprises are very likely to use TQM as a slogan, but not to implement the fundamental concepts.

The operative word for Total Quality Management is *total*. Concentrating only coding defects, measuring only testing, and using the older KLOC metric, violates the basic philosophy of Total Quality Management. These approaches ignore the entire front-end of the life cycle, and have no utility for requirements problems, design problems, documentation problems, or any of the other sources of trouble which lie outside of source code.

It is obvious that there are three sets of factors which must be deployed in order for Total Quality Management (TQM) to be successful with software:

1) Adopting a culture of high quality from the top to the bottom of an enterprise.

2) Using defect prevention methods to lower defect potentials

3) Using defect removal methods to raise pre-release efficiencies .

Synergistic combinations of defect prevention methods can reduce defect potentials by more than 50% across the board, with the most notable improvements being in some of the most difficult problem areas, such as requirements errors.

The set of current defect removal methods include such things as formal inspections, audits, independent verification and validation, and many forms of testing.

However, to bring TQM to full power in an organization, the culture of the management community must be brought up to speed on what TQM means and how to go about it.

In general, no one method by itself is sufficient, and a major part of the TQM approach is to define the full set of prevention and removal methods that will create optimal results. This in turn leads to the need for accurate measurement of defect potentials and defect removal efficiency.

How TQM will evolve for client-server software or object-oriented software is an unknown topic as of 1994. In theory, TQM should work in both domains, and should be even more effective in an object-oriented domain due to class libraries and inheritance. In fact, the topic remains uncertain.

Summary and Conclusions

Software quality has been a concern for the entire 50-year history of software production. Software quality has also been a topic of research and investigation for more than 30 years.

The current survey supports a number of observations which have been made by other researchers and by other studies: 1) There is no "silver bullet." Multiple approaches are necessary to achieve high software quality levels; 2) Software quality measurement is the primary key to effective quality control; 3) Formal reviews and inspections before testing begins augment testing itself, and are necessary in order to achieve cumulative defect removal efficiency levels higher than 95%; 4) The culture of software quality must permeate an enterprise from top to bottom to be successful. Executives, management, and technical staff must all contribute to the vision of high quality.

The "best in class" software producers now have defect potentials in the range of about 1.0 to 3.0 errors per Function Point, coupled with defect removal efficiencies that range from 95% to more than 99% for mission-critical software. This combination yields delivered defect totals of only 0.01 to 0.15 defects per Function Point, or roughly an order of magnitude better than current U.S. norms.

Software is not yet routinely achieving six-sigma quality levels, but advances in software metrics, defect prevention methods, and defect removal methods make six-sigma quality for software a strong future possibility.

Suggested Readings

The available literature on software quality is growing very rapidly. A full bibliography as of mid 1993 would contain more than 100 books and 1000 journal citations. Following are a small number of useful reference books on aspects of software quality. Some of the references are "classics" that have stood the test of time, and some are recent publications that are attempting to break new ground.

Charette, Robert N.; Software Engineering Risk Analysis and Management; McGraw-Hill, New York, 1989.

Crosby, Phil; Quality is Free -- The Art of Making Quality Certain; McGraw Hill, New York, 1979.

Dunn, Robert & Ullman, Richard; Quality Assurance for Computer Software; McGraw-Hill, New York, 1982.

Dunn, Robert; Software Defect Removal; McGraw-Hill, New York, 1984.

Dunn, Robert; Software Quality Concepts and Plans; McGraw-Hill, New York; 1990.

Freedman, Daniel P. and Weinberg, Gerald M.; Handbook of Walkthroughs, Inspections, and Technical Reviews; Dorset House Press, New York; 1992.

Hart, Christopher W.L and Bogan, Christopher E.; The Baldrige; McGraw-Hill, New York, 1992.

Hetzel, Bill; The Complete Guide to Software Testing; QED Press, Wellsley, MA; 1992.

Ishikawa, Kaoru; What is Total Quality Control? The Japanese Way; Prentice Hall, Englewood Cliffs, NJ, 1985.

Jones, Capers; Applied Software Measurement; McGraw-Hill, New York, 1991

Jones, Capers; Software Productivity and Quality -- The Worldwide Perspective; IS Management Group, Carlsbad, CA, 1993.

Jones, Capers; Assessment and Control of Software Risks; Prentice Hall, Englewood Cliffs, NJ, Autumn of 1993.

Jones, Keith A.; Automated Software Quality Measurement; Van Nostrand Reinhold, New York, 1993.

Mosley, Daniel J.; The Handbook of MIS Application Software Testing; Prentice Hall, Englewood Cliffs, 1992.

Myers, Glenford J.; The Art of Software Testing; John Wiley & Sons, New York, 1979.

Perry, William E.; A Structured Approach to Systems Testing; QED Press, Wellsley, MA; 1992.

Roetzheim; William H.; Developing Software to Government Standards; Prentice Hall, Englewood Cliffs, NJ, 1992.

Schulmeyer, G. Gordon; Zero Defect Software: McGraw-Hill, New York, 1990.

Schulmeyer, G. Gordon and McManus, James L.; Total Quality Management for Software; Van Nostrand Reinhold, New York; 1992.

Walton, Mary; The Deming Management Method; Perigee Books, New York; 1986.

Weinberg, Gerald. M.; Quality Software Management, Volume 1, Systems Thinking; Dorset House Press, New York, 1992.

Weinberg, Gerald. M.; Quality Software Management, Volume 2, First-Order Measurement; Dorset House Press, New York, 1992.

Yourdon, Edward; Structured Walkthroughs; Prentice-Hall, Yourdon Press; 1992.

Managing Requirements to Meet Schedules

Paul Dittman
Software Manager

Justin Whitling
Software Engineer

Tektronix, Inc.
Communication Test Products
P.O. Box 500, M.S. 46–115
Beaverton, Oregon 97077
pauld@budley.pen.tek.com
justinw@kiama.pen.tek.com

Abstract

The requirements document helps to insure project focus and accurate scheduling. The proper application of the requirements document and the related processes allows the project team to clearly understand the scope of the project and the impact of proposed changes.

Paul Dittman is the Software Engineering Manager in Tektronix' Communications Test group. During his eighteen years at Tek, he has developed software for over 10 successfully introduced products in 7 different departments. He is often characterized by bright clothes, wild shoes and a penchant for getting into trouble.

Justin Whitling is a software engineer in Tektronix' Communications Test group. During his seventeen years at Tek, he has worked in manufacturing, evaluation and software engineering. His experience spans many products and processes. Having visited over forty countries throughout the world, he likes to encourage co–workers to travel, and wear brighter clothes.

Introduction

Managing and meeting software schedules is often very troublesome for many companies. The problem usually relates to inadequate requirements definition and a failure to understand how changing requirements impact other aspects of a project. This is especially true in projects that are new to a market or are built upon new platforms.

Documenting the requirements and establishing the processes to use these requirements as a basis for other project planning are the first steps in managing a successful software project. This paper will demonstrate how a viable requirements document was created and used to keep a project focused and on schedule.

The Challenge

Three years ago SONET (Synchronous Optical Network), a new standard in the telecommunications industry had just been completed. This new standard would have a major impact in the telecommunications market. The standard allowed for the simultaneous transmission of voice, video and data over a fiber optic line. There was a strong desire at Tektronix to put a new group together to build products for this market.

The challenge for us was how to get into this market. We got a late start and there was a definite window of opportunity that, if missed, would lock us out of the market. This window existed because telephone companies, the major customer, put out bids for products and then standardized on one product. Once they have standardized, they rarely buy other competing products. Our marketing data showed that we had two years to get products into this market or be locked out. We referred to this as our "brick wall" schedule.

The History

At Tektronix, products of the complexity required for this market typically take much longer than two years to develop. It takes time to build and train a new team and it takes considerable time to find out what the customer wants in a product. In our case, two new platforms needed to be developed. The new team had never worked together and new processes needed to be developed. Creeping features were very common in situations like this and were often one of the main time sinks that killed projects. There were many variables and things that could go wrong with very little time to deal with them. We had seen many projects like this take over four years in the past.

On previous projects where the culture had been well established within the working team it seemed very difficult to avoid past problems, largely because of momentum. There was always a feeling of "If I only had it to do all over again I would ....". This new project team had this opportunity. The team was being created from scratch and had the ability to put new processes in place to avoid making mistakes That previous teams had made over and over again.

We started by developing a unique team organization that would develop the two products simultaneously using a single team of engineers instead of two. Each team member would have well defined responsibilities. We wrote explicit job descriptions for each position, did extensive interviewing and hired a group of people who not only fit the jobs well, but had enough experience to know the major software project pitfalls.

With the team in place, we immediately set about defining and documenting a process that would carry us through the rest of the project. This included all of the milestones for the project, deliverables for each milestone, and plan showing all documents that needed to be researched and when they needed to be ready. Included in this was a quality plan, a codebase plan, a tool plan and a way to define the products and manage the changes. Because we were a new team everything needed to be put in writing to avoid ambiguity and to help make the process stick.

The preliminary milestones were Features and Functions Frozen (FFF) and Software Design Complete (SDC). The deliverables for these milestones were a verified requirements document, a schedule at FFF, a software design that would take us through the implementation phase, and a more accurate schedule at SDC (refer to figure 1).

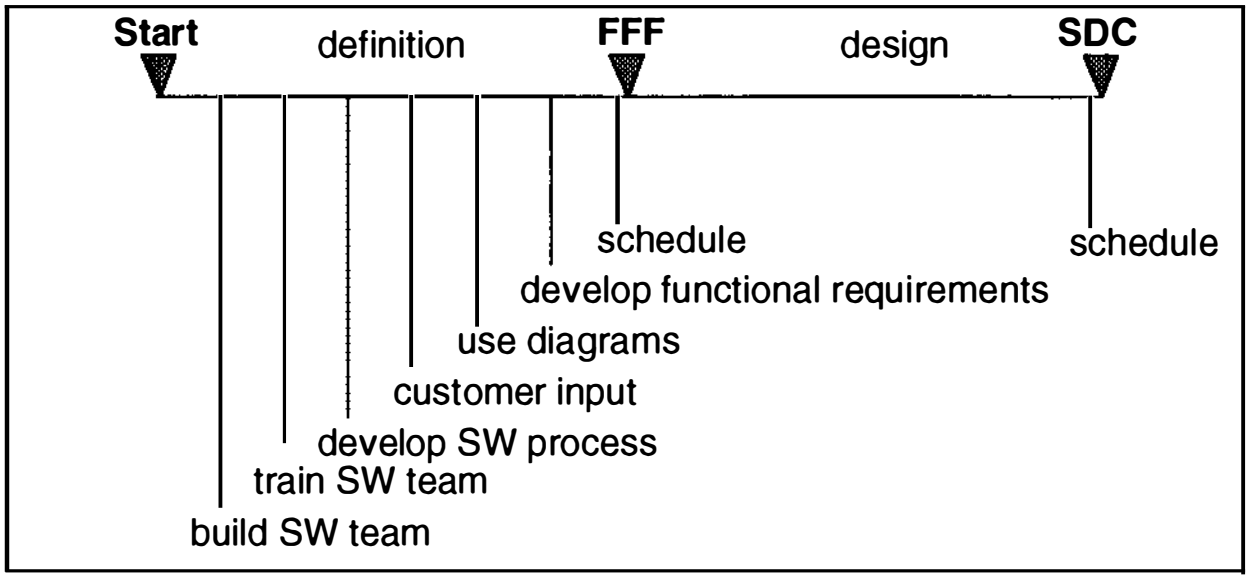


Figure 1. Planned milestones during the beginning of the project

Defining the Products

We talked to customers to determine what they wanted in the products. We were surprised to find out that we got very different answers from each customer even within the same company. The SONET standard was so new that most customers had not had much experience with equipment in this area and were surmising what they thought they might need.

We tried using Quality Function Deployment (QFD); a tool for analyzing features in a product based on market needs and competition. QFD did not help us much since there was little competition at the time and customers gave us very different lists of needs.

We then came up with the idea of "use diagrams" to try to solve the problem. A use diagram is a diagram of the equipment configuration and a short description for a method to solve a very specific problem (refer to figure 2). The actual functionality

of the product could be described by a series of use diagrams, each showing the use of a small group of features or sometimes an individual feature in the instrument.

Performance Verification: Measuring SONET Bit Error Rate (BER)

Task description

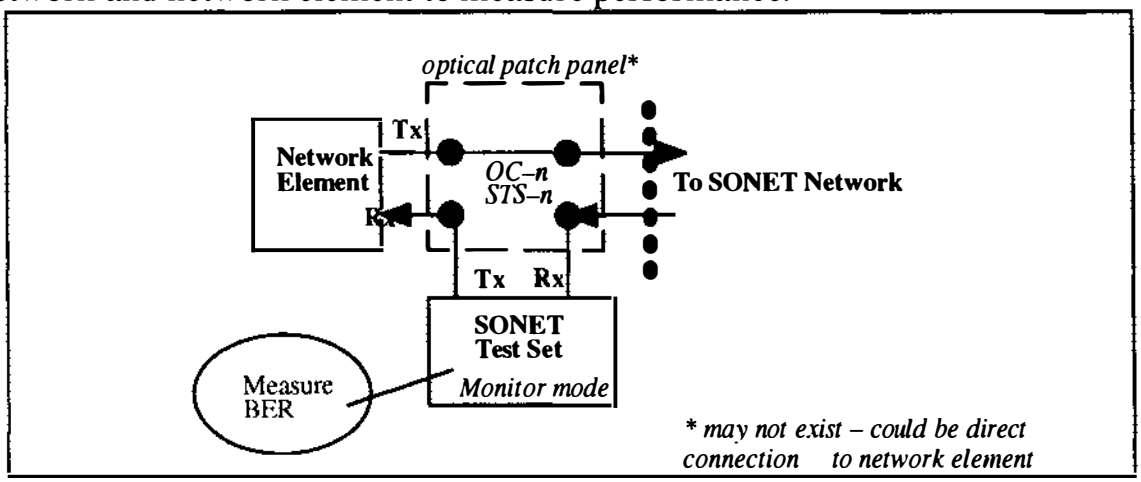
To measure the performance of the upstream and/or downstream SONET paths.

The SONET test set may generate a valid SONET signal (either optical or electrical) .

The SONET Test set will measure the received Bit Error Rate (BER). BER will be calculated from BIP–8(B1, B2, B3) and BIP–2 (V5), PRBS and FEBE. All measurements are made all the time and may be displayed separately or all at once.

Connection Diagrams

These following diagrams show how the SONET test set may be configured to the network and network element to measure performance.



Measuring SONET Signal Performance with unknown payload

Figure 2: Example of a use diagram

We asked the customers to prioritize a variety of use diagrams to correspond with their requirements. This time we got more consistent results and were able to start determining the major features.

We decided to verify what we learned from the use diagrams and also try out a model for our human interface with a prototype. The prototype needed to be something that would have somewhat of the look and feel of the final instrument without costing very much in money, time or resources.

Prototypes in the past consisted of instrument platforms in which temporary code was written to simulate or actually do some of the features of the instrument. These prototypes took time to create and often took on a life of their own. Often the prototype

looked so good that management would order that they be completed and shipped. Since the prototype had not been properly designed, it may not have been possible to complete it and was always a maintenance nightmare.

We elected to force fit a portable computer into the mechanical platform that we wanted to use. We connected the front panel keys to the pc's function keys. We used Plus, a rapid prototyping software package, to simulate the human interface of the instrument. We took this prototype around to customers to watch the problems they had navigating through the human interface. Using the feedback from the prototype the foundation for the feature set and human interface was laid (the prototype did uncover many problems in our original ideas).

The Requirements Document

From the information we collected from the "use diagrams" and the prototype we began writing the requirements document for the products. The creation of this document was not well received by all; some of the engineers and management saw the work that was put into this document as time that could have been applied directly to the implementation of the product. They also saw this document as inhibiting the ability to make quick changes to the definition of the product.

The hardware group had already started on the architectural and lower level design of the hardware. Ambiguity in the features that they were implementing and "creeping" features were already showing up. It soon became apparent to the project manager that the requirements document was necessary.

To write a requirements document it is necessary to know what you intend to do with it (part of the process around the requirements). We identified documents that were needed that could be derived from the requirements document. This helped us identify what should be in the requirements document (refer to figure 3).

Contents of the Requirements Document

Our view of the requirements document was that it should concisely describe the instrument from the customer's point of view. This had to be done with enough detail that the features and the intent would not be ambiguous (refer to figure 4).

In other projects a human interface description or a command reference description (for products that do not have a human interface) are often considered to be the equivalent of a requirements document. These documents are not written from the customer's point of view and changes to these documents are therefore difficult to communicate to all aspects of the project (refer to figure 3).

We were very careful when laying out the table of contents for the requirements document. We wanted to be sure that we captured everything that would be needed by different groups (marketing, manuals, service, manufacturing, etc.) to make the product successful. This had to include information for the life cycle of the product. Our document was divided into 15 chapters.

Table of Contents from the Requirements Document

- "Intent" and "Introduction" describe the purpose and the basic layout of this document.

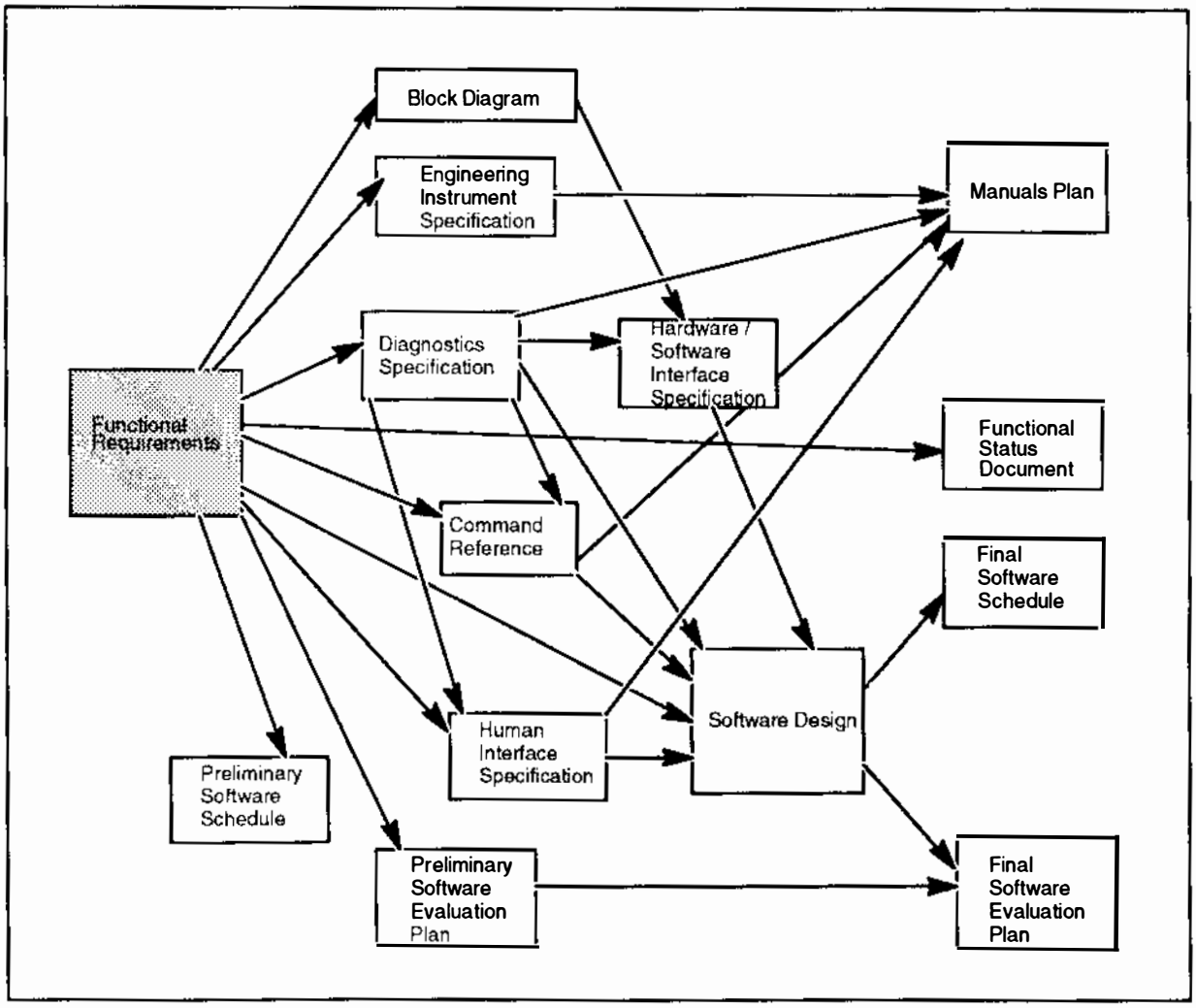


Figure 3: Product documentation road map

- "Basic Test Set" describes the physical description (size, weight, cost etc.) that might be important to a customer.
- "Primary Functions" describes all of the features and capabilities of the basic instrument.
- "Options" refers to features in optional boards that can be added at purchase time or later to the instrument.
- "Human Interface" lists basic mechanisms and testable goals that we were trying to obtain with the human interface of the instrument.
- "Computer Interface" deals with the computer access to the instrument and the protocols to be used in communicating with it.
- "Data Analysis" specifies the more than 100 measurements that the instrument makes.
- "Utility" talks about the disk system, the printing capability and accessories for the instrument.

- "Compatibilities" specifies compatibility between the instrument and other mechanical, electrical or software devices.
- "SDH/SONET Differences" spells out the differences between the U.S. and European versions of the protocol.
- "Diagnostics and Calibration" specifies the capabilities that the instrument has to diagnose and calibrate itself in manufacturing or in the customer's hands.
- "Quality Assurance and Evaluation" details the goals, plan and metrics to be used to achieve a quality instrument.
- "Appendix I" is the list of front and rear panel connectors for the instrument.
- "Appendix II" is a competitive profile of this instrument against it's competitors.

We tried to be as specific as possible when writing the document. The style intention of the requirements was to be more table or bullet oriented and less on the verbose side. This had the advantage of making it easy for people to pick up and read the details of the instrument and easily understand it. It had the disadvantage of making it more difficult to acquire checklists from the document and also made it longer (164 pages in length).

Generator Output During Setup Changes

This section defines the generator output at all times.

Table A-1: Generator Output Summary

Condition	Generator Output
Change to new SONET rate (e.g.: STS–1 to STS–3) or VT rate	Off for 3 seconds (need to tell user)
All other setup changes (eg: PRBS to Fixed pattern)	Always Valid (no alarms generated)

WARNING

The times mentioned above are estimates that are intended to represent the worst case.

Note: A valid signal out of LEO will contain no more than one frame which has any B1 or B2 errors. The data carried in the payload is undefined during this change period. This means that the Path Overhead, including B3, may not be valid. Also any VTs in the payload may not be valid and will have errors in the VT BIP.

Figure 4: Example page from the requirements document

Once the document was written and edited, it was distributed to all groups on the project. Most people were interested in being involved in the review process because the outcome affected what they would be working on. Several weeks were required to review the document. This included time to read the document and make comments and also time to take corrective action on issues that came up.

Scheduling

The requirements document was put under change control giving us a baseline for our scheduling effort. An itemized list of features was extracted from the requirements document to assist in scheduling. The software group estimated time for each feature in terms of a design time and an implementation time (assumptions had to be made as to what the design would be like in order to do this). In some cases time was added for code base (the software platform to start from) and infrastructure work needed to bring up a basic feature. The time estimates on the list were added up to find the end point of the schedule.

The end point turned out to be longer than the window of opportunity (this was no surprise). We gave the list of features and time estimates to the marketing group and asked

them to prioritize the features and remove or delay features until the time fit the deadline (the marketing group was not allowed to change the time estimates). The marketing group was able to do this satisfactorily to give us a baseline schedule. The prioritized feature list was also used as a contingency; low priority features could be removed when new changes or schedule slips occurred.

This method of scheduling in which the schedule was worked out by all interested parties and not just forced to meet an endpoint, brought about a feeling of ownership and commitment within the group. The usual excuses for not meeting schedules were far less frequent.

The Change Process

The hard part was now upon us. How do we control and monitor feature changes such that the requirements document would not quickly become out of date and no longer track the schedule? How would we keep the internal culture that wanted to add new features from subverting the process?

A process was put in place to allow anyone on the project to propose changes to the document. To do this they would submit a form (electronically was the preferred means) to an administrator (refer to figure 5). The administrator once a week would send email to everyone on the project with a summary of the proposed changes pending. Anyone interested or impacted by these changes would come to a meeting that week to discuss the changes. The change form required an impact statement that would have an estimate from the person making the change of what the cost would be.

The change notice meetings became one of the central focal points for communication within the project team. There was a considerable amount of animated discussion that routinely happened at these meetings. The meetings were always productive and changes were usually approved, rejected or delayed during the meeting. Occasionally they were postponed for more information.

The Three Week Rule

If a change notice was approved it was subjected to the "three week rule". Changes could not accumulate more than 3 weeks total slip time (total time impact from all previous changes) without having to remove another feature of equal time value. People would weigh this carefully before submitting changes. This limited the number of "silly" or "nice" features that are usually encountered in a project like this. When features did need to be removed, the prioritized feature list became very handy.

After the meeting the administrator would again send out email telling of the end state of each change notice (approved, rejected or delayed). The change was then incorporated into the requirements under a revision tracking system. The revision tracking system became useful later when it was desired to know why a change was made or who was involved in the meeting (people sometimes conveniently forgot that they had been involved in the decision making process until they saw their signature on the change notice form).

Occasionally the requirements document would be reprinted for the group. Everyone had electronic access to the document so reprinting was rarely needed.

Sonet Change Control Notice
Leo/Gemini Functional Requirements

Initiator: Paul Dittman Date: August 5, 1992

Document: Leo Functional Requirements Rev.#: May 5, 1992

Section: Basic Test Set, Human Interface,Utility

Proposed Change: The firmware in the Leo instrument will only be upgradable via the use of a PC and the System debug monitor. It will not be upgradable directly from the floppy disk as is currently stated in the functional requirements.

This is also true for Gemini except that it was never explicitly stated how this would work.

Impact: (customers, financials, schedules)

This will save approximately 4 man weeks currently in the schedule. If we keep the upgrade capability directly from the floppy disk, it is likely to add an additional 4 weeks to the schedule.

Signatures

Ron Olisar

Paul Dittman

Skip Hillman

Paul Alappat

Aart Konynenberg

V. Prasannan

Barbara Siefken

Others

Disposition

Change Approved: ■

Change Rejected: □

Wait for 2nd Wave: □

Figure 5: Example of a change notice form

Delayed Features

Features that were delayed (usually as a fallout from the three week rule) were marked in a very obvious manner in the requirements document. This was done to make everyone on the project well aware that we were no longer implementing this feature for this release and also to keep track of the feature so that it could be implemented later.

Error Insertion Methods

- ***Rate*** insertion
  Set by rate("1 frame in 20" or "10E−7")
- Continuous insertion
  Set by state (on/off)
- ***(S)*** **S**ingle event
  Single events meet the error criteria based on a single frame.

The following feature has been delayed.

- ***(B)*** Burst insertion
  A burst of errors may be setup as timed duration or a rate.
  Time is from 10msec to 10sec in 10msec steps.
  Burst errors will NOT be repetitive.

NOTE

All insertions are mapped into the above mechanisms. However, menu buttons may be used (maybe even a knob) to control other conditions like pointer adjustment rates.

Figure 6: Example of a delayed feature

As changes were made to the requirements document it was important to insure that these changes rippled through the other documents under change control in the project. These documents included the human interface specification, the command reference specification (commands to control the instrument from another computer), the hardware/software interface specification, and the engineering instrument specification. Changes to these documents that did not originate from the requirements document occasionally happened and did cause problems.

There is a tendency to sneak favorite features into the project even though there was a general awareness that this could introduce inconsistencies that could break the process. "Is it in the requirements document?" became the software battle question when these features would come up. The software group made a pact to not implement any-

thing that was not documented in the requirements. After awhile people became reconciled to using the the process and eventually reduced or removed the problem of "creeping features".

The Functional Status Document

A derivative of the requirements document was the functional status document. This document was essentially a bulletized version of the requirements document. This document was used as a checkoff to show the status of each feature (through partial stages of it's completion). It allowed the evaluation group to know when it was appropriate to start testing the feature. The functional status document provided good communication throughout the design group on the status of features that were being worked on. It allowed manufacturing to know the features they could begin to test in the automated testing environment. It also allowed marketing would know what features could be demonstrated at any point in time.

Things to do Differently Next Time

The largest number of change notices submitted (there were a total of 167) were for ambiguities in the requirements document. People had not written concisely about features, initially. The change notice process was used to remedy these problems.

One of the sources of ambiguity stemmed from descriptions of changes that were out of context from the description that was in the requirements document. We were able to avoid these problems by having submissions show the actual change that would be made to the requirements document.

The Results

There were dramatically fewer changes on this project than previous projects we had worked on. Whether this was from more up front work in the definition or because the process eliminated unnecessary changes is unknown. The few large changes that were made were received without hard feelings and were accurately scheduled and applied. The amount of rework due to mis–communication seemed to be relatively low.

When the software design was completed the project was rescheduled. The results were surprising; the schedule did not change.

The very first milestone was missed by a significant amount (FFF was missed by 3 months). The completion of the design, 3 beta releases and the final system release were all made within 1 week of the originally predicted date at FFF. The release of the product was made within 3 weeks of the date that was first set at the beginning of the project.

The people working on the team feel very strongly about the use of the requirements document and the process surrounding it. It was essential for the solid focus on customer needs and the group commitment, The document and the process allowed for communication of changes to be smoothly incorporated, scheduled and tracked throughout the project. This was an excellent way to tie the documentation throughout the project together and was a good contractual device with groups outside of the direct engineering effort to determine exactly what would be delivered to them.

Object Analysis and Design: Integrating Behavioral Approaches

Frank J. Armour
Principal - Object Methodologist
Best Practices Group
American Management Systems, Inc.
4050 Legato Road
Fairfax, Va 22033
Phone (703) 267 - 2206
FAX (703) 267 -2222
frank_armour@mail.amsinc.com

Abstract

This paper describes a process for integrating behavioral analysis and design approaches such as use cases and responsibility driven analysis to effectively discover and model objects.

Keywords: object-oriented, analysis, design, use cases, responsibility driven design, object behavioral analysis, methodology

Biographical: Frank Armour is a systems consultant with practical and academic experience applying object technology, graphical user interface design, multimedia, and client-server technology to system development efforts. He is currently American Management System's object technology methodologist and is a member of the AMS Corporate Technology Group, where he is responsible for the development of AMS's Life-Cycle Productivity System Methodology: Object Technology Development. He consults and provides guidance to project development teams on the application of object technology.

Frank is currently an Adjunct Professor of Information Systems and Software Engineering at George Mason University. He teaches graduate courses on Software Requirements Engineering and Prototyping.

Object Analysis and Design: Integrating Behavioral Approaches

Frank J. Armour
Principal
Best Practices Group
American Management Systems, Inc.
Phone (703) 267 - 2206
FAX (703) 267 -2222
frank_armour@mail.amsinc.com

Introduction

For over three years, American Management Systems (AMS) Inc. has been involved in applying object technology and other advanced technologies to the high quality, large scale business systems we build with our clients. To help guide these system development efforts, we have combined several commercial object methodologies, and have developed techniques to link these methodologies together. Last year, we developed a version of our Life cycle Productivity System Methodology (LPS-M) that specifically addresses system development using object technology. The methodology utilizes use cases [Jaco93] and responsibility driven design [Wirf90] as well as concepts from Object Behavior Analysis (OBA) [Rubi92] as its primary analysis and design approaches. AMS has been successfully using this combination of behavioral approaches on a sizable number of development efforts over the past several years.

This paper shares AMS's experiences using these behavioral approaches, describing their application and lessons learned. The paper starts with a definition and description of use cases and responsibility driven design, followed by our experiences combining these approaches. Modifications and additional techniques to supplement these approaches and enhance their effectiveness on large business system development are described. Finally, a process model integrating these approaches is documented.

Finding and modeling the objects

To perform successful object oriented analysis and design, analysts need to discover the objects, then model their responsibilities and relationships. Our experience has shown that an effective way to discover objects is in a behavioral context. When objects have been correctly identified early in the development process, factoring of responsibilities can proceed more smoothly, without unnecessary rework of designs. It facilitates a smooth, effective analysis and design process, helping to ensure a high quality system that meets user's needs. We've

found a process integrating use cases and responsibility driven analysis and design effective in meeting these quality objectives.

Use case modeling approach

When individuals use a system, they perform a behaviorally-related sequence of transactions in a dialogue with the system. A use case approach defines a system by identifying and modeling the external events it responses to. The two major constructs used in use case modeling are **Actors** and **Use cases.**

An actor is a user, a person or external system, who initiates system activity for the purpose of completing a business task. An actor represents a role fulfilled by a user interacting with the system and is not meant to portray a single individual or job title.

Use cases are a combination of English descriptions and/or graphical representations that specify the interaction which takes place between an actor and the system for the purpose of completing a single business event. A use case defines the complete course of the business event. System functionality is defined by the collection of all use cases. Exhibit 1 presents a use case of a customer submitting a membership application to a book club.

1.	Customer fills out a membership application and forwards it the book club.
2.	The membership application is reviewed and entered in the system.
3.	The customer's credit worthiness is determined by: 3.1. Asking the accounting system to determine if the customer was a previous member, and if so what was their credit history. 3.2. Requesting a copy the customer's credit report from the credit bureau
4.	Update the customer as a member
5.	Send the customer a new member information packet.

Exhibit 1. Membership application use case

Responsibility Analysis Approach

Responsibility Analysis [Wirf90] defines objects by identifying each object's responsibilities within the system. Responsibilities are object behaviors needed to provide system functionality. To carry out its responsibilities an object will need to interact with other objects. These interactions are called collaborations. Objects are modeled by defining their responsibilities and collaborations with other objects.

A Brainstorming technique, called **role-playing**, can be used to perform responsibility analysis. Role-playing is the acting out of object behavior to simulate the design for the purpose of refining and validating the object model. This technique helps define objects and their behavior, and models the interactions among objects and external users (e.g. users, external systems). Using this technique, a development team employs **Class Responsibility Collaborator (CRC)** cards that are used to document each object's name, its description, a list of its responsibilities, and a list of other objects it collaborates with. Exhibit 2 presents a sample CRC card for a purchase order.

Object Name:	Purchase Order
Description:	Form used by procurement officer to order new commodities, travel, etc.

Responsibilities	**Collaborators**
Update form data	Accounting line
Set vendor to selected vendor	Vendor
Validate accounting line	
Validate myself	

Attributes: Name
ID

Exhibit 2. Example CRC card

Integrating Use cases and Responsibility analysis

Use cases and responsibility analysis complement each other. Use cases describe how users will use the system and they act as a starting point for identifying objects. Responsibility analysis is applied to these results to refine the objects by defining their behaviors and interactions.

This section discusses a process for applying use cases and responsibility analysis in combination. Exhibit 3 outlines the process model for integrating use case modeling and responsibility analysis. Lessons learned from applying these methods as well as any modifications we made are also described. Since these experiences were from many different projects, examples from a composite case study are used to highlight the discussion: A Book Club Member Service System (BCMSS) that automates the processing of new membership applications, book orders, order shipment and billing. This type of system, with its customer service and billing components, is typical of many of the systems AMS builds.

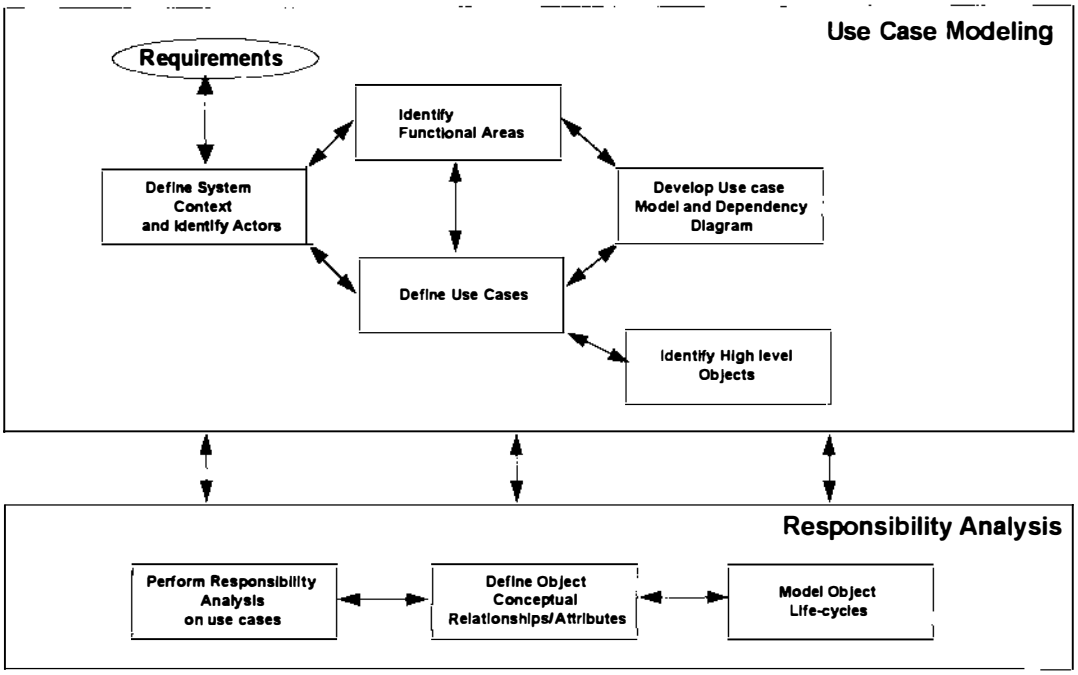


Exhibit 3. Integrated use case modeling - responsibility analysis process model

Applying use case modeling

Our early projects found that without some context for identifying and refined objects, it was unwieldy to model objects directly from a large text based requirements document. There are just too many nouns and verbs to effectively utilize syntactic dissection techniques [Abbo83], such as "underlying the nouns". We found that use cases provided a solution to this problem. By representing the functionality described in the requirement document through use cases, we found that we could effectively apply techniques such as "underlining the nouns" on the use cases.

We apply case use modeling, which describes the primary system functions from the perspective of external users, by identifying actions or events initiated by the users and documenting them from an external viewpoint. The use case model helps communicates system scope to both users and project sponsors.

The key steps we performed during use case modeling include:

- Preparation of a **Context Diagram** and identification of external **Actors**
- Partitioning of the system into **Functional Areas**.
- Identification of **Use Cases** initiated by each actor
- Partition of the collection of use cases into the **Functional Areas** and organizing them in a **Use Case Model**

- Development of a **Use Case Dependency** diagram

After the system context diagram is created and the external actors are identified, functional areas within the system are identified. For a large system, actors should be identified for each functional area. Any interaction between functional areas is represented by a use case, with the initiating functional area playing the actor role. Next, the primary use cases, which model high level interactions with the system without regard for exception conditions or alternative courses, are identified and defined.

Although use cases provide a intuitive and natural way to model the system, projects found it hard to get a "conceptual" grasp of the system when it was represented as an unordered set of use cases. Three issues that continually arose included:

- The partitioning of use cases into functional areas.
- The dependency of one use case on another.
- Modeling of non-behavioral requirements.

In our book club example, there is unique functionality associated with both member servicing and shipping. These are defined as two different functional areas. On a large system, partitioning of system behavior into functional areas is critical in understanding the system architecture and effectively defining a development strategy. To represent this partitioning, a use case model is developed, which shows the functional areas and the use cases they interact with, see Exhibit 4.

Although individual use cases are useful in exploring a single interaction with the system, what about the larger workflow perspective? For example, the workflow of ordering a book and receiving the book may encompass several use cases: the placing of order, the order's shipment, mailing the bill, and finally the customer making the payment. How do the use cases interrelate to support this perspective? Our projects noted that some use cases were dependent on other use cases. One use case leaves the system in a state that is a precondition for another use case. For example, a precondition of an order being shipped, is that the order is already been placed. The event modeled in the *place order* use case must occur before the event modeled by the *ship order* use case can occur.

The use case dependency perspective:

- Enhances understandability of system functionality and scope, by organizing system functionality into a life cycle of events and states.
- It provides increased traceability by helping to map use cases to any work flow models previously developed.

- Identities missing use cases. A use case with a precondition that is not met by the execution of another use case may indicate a missing use case.

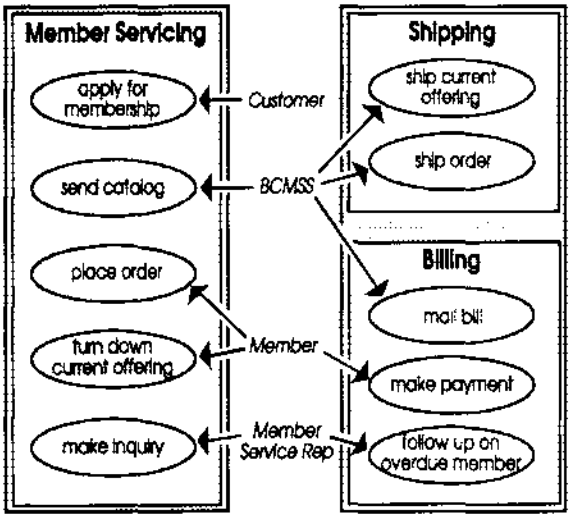


Exhibit 4: Example Use Case model

To represent a dependency view, we create a use case dependency diagram, see Exhibit 5 and have added pre and post conditions to each use case, see Exhibit 6.

We have also added an assumption section to each use case, for documenting any requirements, such as performance or security, associated with the interaction defined in the use case that cannot be naturally modeled as part of the use case course.

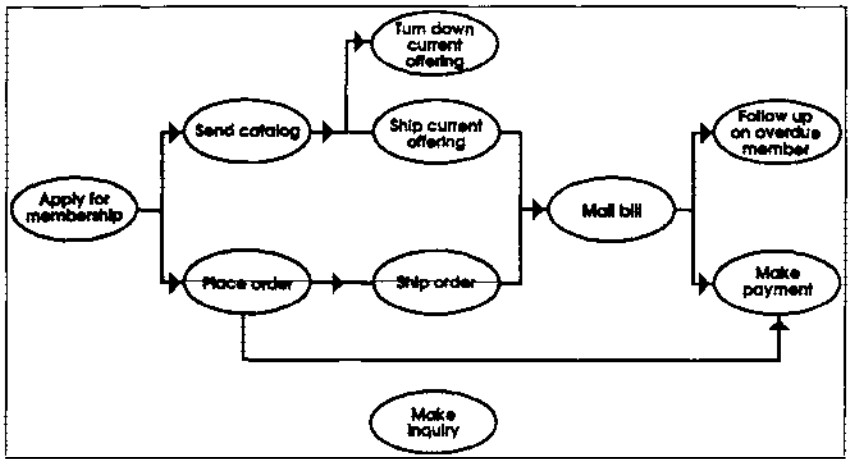


Exhibit 5: Dependency Diagram for Book Club

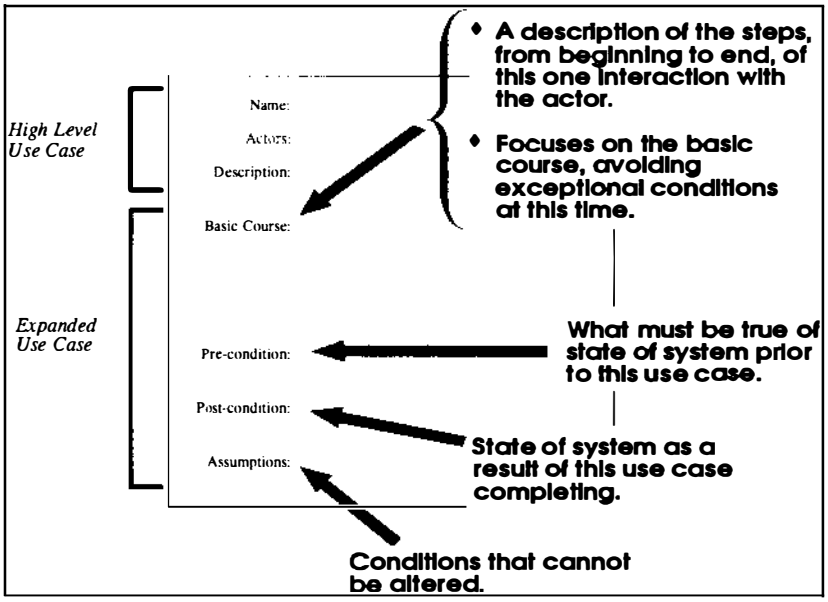


Exhibit 6: Use Case Format

After the primary use cases have been defined and organized, the next step is a "first cut" identification and modeling of objects found in the use cases. During this step the major objects which will make up the system as well as any conceptual relationships between the objects are modeled. The model describes objects that represent "things" in the Business domain. These objects could be as business events such as a billing cycle or "business entities" such customer, payroll form. At this level, concentrate on just describing the objects with a sentence or two.

To perform this step, review each use case and find the terms or nouns that represent business entities or events. When finished with all the use cases, review the results and fit them into the following categories:

- Tangible things
- Concepts
- Events
- Interactions
- Roles played

Then clean up this list of "candidate" objects by removing:

- Synonyms.
- Objects outside the scope of a use case. For example, external entities that are referenced in a use case but are not within the system boundary.
- Nouns that are really actions or attributes.

At this point we have a good conceptual picture of the system. We validate the system concept by reviewing it with the users.

The next step is the definition of alternative courses and exceptions conditions in the use cases. This concept is discussed in-depth in Jacobson [Jabo93].

Lessons Learned about Use Case Modeling

We have found the following issues surrounding use case modeling:

- Use cases assist in performing a quality analysis effort by providing,
    - — A very focused view of system behavior. Use cases get to the "heart" of matter, by describing in a very compact manner, behaviors expected of the system. Since the descriptions of system functionality are not spread through a large document, validation activity can be more focused.
    - — A very understandable format for communication and validation between users and developers. Not only do use cases provide a focused view of system behavior, this view is in a form, textual and high level graphics, which is easy for users to understand and therefore validate. In fact, many of our users have stated that use cases "speak their language".
    - — A starting point to help identify objects and their high level relationships and responsibilities. Several object methodologies suggest that the way to discover objects in a requirements document is to go through the document and underline the nouns. We have found that this techniques does not scale up very well to large projects, where a concept or requirements document can be hundreds of pages long. There may be thousands of nouns, with most of them being discarded after analysis. Use cases provide a means of "distilling" the document down, so that techniques such as underlying nouns can be utilized more effectively.
    - — Finally, use cases can be used to generate testing scenarios and documentation.
- Use cases need to be organized into frameworks, such as functional areas and dependency life cycles, that provide a global, integrated view of system.

Applying responsibility analysis

The events identified during use case modeling are now used as a context to perform responsibility analysis. A **responsibility** is high level behavior associated with an object. For example, a customer object might have a responsibility to "Determine if it was a previous

member" of the book club. Using the objects defined during use case modeling as a starting point, define each object's responsibilities by modeling the interactions between those objects needed to accomplish business events specified in the use cases. Interactions between objects occur when one object requests another's responsibility. For example, a Customer object would request a Credit Criteria object to "Determine credit worthiness", so that the Customer could be approved for membership in the book club. Walking through or role-playing each use case provides a context for identifying collaborations and assigning responsibilities. During this process you will probably discover new objects, further define existing ones, and discarding some others. At the end of this step you will have an object model describing the objects, along with their responsibilities and interactions between them needed to perform the system's functionality as reflected by the use cases.

More formally the process proceeds as follows: The initial objects, found in the use cases, are written down on object description forms. These forms are similar to CRC cards but include several other fields including object attributes and traceability links to use cases. Each group member, participating in the role-playing session, takes an object description form. The group walks though one use case at a time, simulating the collaborations between the objects. Each individual plays the role of an object in carrying out its behavior in the context of the use case. For example, the group would simulate the interaction among objects in order to process a new book club membership. One person would play the role of a Membership Application object, another would be a Customer object, etc. As the role-playing progresses, the objects' responsibilities and collaborations become clear. Object responsibilities are recorded and refined, collaborators are defined for each object, and new objects are created to complete the design. Exhibit 7 graphically describes this process.

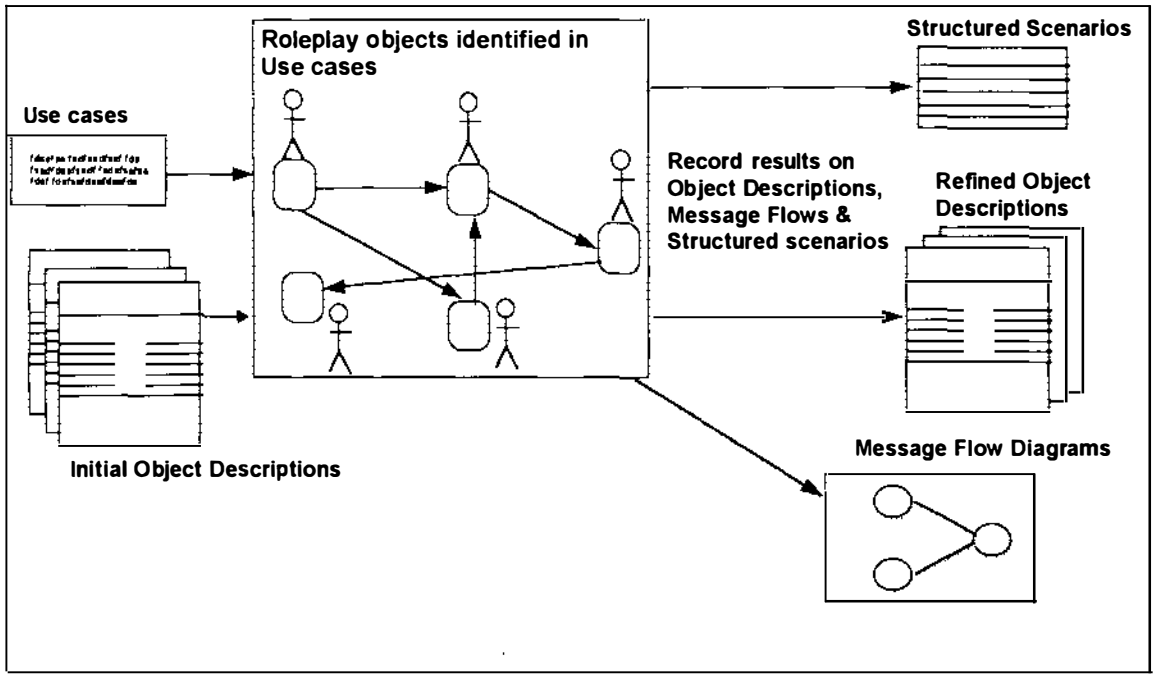


Exhibit 7: Role-playing using use cases

The result of the process is a set of object descriptions captured on the object description forms, a set of structured scenarios (See Exhibit 8) and message flow diagrams (See exhibit 9) describing the object interactions. A structured scenario describes the interactions between objects that carry out the behavior defined in the use case. It records what messages objects send and receive, the action requested in each message, and the order in which these message interactions occur. Each message is defined on one line of the scenario. Objects are either senders or receivers of a message. The message request is described in the Request/Action field and the result (any information sent back from the server object to the client object) is recorded in the result field. Groups liked to diagram the object interactions using message flows on a white board as they role-played the use case. Message flows diagrams capture the collaboration patterns between objects. After the role-playing was complete, the results were transferred to structured scenarios. The objects and their relationships can now be documented in a model.

Structured scenarios are similar to scenarios as described by Rubin and Goldberg [Rubi92] in Object Behavior Analysis (OBA). In OBA they are the derived directly from high level requirements, we deviated from this approach and defined use cases first. We found the combination of free form use cases and structured scenarios provided a fluid transition from unstructured requirements to an a object model describing the objects needed to implement the requirements. OBA also stresses the identification and modeling of the "roles" that are played within the system. We found this perspective extremely useful when identifying objects and defining object behavior.

Name:	Process New Membership
Description:	A customer applies for membership to the book club. A check is done on the person's/organization's credit worthiness. A new member packet is sent to the approved person/organization.
Preconditions:	The membership application has been approved and a new membership packet has been sent to the new member
Post Conditions:	N/A

	Sender (Client)	Receiver (Server)	Request/Action	Result
1	User	Membership Application	Check Application	Approved/Application
2	Membership Application	Customer	Approve Customer	
	Customer	Member	Check for prior Membership	Prior membership status
	Customer	Credit Criteria	Determine Credit Worthiness	Credit rating
	Credit Criteria	Accounting System	Get Member Credit	Credit history with company
	Credit Criteria	Credit Bureau	Get Customer Credit	Credit Rating
	Membership Application	Member	Sign up member	
3	Membership Application	New Member Packet	Mail to Member	

Exhibit 8. Structured scenario describing the processing of new memberships

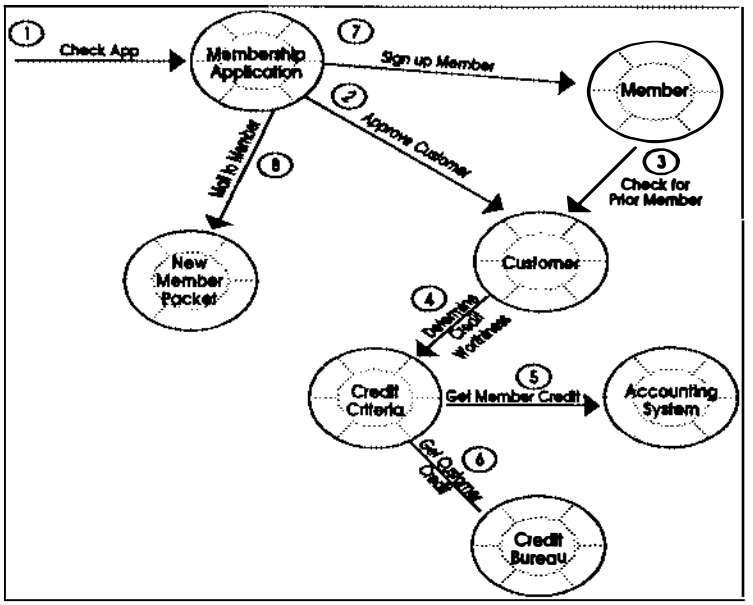


Exhibit 9. Example of a message flow diagram

Once responsibility analysis has been performed other conceptual relationships, see Exhibit 10, in addition to object interactions are defined, these relationships include:

- Associations between objects (including cardinalities) and object attributes
- Generalization/specialization relationships between business objects (e.g. Inheritance or IS-A relationships). At this time, capture any generalization relationships that help you understand the business problem.
- Aggregation relationships between business objects (e.g. object A is part of Object B)

We found the rich notation defined by Rumbaugh [Rumb91] and Booch [Booc94] effective in modeling these relationships. These conceptual relations are especially important if you are deriving a logical data base for a non-object database (e.g. relational). We also find it useful to model the object life cycles of objects that are persistent and/or change state frequently. Shlaer and Mellor [Schl92] provide a very extensive set of techniques and notations for modeling object life cycles.

Although responsibility analysis is performed first, the tasks described above are not meant to be applied in a linear or waterfall manner. Projects iterated several times over the above steps, since it is very difficult to get the object model "right" the first time. For example, the discovery of an aggregation relationship between two objects will impact the distribution of responsibilities between the objects.

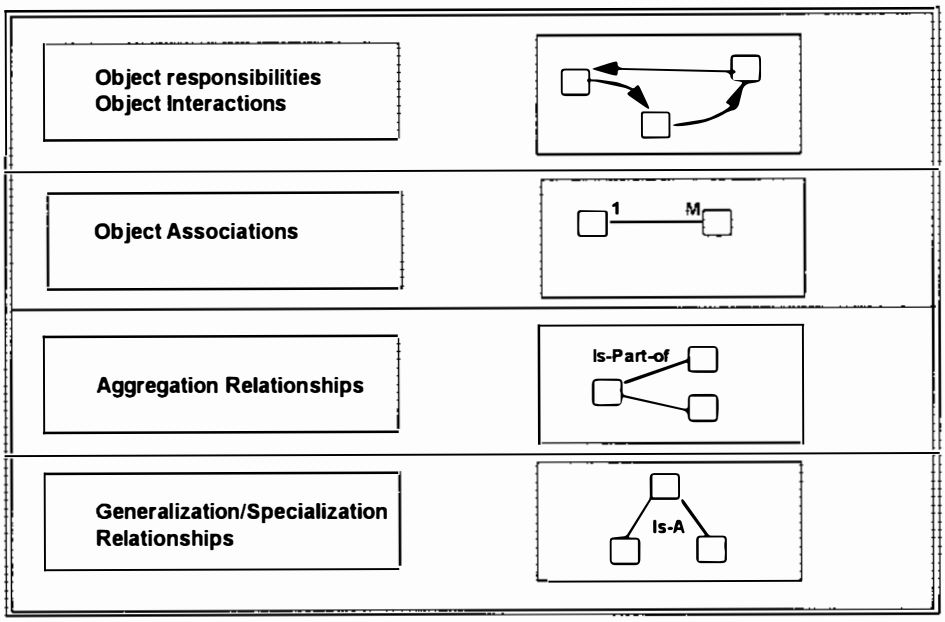


Exhibit 10. Object concepts and relationships defined

Lessons Learned about responsibility analysis

We found the following issues surrounding responsibility analysis:

- It helps protect the encapsulation of an object's internal details. Since the transformation between the levels of object abstraction is very seamless, it is very easy for designers to slip into a tendency to describe objects at greater detail than appropriate during analysis and high level design. Responsibility analysis helps prevent this definition of internal detail too early.
- It forces designers to "think" in the object paradigm. One of the hardest things for individuals with data modeling background to overcome is viewing objects as just data with behaviors assigned to them. By stressing object behavior, this technique helps individuals make a faster shift to object development.
- When designers have little object experience, formal role-playing rules should be adhered to. As designers gain experience, this technique can be relaxed.
- Use cases provide a natural context from which to role-play the objects, providing an event and a set of associated actions to model. Without this context it is harder to "know what to role-play".
- There should be a manageable number of objects involved in any role-playing. We have found that role-playing "breaks down" if the number of objects in any one role-playing session gets to large, normally 7 to 10 is the upper limit. For large systems we tend to use the technique early in analysis and design, when the number of objects is generally small. Role-playing one use case at a time helps manage the number of objects involved in any one role-playing session.

- Techniques such as structured scenarios and message flow diagrams increase the effectiveness of role-playing sessions as well as serving as a documentation tool. Without these techniques, the specific collaboration patterns between objects is difficult to remember both during the session as well as later during development.

- Thinking in terms of the role(s) that objects play helped us more effectively identify objects and define their behavior during responsibility analysis.

Conclusions

Our projects have found the application of use case modeling, responsibility driven analysis and concepts from OBA effective in discovering objects and defining their responsibilities and relationships, resulting in a quality object analysis and design. By synthesizing these techniques and enhancing them with other analysis and design techniques, as well as documenting a process model for applying them, we feel we have developed a practical, robust object analysis and design methodology that serves as a guide for object technology based business system development.

References

Abbo83 Abbott, R, ,"Program Design by Informal English Descriptions", CACM, August 1983.

Jaco93 I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, Object-Oriented Software Engineering: A Use Case Driven Approach, New York: Addison-Wesley, 1993.

Booc93 G. Booch, Object-Oriented Analysis and Design, Second Edition, Redwood City: Benjamin/Cummings, 1994.

Rubi92 K. Rubin and A. Goldberg, "Object Behavior Analysis," CACM, September, 1992 pp. 48 - 62.

Rumb91 J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-Oriented Modeling and Design, Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

Schl92 S. Shlaer and S Mellor, Object Lifecycles: Modeling the World in States, Englewood Cliffs, New Jersey: Yourdon Press, 1992.

Wirf90R. Wirfs-Brock, B. Wilkerson, and L. Wiener, Designing Object-Oriented Software, Englewood Cliffs, NY: Prentice-Hall, 1990

Empirical Evaluation of Complexity Metrics For Object-Oriented Programs

Michael W. Cohn
and
William S. Junk

3700 Ocaso Court
Cameron Park, CA 95682
70324,3535@compuserve.com

University of Idaho
Department of Computer Science
Moscow, Idaho 83844
billjunk@cs.uidaho.edu

Abstract

The complexity inherent in object-oriented programs cannot be fully measured by traditional metrics such as the Halstead and McCabe measures. Object-oriented concepts such as inheritance and polymorphism affect complexity. This paper presents fourteen metrics intended to measure aspects of object-oriented complexity. Statistical tests of significance against defects found during code inspection are presented.

Keywords and Phrases: Metrics; Object-Oriented; C++; Halstead; McCabe; Complexity

Biographical:

Michael W. Cohn is the Vice President of Engineering at Telephone Response Technologies, Inc. in Roseville, CA and has a Master's degree in computer science from the University of Idaho. His interests are object-oriented programming, project management, the use of formal methods with object-oriented languages, and computer telephony.

William S. Junk has been a faculty member in the Computer Science Department at the University of Idaho since 1980. Prior to joing the University he spent 12 years with the General Electric Co, Space Division, developing hardware and software for ground-based spacecraft test systems and performing project management on large command and control systems. His research interests focus on topics in the areas of software quality assurance, software process management, software project management, and technology transfer.

1 INTRODUCTION

As has been pointed out in prior research, the metrics used to measure the complexity of programs written in traditional, procedural languages do not adequately measure the complexity encountered in object-oriented programs (see [Chid91], [Chid94], [Kole93], [Lake92], [Morr89] and [Whit92]). By themselves, these metrics cannot be used as reliable indicators of defect-prone objects or classes. Many traditional metrics, such as McCabe's cyclomatic complexity, focus on control flow complexity. Control flow complexity, however, is not the only type of complexity present in programs written in object-oriented languages.

Typical object-oriented programs contain functions whose sizes are smaller than would be seen in their functional counterparts and measurements of complexity for programs of this type should take into account considerations other than control flow complexity. Because most object-oriented programming languages contain control flow statements, it is not appropriate to abandon all these measures of complexity. Therefore, traditional complexity metrics may still be of some use. However, in addition to control flow complexity, object-oriented languages possess what can be referred to as *type complexity*. Type complexity is that portion of complexity in a system that is attributable to the manner in which data types have been expressed in the system. Because object-oriented languages provide a more expressive set of language constructs for expressing type information than do traditional, procedural languages the type complexity in object-oriented programs will tend to exceed that found in procedural languages. This is, of course, the method through which object-oriented languages are able to reduce control flow complexity. There is a tradeoff between control flow complexity and type complexity.

In order to accurately determine the complexity of source code written in an object-oriented language, it is important to take both control flow complexity and type complexity into consideration. Since control flow complexity can already be measured with existing metrics, the goal of this research was to identify a set of metrics intended for use with object-oriented programs that can be used to measure the type complexity of those systems. Such metrics can then be used to identify defect-prone classes or object-oriented practices that increase the likelihood of inserting a defect into a program.

1.1 Class Level Metrics

One of the features that makes an object-oriented language object-oriented is the ability of the language to combine the code and data used to model a real-world entity into a single structure. These structures are sometimes known as objects or classes. In this paper, *class* will refer to a type of structure and *object* will refer to a specific instance of a class. For example, if *Automobile* is a class then the 1967 Volkswagen in my garage is an object. Because much of the work in creating an object-oriented design is related to the correct partitioning of a system into the appropriates classes ([Booc86], [Coad90], [Booc91], [Rumb91]), efforts at calculating type complexity are correctly targeted at the class level.

The decision to define metrics measurable at the class level is based on three contributing factors:

- **Encapsulation:** One of the cited advantages of object-oriented design is the ability to encapsulate implementation details into a class.[Booc91][Rumb91] This occurs because the data and functions necessary to implement the data structure are both contained in the class definition. If everything necessary for implementing a data structure has been encapsulated into a class, it seems logical to conclude that whatever complexity is inherent in that data structure has also been encapsulated into the class definition.
- **Code Reuse:** One of the argued advantages of using an object-oriented language is that it facilitates code reuse. Code reuse can occur more naturally and more simply than in non-object-oriented languages because all aspects of a class are tied together in the class concept. Classes cannot be partially reused, it is generally an all or nothing proposition. Because reuse occurs at the class level it will be useful to have as a goal a metric that can be applied to individual classes.
- **Classes are Types:** Because the primary mechanism for defining a new data type is the definition of a new class, it makes sense that a metric targeted at measuring type complexity should occur at the class level.

Not all of the metrics discussed here are class level metrics. The metric *Inheritance Tree Size* is calculated at the system level.

1.2 Scope of Study

It was considered important to propose a set of metrics and then to offer some form of empirical justification for them. Statistical tests of significance on the correlation between each metric and the number of defects discovered during code inspections were performed in order to ascertain the extent of the relationship. Only those defects found during inspection that would be manifested in the runtime operation of the program were counted. As empirical evidence was desired, an object-oriented language had to be selected for which it would be possible to gather the necessary metrics. The C++ programming language was selected for use in this research for the following reasons:

- it is the dominant object-oriented language in use today. Plauger remarks, "whatever object-oriented language is second in popularity is a very distant second to C++."[Plau93, p. 16]
- it will probably continue to be the dominant object-oriented language at least through the end of the century[Plau93]
- it was possible to access a body of C++ code and code inspection defect data

It was then possible to collect the data necessary to perform the statistical testing. For the defect quantities, data was available from records kept during code inspections held over a two-year period. Most metrics which were calculated from C++ source code were generated using the PC-METRIC tool from SET Laboratories. After all of the defect data and metrics were collected, the correlation coefficients were calculated and tested for significance using standard statistical procedures.

1.3 Importance

Software development and maintenance companies are continually in search of productivity-enhancing changes to their software life cycles. This quest for increased productivity is what lead to the creation of languages in the first place and what is causing some companies today to move toward the use of object-oriented programming languages. According to Jones [Jone91], using a typical third-generation language it takes an average of 91 logical source statements to implement each function point in a program. The same study indicates that only 29 logical source statements are needed per function point using a typical object-oriented language. Although Jones himself admits that this data has a "very large margin of error" [Jone91, p.77] the implication that object-oriented languages are more expressive is clear. This provides strong incentive for software managers to contemplate switching to the use of object-oriented languages in pursuit of increased productivity.

Unfortunately, once the level of productivity possible with object-oriented languages becomes the norm and is expected of well-managed projects, software managers will have to look elsewhere in the perpetual search for increased productivity. One fertile spot in which to search for increased productivity will be the use of complexity metrics to identify classes that pose a high risk for defects. If such classes can be identified then it is feasible that high-risk classes can either be redesigned to remove some of the risk; or, failing that, additional code review or testing time can be planned.

Building upon the foundation laid in prior work by Lake and Cook[Lake92], Whitmire[Whit92] and Chidamber and Kemerer[Chid91],[Chid94], this research will hopefully aid in the identification of high defect risk, complex classes. Using the metrics identified in this paper as possessing a statistically significant correlation when compared against discovered defect densities, a software manager may be able to adjust code review or test schedules to emphasize those classes that are overly complex. The knowledge of which classes may be particularly prone to defects can then be used by project management to more accurately position the project between the potentially opposite goals of fewer defects and earlier delivery.

2 Problems With Existing Metrics

Traditional complexity metrics, such as Halstead's Software Science and McCabe's cyclomatic complexity, do not adequately capture all of the complexity of object-oriented systems. Object-oriented languages utilize a number of extensions to purely procedural languages that result in an additional kind of complexity.

As an example, consider two different class implementations of the same concrete, real-world entity. Each class has four member functions and four member variables. Further, the source code for each class implementation is identical. Given such identical classes it is not surprising that the Halstead and McCabe measures for each class would be the same. To show one possible effect of an object-oriented language on complexity, define the class *Simple* to have all public members and the class *MaybeSimpler* to have two public and two protected members. Is it reasonable to argue that the complexity of these two classes is still equal? The class *MaybeSimpler* presents a restricted interface to the rest of the system; therefore, using it could be

conceptually less complex. On the other hand, in a system that needs to derive a class from *MaybeSimpler* in order to have access to the protected functions, this may actually be more complicated than using the all-public *Simple* class would have been.

Similarly, the effects of inheritance on complexity are not captured by traditional metrics. Because the use of inheritance will spread complexity among the classes in the inheritance tree, it will cause metrics that look only at a single class or function not to detect one aspect of a class' complexity: the complexity inherited from its superclasses. For example, a *Btree* class may inherit properties from a *File* class that is responsible for all file reading and writing. The problem with applying procedural metrics to object-orieted languages is that it is not clear at which level they should be applied. Should procedural metrics be applied at the function level? At the class level? Or to an entire inheritance tree? To what extent is the complexity of the *Btree* class influenced by the design of the *File* class? The inclusion of inheritance in object-oriented programs introduces an additional type of complexity into object-oriented programs. In order to measure this additional type of complexity it will be necessary to identify a new set of metrics. Metrics that capture this additional type of complexity can then be used in conjunction with traditional metrics of control flow complexity in order to present a more accurate measure of the true complexity.

3 A Proposed Set of Object-Oriented Metrics

This section identifies the set of fourteen metrics which are evaluated in this paper. As each metric is introduced the method for calculating it is described.

3.1 Inheritance Tree Depth

Increasing the depth of the inheritance tree is likely to add complexity to classes because classes that are deeper in an inheritance hierarchy usually receive a relatively greater proportion of their functionality from their superclasses. As the depth of an inheritance tree increases, the burden placed on programmers to remember and understand the interaction between members of the hierarchy can become significant. Simple acts such as determining which class will process a received message can become complicated issues in deep inheritance trees. This metric is one of the metrics proposed by Chidamber and Kemerer [Chid91].

Method of Calculation: This metric is calculated by counting the number of levels in the inheritance tree beginning at the top of the tree and descending to the class being measured. Classes derived from the class being measured are not counted.

3.2 Inheritance Tree Size

The size of an inheritance tree measures the tree's breadth as well as its depth. Wide inheritance trees are indicative of class relationships that go beyond simple hierarchical inheritance. The width of an inheritance tree can increase for one of two reasons: either two or more classes are derived from a common superclass or multiple inheritance has been used.

In situations where multiple inheritance has been used it is likely that class complexity will increase because of the likelihood that there will be subtle but important interactions between the inherited functionalities of the two superclasses. Where multiple classes are derived

from a common superclass the positive or negative impact on complexity will be determined by the original implementor's correct partitioning of member functions and data between the classes.

Method of Calculation: This metric is calculated by counting the total number of classes in the inheritance tree.

3.3 Weighted Methods per Class

When object-oriented source code is viewed at the function or procedure level instead of at the class level then there may be great reason to continue using established, traditional metrics such as McCabe's cyclomatic complexity or Halstead's Difficulty measure. Traditional, procedural metrics remain valuable metrics of complexity. Kolewe remarks that

> The Halstead metrics may be just as useful in the world of objects as they are in their original realm. However, the traditional metrics tend to concentrate on the procedural aspects of code, and object-oriented software involves more than procedure. [Kole93, p.56]

This metric is identical to the metric of the same name proposed by Chidamber and Kemerer [Chid91]. Each method in the class is weighted by its cyclomatic complexity.

Method of Calculation: This metric is calculated by summing the individual cyclomatic complexities for all member functions in the class being measured.

3.4 Halstead Class Difficulty

This metric, along with the *Weighted Methods per Class* metric, is intended to capture that portion of a class' overall complexity that is measurable by more traditional metrics. Halstead [Hals77] proposed a difficulty measure based on n_1 (the number of unique operators in a program), N_2 (the total number of operands), and n_2 (the number of unique operands). Halstead defined his Difficulty measure by combining these as follows:

$$D = (n_1 / 2) \bullet (N_2 / n_2)$$

Halstead's Difficulty measure is intended to measure the pyschological complexity of writing a section of code and is therefore a viable measure of the complexity of the methods contained within a class.

Method of Calculation: This metric is calculated by summing the individual Difficulty measures for all member functions in the class being measured.

3.5 Number of Classes Used

This metric measures the actual number of classes that are referenced from a given class. The logic behind the inclusion of this metric is that as the number of classes used by a class increases, the complexity of the calling class is likely to increase. Although this metric may seem similar to simply counting the number of lines of code in a class, it can be argued that a line of code that sends a message to a class is not as likely to be free of hidden complexity as is a line that calls a function in the C standard library. Because classes hide the implementation details of

the class from users of the class there is always the inherent risk that a class will also hide important assumptions about how the class should be used. Encapsulation and abstraction are double-edged in that while desirably hiding implementation details they may also inadvertently hide important assumptions. This metric is similar to the *Response for a Class* metric proposed by Chidamber and Kemerer [Chid91].

Method of Calculation: This metric is calculated by counting the number of classes with functions invoked by code in the class being measured.

3.6 Deepest Class Used

Just as complexity is likely to increase in relation to the number of actual classes referenced by a class, it is worth investigating the possibility that class complexity increases based upon the depth of the classes it references. Calling a function in a class that is deeply nested in an inheritance tree theoretically should be no more complicated than calling a function in a class at the top of an inheritance tree. However, in practice, it is possible that the likelihood of inserting a defect into the code will increase. When a programmer issues a call to a function in a class located in a deep inheritance tree he or she must trace through the inheritance tree to be convinced that the function will be handled by the appropriate class within the inheritance tree. The use of automated design tools such as class browsers can help minimize the impact of inheritance tree depth but cannot eliminate it.

Method of Calculation: This metric is calculated by determining the depth of all classes containing functions called by the class being measured. Its value is the inheritance tree depth of the deepest called class.

3.7 Number of Public Class Members

Public class members are visible outside the scope of the class. They are, in effect, the external interface to the class. This metric represents a count of the public class members, both public member functions and data variables, in a given class. As the number of interfaces into a class increases, the number of ways in which that class may be misused increases. This metric may be thought of as a measure of the coupling between the class and the rest of a system.

Method of Calculation: This variable is calculated by summing the number of public member functions and public member variables.

3.8 Number of Protected Class Members

In C++, protected class members are indicative of the class designer's intent in regard to the inheritability of the class. In C++, inheritance is generally planned for by making member data and functions protected if access to them is to be allowed in subclasses.

Method of Calculation: This variable is calculated by summing the number of protected member functions and protected member variables.

3.9 Number of Private Class Members

Private class members are restricted in scope so that they are accessible only to the class itself. As such, the number of private class members may indicate the amount of complexity which has been encapsulated within the class.

Method of Calculation: This variable is calculated by summing the number of private member functions and private member variables.

3.10 Number of Member Functions

It is reasonable to postulate that as the number of member functions contained within a class increases, the complexity of that class increases. The more member functions contained within a class, the more messages the class is capable of responding to. Classes that contain a large number of member functions may either be justifiably complex or they may be indicative of a system that has been poorly partitioned and would benefit from the inclusion of additional classes.

Method of Calculation: This variable is calculated by summing the total number of member functions in the class being measured. This will include all public, protected and private member functions.

3.11 Number of Member Data Variables

Member data variables are used to store information about the state or context of a particular instantiation of a class. As more data is stored within the class it is reasonable to postulate that the class becomes more complex. Additionally, classes containing a large quantity of data variables are likely to exist only in complicated systems or in systems that were incorrectly decomposed into classes.

Method of Calculation: This variable is calculated by summing the total number of member data variables in the class being measured. This will include all public, protected and private variables.

3.12 Number of Friend Functions and Classes

In C++, private data variables are accessible only by the class' member functions. However, there is a mechanism for providing access to private data through the use of the *friend* keyword. By using this keyword, access to a class' private data variables can be selectively given to other classes or functions. The granting of friend privileges to other classes or specific functions is usually indicative of a high-level of coupling between the two classes. Therefore, this metric is included as a measure of the degree of interclass coupling present.

Method of Calculation: This metric is calculated by summing the total number of friend declarations in a class. Equal weight is given to the declaration regardless of whether friend access is given to a class or a function.

3.13 Number of Overloaded Functions and Operators

Overloaded functions are functions that have the same name but are called in different circumstances. In C++, this is accomplished by using the parameter lists passed to the functions.

Overloaded operators are similar to overloaded functions except they refer to overloading the operators built into the C++ language. The ability to overload operators such as +, -, >, <, +=, etc., is a very powerful aspect of the language; however, it frequently adds complexity to C++ programs because there is no language enforcement of what the operators can be overloaded into doing by a class implementor.

As the number of overloaded functions in a class increases, the user of the class will be required to understand the class inheritance tree in greater detail. Similarly, the overloading of operators is a C++ feature that easily can be abused and can therefore add to overall class complexity.

Method of Calculation: This metric is calculated by summing the number of overloaded operators and overloaded functions in the class being measured into a single value for each class.

3.14 Number of Static Class Members

In C++ it is possible to declare static variables and functions. Static class members, whether variables or functions, are not members of a particular instance of the class; instead, there is a single copy of the member that is shared by all instances of a class. In effect, static members are the C++ equivalent of global variables with the advantage that static members are local to the class in which they are declared.

Method of Calculation: This metric is calculated by summing the number of static member functions and static data declarations in the class being measured.

4 Validation Of The Proposed Metrics

The process for validating these metrics will be to compare each of the proposed metrics against the number of discovered defects in a set of C++ source code used in the development of a commercial software system. Defect counts are the result of code inspections rather than testing. Using the number of discovered defects as a proxy for complexity, the assumption is made that the number of discovered defects will be the greatest in those classes that are the most complex. The correlation between each metric and the discovered defects will be calculated and the significance of the correlations will be tested using standard statistical procedures.

4.1 Statistical Process

Source code comprising seventy-two C++ classes drawn from eight related projects was analyzed. The classes involved in this study were all drawn from a series of programs that, as a whole, function to provide a high-level of control over telephony and interactive voice response systems (e.g., "Press one for this; press two for that..."). Individual classes were related to a wide variety of the components in the overall system including digitial switching, serial communications, database access, tone detection and generation, and the management of distributed processes. The entire system, much of which was in existence prior to the creation of the classes in this study consists of approximately 64,000 logical source statements while the classes discussed here accounted for only 3,422 logical source statements, or just over 5%. The bulk of the code not considered here was either pre-existing or was implemented in C.

The C++ code was produced over a two-year period by four programmers. Although there are a handful of exceptions, classes were generally worked on, from inception through design and coding, by a single developer. When the developer responsible for implementing a class felt that the class was functional and had passed its preliminary unit tests, the code for that class was distributed to the other development team members for their review. Code review participants were expected to spend approximately an hour reviewing a typically-sized piece of source code (between 250 and 500 logical source statements). A code review meeting was then held a few days after the code was distributed to ensure sufficient time for all parties to thoroughly read the code. At the code review, the code was discussed and all attendees had the opportunity to identify any defects they had discovered. A count of defects by type was kept.

During the code review process the author of the code kept track of changes that were suggested by reviewers. The author was then expected to correct any deficiencies after the review was adjourned. The final step in each code inspection was to determine whether the code had passed the review or whether the extent of the suggested changes warranted an additional review. If any class was inspected more than once, its defect total was the sum of unique defects discovered. It is assumed that differences in the abilities of the programmers did not significantly affect the results. This is based on the fact that the number of years of experience with C++ was roughly equivalent among the four programmers (ranging from three to five years).

To validate the metrics the correlation between the number of defects discovered during code inspections and each of the metrics was calculated. Similar approaches are common in software metrics research ([Curt79], [Schn79], [Kafu87], [Henr81]).The Pearson product-moment correlation coefficient, usually referred to as simply the correlation coefficient, r, was calculated for each metric. Similarly, the coefficient of determination, r^2, was calculated in order to measure the proportion of variation in the number of defects explained by each metric.

Beyond simply calculating correlation coefficients and coefficients of determination it is necessary to test the significance of the results. The standard method for testing the significance of correlation coefficients for a sample size greater than fifty (as in this case where seventy-two classes were observed) is to perform a z-test. For the testing of each individual metric the following hypotheses are postulated:

H_0: The correlation coefficient, r, in the population is less than or equal to zero.
H_1: r in the population is greater than zero.[1]

[1] For the metrics, *Number of Private Class Members* and *Number of Overloaded Functions and Operators* (the only metrics for which the correlation coefficients were negative) the following hypotheses apply:

H_0: The correlation coefficient, r, in the population is *greater than or equal* to zero.
H_1: r in the population is *less than* zero.

However, since the z-test is for a difference from zero the rest of this discussion applies.

These hypotheses can be used to test the direction of the corrleation between each specific metric and the number of discovered defects. If H_0 is rejected for a given metric then it has been shown that there exists a positive correlation between that metric and the number of defects. For example, if H_0 is rejected for the *Inheritance Tree Depth* metric then it can be said that there is a significant positive correlation between tree depth and number of discovered defects.

Using these hypotheses a one-tailed test can be performed. Therefore, using a 0.01 level of significance, if the calculated *z*-value for any metric is less than or equal to 2.33 (determined by lookup in a table of areas under a normally distributed curve) then the hypothesis H_0 is accepted. However, if the calculated *z*-value for any metric is greater than 2.33 then H_0 may be rejected at the 0.01 level of significance. The 0.01 level of significance was chosen as the most rigorous of the commonly applied levels. Other frequently applied levels of significance are 0.05 and 0.10. At these levels of significance the calculated *z* values for rejecting H_0 would need to exceed 1.65 and 1.29, respectively. The results of these calculations are summarized in Table 1.

Metric	Correlation Coefficient (r)	Coefficient of Determination (r^2)	Z Value	Level of Significance
Inheritance Tree Depth	0.2729	0.0745	2.2994	0.05
Inheritance Tree Size	0.2543	0.0647	2.1431	0.05
Weighted Methods per Class	0.1881	0.0354	1.5849	0.10
Number of Classes Used	0.5389	0.2904	4.5407	0.01
Deepest Class Used	0.3897	0.1518	3.2834	0.01
Number of Public Class Members	0.0514	0.0026	0.4331	
Number of Protected Class Members	0.4042	0.1634	3.4062	0.01
Number of Private Class Members	-0.0692	0.0048	-0.5832	
Number of Member Functions	0.3225	0.1040	2.7175	0.01
Number of Member Data Variables	0.0262	0.0007	0.2210	
Number of Friend Functions and Classes	0.0505	0.0026	0.4256	
Number of Overloaded Functions and Operators	-0.1169	0.0137	-0.9854	
Number of Static Class Members	0.1161	0.0135	0.9785	
Halstead Class Difficulty	0.4185	0.1752	3.5265	0.01

Table 1. Summary of Findings

4.2 Analysis of the Results

Of the fourteen proposed metrics, four measured characteristics of the inheritance tree: *Inheritance Tree Depth, Inheritance Tree Size, Number of Classes Used* and *Deepest Class Used*. Each of these metrics tested significant at either the 0.05 or 0.01 level. This supports the claim that "the depth and size of the inheritance tree are major contributors to complexity"[Lake92, p. 1]. In conjunction with Lake and Cook's prior work[Lake92] testing the effects of changes in the inheritance tree on the ability of programmers to perform selected tasks, these four metrics add to the evidence that the size and shape of the inheritance tree must be considered in any object-oriented metric set.

In addition to the metrics which reflect attributes of a class' position within, and use of, the system's inheritance tree there were also two metrics that measure aspects of the class' internal construction and tested significant. Both *Number of Protected Class Members* and *Number of Member Functions* were siginificant at the 0.01 level. The *Number of Member Functions* metric is highly dependent upon the class designer's partitioning of services among the classes in a system. If the system is inadequately decomposed there may be too few classes, each of which is called upon to perform too wide a variety of services. Such classes will include a larger than optimum quantity of member functions. Additionally, the number of member functions in a class will tend to increase toward the bottom of an inheritance tree. Because a subclass can inherit functions from one or more superclasses it has the possibility of overridding these functions, thereby creating more member functions for itself.

The *Number of Protected Class Members* metric probably tested significant at the 0.01 level because of its strong relationship with the size and shape of the inheritance tree. Although C++ developers are admonished to declare members as protected if they think the class will ever serve as a superclass for some other class[Stev90], subjective experience indicates this is not always the case. It is not uncommon for the initial implementation of a class to be created with certain members declared as private and for a subsequent revision of the class to change the declaration of some members to protected. Such changes, as well as the fact that protected members are more likely to appear in classes making greater use of inheritance, probably contributed to the significance of the *Number of Protected Class Members* metric. Classes that contain a large number of member functions may either be justifiably complex or they may be indicative of a system that has been poorly partitioned and would benefit from the inclusion of additional classes.

One of the ancillary goals of this research was to test the validity of a subset of the metrics proposed by Chidamber and Kemerer [Chid91]. Three metrics included here, *Inheritance Tree Depth, Weighted Methods per Class* and *Number of Classes Used,* are similar to metrics proposed by Chidamber and Kemerer. Each of these metrics proved significant at one of the tested levels of significance. *Weighted Methods per Class,* however, was significant only at the 0.10 level.

It is not surprising that *Halstead Class Difficulty* is more stongly correlated with defect numbers than is *Weighted Methods per Class* because the *Halstead Class Difficulty* metric has a built-in adjustment when used with object-oriented languages. Because Difficulty is determined

by counting the number of operators and operands used within a class, if a subclass uses a member variable that it has inherited from one of its superclasses this will be reflected in the calculation of Difficulty. No such adjustment is made for cyclomatic complexity (and therefore for *Weighted Methods per Class*) and the results here indicate that *Halstead Class Difficulty* is more indicative of complexity than is cyclomatic complexity.

The relatively low r^2 values indicate that no single metric in this set is able to explain a large measure of the defect counts. The largest r^2 value, 0.29, is associated with *Number of Classes Used* and indicates that this metric explains only 29% of the variation in the number of discovered defects. Such low r^2 values for the statistically significant metrics is likely to be an indicator that there are additional contributors to the number of defects which were not identified in this study.

4.3 Non-significant Results

Although none of the remaining metrics tested significant at even the 0.10 level of significance, some discussion of the measured results is warranted. It is important to remember that the correlation coefficients, r, for these metrics did not generate large enough z values to reject their respective hypothesis, H_0 . Because of this it cannot be assumed that the population correlation coefficients for these metrics is greater than zero.

The slight negative correlation indicated between *Number of Overloaded Functions and Operators* and discovered defects is indicative of the fact that overloaded functions and operators can have a mixed effect on complexity. When used in the right way and on the appropriate functions some amount of operator and function overloading can reduce complexity by increasing program readability. However, complexity can increase if overloading is used inappropriately or too extensively. For the sample data analyzed here it is most likely the case that operator and function overloading were usedly appropriately to achieve a slight decrease in class complexity.

Similar statements can also be made for the *Number of Private Class Members* metric which showed a slight negative correlation with discovered defects. Holding the functionality of a class constant, as the number of private class members increases the visibility of the class to the external world must decrease which should result in a decrease in complexity.

It is not surprising that the metrics, *Number of Static Class Members* and *Number of Friend Functions and Classes,* did not prove significant. These are C++ language features that are not present in most classes and will therefore be unlikely to test as significant in a study such as this one. Bill Curtis reports a similar finding in a study in which correlation coefficients are calculated using Halstead and McCabe metrics: "uniformity in the sizes of programs employed may have limited these results. The range of values assumed by complexity metrics may not have been sufficient to allow correlational tests."[Curt79, p. 301]

The metric, *Number of Member Data Variables*, is little more than a measure of class size and as such would not be expected to have a strong correlation with discovered defects. Although at the extremes it can be claimed that a class consisting of a single member data variable is

probably less complex than a class consisting of a large number of member variables, the middle ground is more open to the impact of other factors.

4.4 Correlations of Object-Oriented to Non-Object-Oriented Metrics

In order to determine if these metrics are capturing aspects of complexity other than what is captured by the use of McCabe or Halstead metrics alone, it is necessary to examine the correlation between the object-oriented metrics and the traditional metrics. Table 2 shows the correlation coefficients calculated between the object-oriented metrics (shown in the rows) and the more traditional metrics (shown in the second and third columns).

	Weighted Methods per Class	Halstead Class Difficulty
Inheritance Tree Depth	-0.0255	0.0261
Inheritance Tree Size	-0.0279	0.0328
Number of Classes Used	0.1803	0.3492
Deepest Class Used	-0.0388	0.0142
Number of Public Class Members	0.6527	0.4195
Number of Protected Class Members	0.4172	0.7500
Number of Private Class Members	0.2646	0.0873
Number of Member Functions	0.3768	0.1469
Number of Member Data Variables	0.7267	0.8718
Number of Friend Functions and Classes	0.0897	0.1888
Number of Overloaded Functions and Operators	-0.0197	-0.0210
Number of Static Class Members	0.5371	0.3762

Table 2. Correlation Between Object-Oriented and Non-Object-Oriented Metrics

This table does show some significant correlations. *Weighted Methods per Class* appears to possess significant correlation most notably with *Number of Public Class Members* and *Number of Member Data Variables. Halstead Class Difficulty* is significantly correlated most notably with *Number of Protected Class Members* and *Number of Member Data Variables.*

The belief that type complexity must be measured separate from control flow complexity is supported by the extremely low correlations between both traditional metrics and the object-oriented metrics that measure inheritance and polymorphism. The metrics *Inheritance Tree Depth*, *Inheritance Tree Size* and *Deepest Class Used* are the best measures of inheritance and polymorphism and none were correlated with either of the traditional metrics.

5 Conclusions

This study provides evidence that there are aspects of code complexity in C++ programs which are not adequately captured by the use of traditional, procedural metrics. A set of metrics is proposed and evidence to suggest the usefulness of some of these metrics is shown. Although the C++ language was used in this research there is no reason to suspect that it cannot serve as a

proxy for other object-oriented languages. Not all object-oriented aspects of the C++ language (e.g., friend declarations, static members, multiple inheritance) may be present in all object-oriented languages but this should not preclude the generalization of these results to other object-oriented languages.

Using a 0.05 level of significance each of the following metrics were determined to have significant correlation with the number of defects discovered during code reviews:

- Number of Classes Used
- Deepest Class Used
- Number of Protected Class Members
- Number of Member Functions
- Inheritance Tree Depth
- Inheritance Tree Size
- Halstead Class Difficulty

Additionally, the metric *Weighted Methods per Class*, which is the sum of the cyclomatic complexity of each member function in the class, was significant at the 0.10 level.

Some of the metrics proposed in this study, such as *Inheritance Tree Depth* and *Weighted Methods per Class,* are similar to or identical to metrics proposed elsewhere in the limited literature on object-oriented complexity metrics.[Morr89][Chid91][Lake92] Because both Morris [Morr89] and Chidamber and Kemerer [Chid91] present metrics but do not give empirical support for them, the evidence presented in this study can serve as quantitative support for at least a subset of their proposed metrics. Similarly, Lake and Cook [Lake92] document an experiment in program comprehension, debugging and modification which leads them to the conclusion that the depth of the inheritance tree is the primary factor affecting complexity in object-oriented code. This is supported by evidence given here showing the highly significant correlation between discovered defect rate and the proposed metrics *Inheritance Tree Depth* and *Inheritance Tree Size.*

Five metrics—*Inheritance Tree Depth, Inheritance Tree Size, Number of Classes Used, Deepest Class Used* and *Number of Protected Class Members*—are measures of the use of inheritance. All five metrics tested significant. Of the five metrics that measure polymorphism—*Inheritance Tree Depth, Inheritance Tree Size, Number of Classes Used, Deepest Class Used* and *Number of Overloaded Functions and Operators*—all except *Number of Overloaded Functions and Operators* were significantly correlated with discovered defects.

The conclusion that the use of inheritance and polymorphism may lead to increased class complexity must be weighed against the resulting decrease in control flow complexity, such as is measured by McCabe's cyclomatic complexity. The ultimate complexity of the resulting code will depend upon whether the increase in type complexity from the use of inheritance and polymorphism is more than offset by the reduction in control complexity. This question can only be fully settled on a case-by-case basis. The metrics shown to be correlated with defects in this study can serve as guidelines for a project manager in determining which classes may be suspected of being on the short-end of that tradeoff.

It is not suggested that there is a single magic number for any of these metrics which can serve as an indicator that a class has become "too" complex. Instead the metrics serve as directional indicators merely implying that, for example, as the size of the inheritance tree increases the class tends to become more complex. There is no specific value at which an inheritance tree becomes too complex; however, there may be a desirable range in which, on average, the tradeoff between type complexity and control complexity is optimized.

The usefulness of the metrics indicated by this research as possessing statistically significant correlation with defect rates can be found not in predicting high defect rates for any single piece of code but in identifying high defect risk designs and practices. Object-oriented languages offer a number of extensions to their predecessor languages. Each of these extensions is presumably included to give the newer language greater expressive power, more flexibility, more run-time safety or some similar advantage. In some cases these features may only be able to achieve these goals at the risk of increased complexity. This research indicates features such as inheritance and polymorphism are not without their risks in adding complexity.

Although this study discusses the use of object-oriented metrics at the level of C++ source code, there is nothing inherent in these metrics to preclude their application at the design level. Because most proposed object-oriented design approaches, including those documented by Booch, Rumbaugh and Coad ([Booc91], [Rumb91] and [Coad90]), recommend iterative approaches, class designs are created relatively early in the design process. By the early calculation of these metrics from class designs it may be possible to identify specific classes, or entire inheritance trees or subtrees, which may be at risk for a high level of defects.

5.1 Limitations and Need for Future Work

The most obvious shortcoming of this study is the small set of classes and programmers on which it is based. Although a number of the proposed metrics tested were statistically significant, a corroborating study of greater magnitude is called for. The heterogeneity of the studied classes is suspect because all of the classes were implemented by only four programmers whose experience levels with C++ all fell into the tight range of from three to five years. The fact that all classes were targeted for use on MS-DOS platforms also weakens any general claims as to the usefulness of these results.

Future work is called for in order to eliminate some of these concerns. A similar study encompassing more classes created by more programmers with more diverse experience levels and targeting additional operating systems would be an excellent step toward reinforcing or refuting the results presented here.

Two of the proposed metrics, *Number of Static Class Members* and *Number of Friend Functions and Classes*, rely on the use of C++ language features that were not heavily used in the sample classes. In order to ascertain any correlation between these metrics and class complexity it would be necessary to study classes with more diverse values for these metrics. For example, most classes studied here had either zero or one friend members and only one class went as high as four friends. A study using sample data that includes more classes with higher

values would be required to better understand the impact of the *Number of Friend Functions and Classes* on complexity.

Another possibly valuable area for further study would be to take an approach similar to that taken here but to use the number of defects discovered during system test rather than the number of defects found during code review.

This research has taken the positive step of building upon prior work in the area of metrics for object-oriented languages and has identified eight metrics that were statistically significant measures of class complexity as indicated by their susceptibility to defects. The next step should be to reinforce or refute these results, perhaps in one of the ways outlined above.

REFERENCES

[Booc86] G. Booch, "Object-Oriented Development," *IEEE Transactions on Software Engineering,* Vol SE-12(2), February 1986, pp. 211-221.

[Booc91] G. Booch, *Object-Oriented Design: With Applications*, Benjamin/Cummings Publishing Company, Inc., 1991.

[Chid91] S. Chidamber, and C. Kemerer, "Towards a Metrics Suite for Object Oriented Design," *Object Oriented Programming Systems, Languages and Applications (OOPSLA),* Vol 10, 1991, pp 197-211.

[Chid94] S. Chidamber and C. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering,* Vol SE-20(6), June 1994, pp. 476-493.

[Coad90] P. Coad and E. Yourdon, *Object-Oriented Analysis,* Yourdon Press, 1990.

[Curt79] B. Curtis, S.B. Sheppard, P. Milliman, M.A. Borst, and T. Love, "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," *IEEE Transactions on Software Engineering*, Vol SE-5(2), March 1979, pp. 96-104.

[Hals77] M.H. Halstead, *Elements of Software Science*, Elsevier, 1977.

[Henr81] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering,* Vol SE-7(5), September 1981, pp. 510-518.

[Jone91] C. Jones, *Applied Software Measurement: Assuring Productivity and Quality*, McGraw-Hill, 1991.

[Kafu87] D. Kafura and G.R. Reddy, "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Transactions on Software Engineering,* Vol SE-13(3), March 1987, pp. 335-343.

[Kole93] R. Kolewe, "Metrics in Object-Oriented Design and Programming," *Software Development*, Vol. 1, No. 4, October 1993, pp. 53-62.

[Lake92] A. Lake and C. Cook, "A Software Complexity Metric for C++," *Annual Oregon Workshop on Software Metrics*, March 1992.

[Morr89] K. Morris, "Metrics for Object-oriented Software Development Environments," Masters Thesis, MIT, 1989.

[Plau93] P.J. Plauger, "Programming Language Guessing Games," *Dr. Dobb's Journal*, October 1993, pp. 16-22.

[Rumb91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design,* Prentice-Hall, 1991.

[Schn79] N.F. Schneidewind and H-M. Hoffman, "An Experiment in Software Error Data Collection and Analysis," *IEEE Transactions on Software Engineering,* Vol SE-5(3), May 1979, pp. 276-286.

[Stev90] A. Stevens, *Teach Yourself C++,* MIS Press, 1990.

[Whit92] S. Whitemire, "Measuring Complexity in Object-Oriented Software," presented at the *Thrid Int. Conf. Applicat. Software Meas.*, La Jolla, CA 1992.

Appendix—Class Data

Class Number	Tree Depth	Tree Size	Summed v(G)	LSS	Classes Used	Deepest Class Used	Total Public Members	Total Protected Members	Total Private Members	Total Member Variables	Total Member Functions	Total Friends	Overloaded Ops & Functions	Total Static Members	Halstead's D	Total Defects
1	1	1	2	2	1	1	2	0	0	0	2	0	0	0	4	0
2	1	1	22	55	1	1	11	0	8	8	11	0	0	0	90	3
3	1	1	7	8	1	1	7	2	0	5	4	0	1	0	21	0
4	2	2	11	21	2	2	14	0	6	10	10	0	0	0	25	0
5	3	3	5	8	5	3	4	0	1	1	4	0	0	0	14	5
6	1	1	16	41	2	1	14	5	0	13	6	0	0	0	55	2
7	1	1	23	61	9	3	18	0	8	10	16	1	0	0	131	6
8	1	1	24	58	11	2	15	0	6	10	11	0	0	0	104	11
9	1	1	5	9	1	1	4	0	1	1	4	0	0	0	19	0
10	2	2	72	240	4	2	39	0	0	0	39	0	0	4	494	3
11	1	1	8	12	4	2	4	0	6	6	4	0	0	0	23	0
12	1	1	15	37	4	2	3	1	18	18	4	1	0	0	54	1
13	1	1	3	3	1	1	4	0	0	0	4	0	0	0	5	0
14	1	1	9	16	1	1	9	2	0	3	8	0	0	0	35	3
15	2	2	10	19	2	2	7	0	1	1	7	0	0	0	35	6
16	1	1	18	45	11	2	13	6	0	9	10	1	0	2	68	7
17	2	2	39	82	8	2	24	26	0	7	43	4	0	1	232	6
18	1	1	30	72	8	2	21	7	0	7	21	0	0	0	251	6
19	1	1	92	249	1	1	26	0	16	16	26	0	0	7	347	5
20	2	2	4	4	2	2	5	5	4	5	9	0	0	2	6	2
21	2	2	11	89	2	2	4	0	3	2	5	0	0	1	52	0
22	1	1	15	29	1	1	16	6	0	12	10	2	2	0	40	2
23	1	1	15	22	1	1	18	0	10	10	18	0	0	0	101	2
24	2	2	13	37	5	2	4	0	0	0	4	0	0	0	50	4

Class Number	Tree Depth	Tree Size	Summed v(G)	LSS	Classes Used	Deepest Class Used	Total Public Members	Total Protected Members	Total Private Members	Total Member Variables	Total Member Functions	Total Friends	Overloaded Ops & Functions	Total Static Members	Halstead's D	Total Defects
25	2	2	22	51	13	3	18	0	8	11	15	0	0	4	84	13
26	1	1	2	2	1	1	3	0	0	1	2	0	0	0	3	7
27	1	1	3	7	1	1	3	0	1	2	2	0	0	0	11	0
28	1	1	7	27	1	1	9	3	0	5	7	0	0	0	45	3
29	1	1	73	280	1	1	27	24	0	26	25	0	0	3	305	2
30	2	2	32	101	3	2	13	0	1	1	13	0	0	1	4	0
31	1	1	8	5	3	2	3	0	1	1	3	0	0	0	3	8
32	1	1	2	2	1	1	3	0	0	0	3	0	0	0	3	9
33	2	3	27	41	5	2	23	10	0	9	24	0	0	0	101	0
34	2	2	70	126	11	2	12	64	0	4	72	1	0	3	1084	22
35	2	2	27	77	2	2	10	0	2	2	10	0	0	0	100	0
36	2	2	13	13	6	2	14	9	0	9	14	0	0	0	22	9
37	1	1	5	7	1	1	7	0	0	2	5	0	0	0	9	0
38	2	2	1	5	2	2	5	0	0	4	1	0	0	0	2	3
39	2	2	15	17	2	2	12	4	0	4	12	0	0	0	45	5
40	2	3	22	40	5	2	13	6	0	4	15	0	0	0	194	6
41	1	1	9	13	1	1	5	3	0	2	6	0	0	0	21	4
42	1	1	28	86	1	1	32	2	0	24	10	0	0	1	77	12
43	2	2	33	115	9	3	5	3	0	2	6	0	0	0	118	7
44	3	3	13	17	3	3	15	14	0	16	13	0	0	2	62	5
45	4	4	4	7	4	4	8	8	0	8	8	0	0	0	29	11
46	2	2	1	1	2	2	1	1	0	1	1	0	0	0	3	6
47	3	4	4	7	10	3	10	9	0	9	10	0	0	0	49	9
48	1	1	1	2	7	3	4	0	0	3	1	0	0	0	2	14

Class Number	Tree Depth	Tree Size	Summed v(G)	LSS	Classes Used	Deepest Class Used	Total Public Members	Total Protected Members	Total Private Members	Total Member Variables	Total Member Functions	Total Friends	Overloaded Ops & Functions	Total Static Members	Halstead's D	Total Defects
49	3	3	5	10	3	3	4	4	0	5	3	0	0	2	17	11
50	4	4	12	32	4	4	2	0	0	0	2	0	0	0	17	7
51	1	1	2	2	1	1	3	0	0	0	3	0	0	0	3	0
52	4	4	10	26	4	4	2	0	0	0	2	0	0	0	16	8
53	1	1	17	24	1	1	14	0	5	5	14	0	0	0	54	5
54	2	2	33	55	6	2	22	3	0	3	22	0	0	4	121	7
55	2	2	43	97	6	2	22	2	0	2	22	0	0	0	195	7
56	1	1	6	10	5	2	36	5	0	20	21	0	0	0	9	0
57	2	2	42	70	4	2	26	6	2	7	27	0	0	12	115	3
58	1	1	26	56	1	1	7	0	0	3	4	2	2	0	132	4
59	1	1	41	100	1	1	14	0	8	7	15	0	0	1	178	5
60	1	1	20	78	3	2	3	13	0	12	4	0	0	0	82	7
61	2	2	5	3	1	1	5	8	0	8	5	2	0	0	6	2
62	2	2	36	85	3	2	3	3	14	17	3	0	0	0	142	6
63	1	1	1	1	1	1	1	1	0	1	1	0	0	0	2	0
64	1	1	45	157	1	1	14	15	0	13	16	0	0	3	174	4
65	1	1	2	6	2	1	2	6	0	5	3	1	0	0	3	6
66	1	1	8	26	2	1	3	5	0	5	3	0	0	0	38	4
67	1	1	19	77	3	2	3	13	0	12	4	0	0	0	89	0
68	1	1	13	37	1	1	6	7	0	6	7	0	0	0	48	3
69	2	2	40	108	2	2	19	15	0	20	14	0	0	0	147	6
70	1	1	10	14	1	1	8	3	0	4	7	0	0	0	23	9
71	1	1	19	57	1	1	10	0	8	10	8	0	0	0	81	4
72	1	1	14	23	1	1	6	0	2	2	6	0	0	0	48	10

Mother2 and MOSS: Automated Test Generation from Real-Time Requirements

Joe Maybee

Tektronix, Inc.
Graphic Printing and Imaging Division
P.O. Box 1000, M.S. 63-430
Wilsonville, Or 97070
maybee@pogo.wv.tek.com

Portland State University
Department of Computer Science
P.O. Box 751
Portland, Or 97207
maybee@cs.pdx.edu

Abstract

Can the effort used for test writing be used to write requirements instead? A simple requirements specification system combined with an automated test fixture holds great promise. This paper examines one such approach.

Keywords and Phrases: Prolog, real-time embedded systems, formal methods, automated test methods, requirements testing, electromechanical systems.

Biographical: Joe Maybee has been building real-time embedded systems for Tektronix, Inc. since 1978. Recently, Joe has focused on software quality in reactive systems and the accurate definition of software requirements. Joe also teaches Software Engineering at Portland State University.

Joe is the recipient of an *Outstanding Achievement Award for Innovation* from the *1992 Pacific Northwest Software Quality Conference* for his work with the MOTHER test system, and first-place winner at the *1993 Tektronix Engineering Conference* for his analysis of data gathered about the MOTHER system.

Mother2 and MOSS: Automated Test Generation from Real-Time Requirements

Joe Maybee
Tektronix, Inc. / Portland State University

I. Introduction

Test fixtures can be valuable tools as part of a comprehensive QA strategy [1,2]. Fixtures can buy valuable time during the endgame phase of a project. However, fixtures are only the beginning of a strategy to build an effective QA process.

The Graphics Printing and Imaging Division's (GPID) print engine QA group is involved in the evaluation of real-time process control firmware for desktop color printers. Our QA evaluation process is geared toward the fast evaluation of printer firmware. Since we refer to the firmware as "software" in our group, I will be using the word "software" throughout this paper.

In order to improve GPID's QA process, we are engaged in constant evaluation of both strategies and tools. Having evaluated our previous successes and failures, we have installed incremental improvements in our process. In the case of our test fixture, we have redesigned and re-implemented MOTHER[1] [1] to eliminate the shortcomings in the original fixture.

The original MOTHER fixture allowed us to run large numbers of tests in a predictable amount of time. In addition, the original MOTHER tests were embedded in the requirements specification to allow easy maintenance of requirements and tests.

Our original system allowed our QA group to focus on maintenance of requirements, rather than exclusively focusing on the maintenance of tests. Since we were able to maintain requirements and tests efficiently and to run tests quickly, we turned to the problem of writing requirements and generating tests directly from these requirements. This paper describes the second generation test fixture (Mother2) and the requirements specification system we developed to generate tests directly from the requirements (MOSS)[2].

1. MOTHER stands for "Maybee's Own Test Harness for Evolving Requirements"

II. A Key QA Contribution: Fast Evaluation of Software Releases

Fast evaluation of software releases is an important QA contribution in the final phase of a project. In the following sections, I will discuss the reasons for this and how our organization has supported fast evaluation in the past.

The Key Project Phase: "Endgame"

The problems faced by QA engineers are almost universal. Specifically, the environment in which they work is chaotic and stressful. In particular, requirements are not always well considered and are volatile: change to the requirements can occur on an hourly basis in some environments. In addition, there is a great deal of pressure on the QA team to produce results quickly.

It is a fact of life that requirements are not always well considered. In many companies, market competition produces time pressure that is extreme. These time pressures frequently translate to pressures to reuse requirements that were incomplete, or to skip writing requirements altogether.

Today the belief that requirements can be "frozen" has fallen by the wayside. Markets have become far too volatile and unstable to coexist with long product development cycles. It has become necessary to be able to change product features in midstream easily and effectively.

The endgame phase of a software project brings with it higher levels of stress. Like the endgame in a chess match, it is the point in the game where you have either prepared yourself to be in the winning position or you have lost the game. In a chess game it is usually possible to recognize winning or losing endgame positions. In a software endgame, winning positions are defined by whether you are prepared to discharge your duties and responsibilities in an effective and efficient manner.

A losing position in a software endgame is usually recognized by who is getting most of management's attention. In all fairness to management, their close attention at the closing phase of the project is not entirely arbitrary. Typically, advertisements have been bought at great expense, usually at least three months prior to publication. Also, in many instances, release dates have been chosen to coincide with conferences and expositions. Floor space in the vendors section has been rented, and is usually quite expensive. Finally, advance sales contribute to the endgame pressure.

In light of these factors (advertisements, conference schedules and advance sales), there is little or no surprise that pressure escalates as the final ship date approaches.

---

2. MOSS stands for "Maybee's Own Specification System". I hope that future tools will be named after someone or something else.

The QA Organization and the Project Endgame

A good QA organization will understand and support the schedule in the most efficient manner possible. In short, QA will keep itself off the critical path for the software. At the end of the project, the software is typically passed from design engineering to QA in periodic releases. After a release of the software has been given to QA, management attention turns to QA. When the attention of management is focused on QA, it is best if the QA process runs like a well-oiled machine.

QA can be most effective during the closing phase of the project by producing comprehensive test results quickly. Project managers are under a great deal of stress after a release goes out, and some of that stress can best be relieved by a quick enumeration of defects. In our organization, we strive to provide complete functional testing within 48 clock hours of a release.

By "complete functional testing" we mean that all test suites that test functionality will be run within 48 hours. After the initial test cycle, release testing needs to answer two basic questions for management:

1. Did design engineering fix the defects they claim they fixed?
2. In the course of fixing defects or adding features, did the design team break anything that worked previously?

These two questions are best answered by checking test results with previous runs. Consequently, QA needs to be prepared with complete test suites before the first release, in order to establish a baseline of results for future comparisons.

Our Old Method: MOTHER

The original test fixture (MOTHER) demonstrated that fast and effective turnaround was best served by an automated test fixture. MOTHER also demonstrated that test fixtures are powerful tools for serving the project endgame strategy outlined above [2].

With the original MOTHER fixture, we embedded the tests in the requirements document so that when a requirement changed, the test was easy to locate and change as well.

It became apparent that there were problems with the MOTHER approach, however. While MOTHER supported the real-time aspects of testing reasonably well, the sequential nature of its test language did not support changes to timing relationships. Consequently, the burden for calculating time differentials was placed on the test writer. In addition, the *sequence* of operations in the time domain had to be determined by the test operator. If a response to a system stimulus was a series of operations, the test writer had to determine the order of these operations, and calculate the time differential between each of the operations.

Timing relationships between the operations became a problem to maintain, especially with the number of tests that were written. In one instance, a timing relationship change caused a significant delay while we updated the tests: we were not able to run complete

tests for 10 days. We learned a valuable lesson from this: we were not serving the test writers needs with the current MOTHER language. Luckily, this happened only once on the original project.

III. Our New Approach

We decided to take what we had learned from the original MOTHER fixture and specify, design, implement and test a new fixture: Mother2.

Mother2 was designed to implement the concept of formal real-time requirements in a fashion that would be easy to maintain. In addition, plain English text requirements could be generated from the formal notation for review by engineers not familiar with the formal notation.

What is a Real-Time Requirement?

We used Real-time finite state automata (RT/FSA) as the fundamental model for our product software. Real-time FSA is one of the most natural and straight forward requirement methods for real-time reactive systems.

We may consider an FSA requirement a series of facts about:

- Initial State for this requirement. That is, the state that the system is in at the beginning of this requirement.
- Stimulus applied to the system.
- Response of the system to the state.
- Final state of the system. That is, the state that the system is in after the response to the stimulus is complete.

Since each and every requirement should have a unique identifier, we could write an FSA requirement as a five-tuple:

```
Requirement (Unique_id, Start_state, Stimulus, Response,
    Final_state).
```

However, this is insufficient for real-time requirements.

With real-time systems, we consider time as an unconstrained input to the system. In real-time systems we measure this input via a timebase, usually a real-time clock or deadman timer. Time becomes a critical factor in real time systems and Jaffe, et al. suggest that every input and output in a requirement should be constrained by a time window.

Consider the following example requirement for a Phaser 300i color printer:

- Initial State: ready.
- Stimulus: Media Eject (output) sensor is blocked [0 to 1000 mSec].

- Response: Transition to final state.
- Final State: fault_jam state achieved within 2000 mSec.

In this instance, the system response is covered by its transition to the final state. That is, nothing out of the ordinary is required of the system other than to attain the required final state. In the notation of our requirement system we could write this requirement in a fashion very similar to the form previously mentioned:

```
req(ready_12,ready,output_sensor(blocked),nothing,
    fault_jam, t2sec).
```

The requirement above follows our original requirement format with an additional parameter at the end of the list. This additional parameter specifies how long the system has to make the transition to the final state.

We refer to the stimulus as the *input predicate* and the system response as the *output predicate.* In the above example the input predicate is a single input with a simple value: output_sensor takes on the value "blocked". The output predicate is "nothing", indicating that the requirement is satisfied if the final state is achieved and nothing else.

Maintaining the Time Relations

It was apparent that we needed to make the Mother2 fixture intelligent enough to determine what the timing relationships should be. This feature would allow the test writer to specify in a single place what the expected time for a response should be, and to compose complex responses based on combinations of these times. If the time should change, it can be revised easily, and the new timing relationships would be automatically calculated by the Mother2 fixture.

Checking the Requirements

Previously, the only way to check for errors in the test itself (i.e. semantic or syntactic errors) was to run the test. The original MOTHER fixture would then flag problems with the tests. This was considered to be time consuming effort. We designed our new requirements specification system (MOSS) to allow for a test "prescan". Since complex responses are built up from elementary "building blocks" (inputs and outputs arranged in time), it is possible for us to check these complex responses. In order to "prescan" the test, we simply need to check that every requirement in the system is composed of valid components: valid states, stimuli and responses. Since system stimuli are composed of inputs, outputs and specified times, these in turn can be checked to ensure that their basic elements have been specified.

For maintenance purposes we decided to modularize the knowledge bases for the requirements. By allowing the test writer to build and maintain separate knowledge bases for the requirement, we distributed the knowledge base across units of manageable size. For instance, the inputs, outputs and their valid values could be maintained in one

knowledge base file, while the combinations of outputs that compose a system "response" could be put in another knowledge base file, etc.

This new requirements specification system had the capabilities of emitting Mother2 tests as well as English language requirements from the new requirements database. We prefixed each Mother2 test with the English language representation of the requirement which produced a nice "header" for the test. (See the example test later in this paper.)

In short, we eliminated the test writers and turned them into requirements writers. We built a primitive expert system that would analyze the requirements, report errors in the requirements and generate tests. Since the tests were generated automatically, we were able to use the effort of our test writing team entirely for support of our requirements specifications.

Our new requirements specification system (MOSS) was written in Prolog.

The Power of Prolog

Prolog is a language that supports logic programming. Logic programming is a powerful tool for establishing relationships among facts in a knowledge base. We found Prolog to be a very natural tool for dealing with our requirements for several reasons:

1. Requirements are a collection of assertions about the desired behavior of the product software.
2. Requirements are composed of facts about the system (i.e. inputs, values of inputs, outputs, values of outputs, etc.).
3. Problems with the requirements can be detected using a set of rules. For instance, if a requirement has in its input predicate something that cannot be identified as an input, or a list composed of inputs, then we have a malformed requirement. Prolog allows us to make the assertion that a particular parameter is a fact (already listed in the Prolog fact base), and to detect exceptions to this assertion.
4. Tests can be generated using a set of rules on how to translate facts about requirements into the Mother2 test language.

The Problems of Prolog

Prolog, however is not without it's drawbacks. In particular, we encountered three types of problems:

1. Database order dependencies.
2. Range problems.
3. Data Typing.

It takes some knowledge and experience to write effective Prolog programs. Since we were terribly inexperienced with Prolog, we were capable of coding some unusual but fascinating bugs into MOSS. Our greatest problem was that we were unable to locate any Prolog experts to answer our questions. However, after some time and research we were able to come up with several excellent sources dealing with logic programming guidelines and Prolog style. [4,5,6]

Database order dependencies

Some of the major problems that we encountered were order dependencies in the fact base. Prolog has strict rules about the evaluation and search order of the Prolog database, and the order in which rules and facts are set down can have consequences. Knowledge of good Prolog programming style will prevent many of these problems.

Range problems

Prolog cannot solve certain problems well. In particular, the nature of the language suggests that all assertions based on the rule base can be dealt with explicitly. With Prolog, this is not entirely true. Dealing with constraints, such as constrained numeric ranges, requires knowledge of Prolog's limitations. CLP(R) is a language that has been developed which deals with these problems in a much more natural fashion. Any QA team building a MOSS-like requirements specification system may wish to investigate CLP(R).

Typing

Prolog is not a strongly typed language. If data is to carry type information, that type information must be built into the database. This explicit typing requires more information to be built into the fact base, and adds to the complexity of the Prolog database. For example, if output_sensor is a valid system input, a fact to that effect must be present in the database for purposes of validating the predicates:

```
system_input(output_sensor).
```

We have determined that a typed language, such as GOEDEL is a viable and desirable alternative to Prolog for solving this particular problem.

Composing Requirement Building Blocks

We have already stated that we wished to have automatic evaluation of time relationships determined by the Mother2 fixture. Typically, these times are significant to many different requirements.

We use "time tags" to describe these significant time intervals with a symbolic name. The general form of the time tag is:

```
time(tag_name, T_zero, Delta_T, 'time units' ).
```

For example, the amount of time to wait for a power-up event may be described as "pwr_time", and information about it could be contained in the Prolog fact base:

```
time(pwr_time, 200, 100, 'ms' ).
```

This "time tag" can now be used in our requirements and need not be explicitly stated. Wherever we use the "pwr_time" tag, our system will know that we are talking about an interval beginning at 200 mSec and continuing to 300 mSec (the 100 mSec is a delta-t from the beginning of the interval.)

We can now use these symbolic names to assemble "time lists", which are lists of time constrained events.

Building a Time List

If the time required for a power-up event should ever change, we can change the time tag description in the database, and the time will change throughout the requirements database.

In addition, when composing lists of events, these events need not be in time order. Consider the following "initial state list":

```
initial_state_list(ready,
[
    [reset_system, 'push', push_time],
    [reset_system, 'free', free_time],
    [system_pwr, 'on', pwr_time],
    [front_panel, '"AReady-- "', t20min]
]
).
```

This "initial state list" contains a specification on how to walk the Phaser 300i printer into a state we have named "ready":

1. Set the Mother2 "reset_system" output to "push" at "push_time".
2. Set the Mother2 "reset_system" output to "free" at "free_time".
3. Set the Mother2 "system_pwr" output to "on" at "pwr_time".

4. Wait for the Mother2 input "front_panel" to attain the ASCII value "Ready--" during the interval "t20min".

The Mother2 system contains a scheduler that will arrange these items in time order. So, we need not be concerned with whether the events occur in the order specified: The Mother2 fixture will impose the proper time order just before run time.

Many parts of a requirement can be composed of these "time lists". Therefore, our requirements are composed of:

- *"Initial" state lists:* a list of events (and their times) that must occur to "walk" the system into the "initial" state for testing.[1] The t-zero for our tests occurs immediately after the Mother2 fixture has verified that we have achieved our initial state.
- *Input predicate lists:* a list of events that describe a stimulus (and their times) that should be applied to the system after the "initial" state has been reached.
- *Output predicate lists:* a list of events that describe a system response (and their associated times) that should be expected from the system after the input predicate has been applied.
- *Terminal state lists:* a list of values for outputs that should be expected in order to verify that the system has achieved its terminal (final) state.[2]

How Tests are Generated from Time Lists

As one might expect, the generated tests for the Mother2 fixture follow the natural progression of the requirement:

1. *Walk the system into it's initial state, using the information in the initial state list.* If the system fails to achieve the initial state, a failure exists.[3] If the initial state cannot be achieved, the test is flagged as failed, and Mother2 proceeds to the next test.
2. *Apply the stimulus to the system as defined in the input predicate list.* The time orders of the stimuli and responses are determined by the Mother2 fixture.

---

1. "initial state" does not imply the initial state for the system, rather it is intended to denote the point of departure. Therefore, our initial state could be a ready state, error state or any other state in our requirements specification.

2. As with the "initial state", "terminal state" refers to the state the system should arrive at after the response is complete. The "terminal" state could be the ready state, the power-up state, or any other valid system state.

3. A printer may fail to achieve a state because it has exhausted its consumables: the paper tray may be empty, or the printer may be out of ink. This does not imply a failure of the requirement, it means that the printer needs attention. In this instance Mother2 will "ask" the operator to correct the problem and then try again.

3. *Watch the system to see that the response from the system occurs as defined in the output predicate list.* If any of these responses fail to occur within the times specified in the output predicate, the test is flagged as failed, and Mother2 proceeds to the next test.
4. *Check to see that the terminal state is achieved as defined in the terminal state list.* If the terminal state cannot be achieved in the time specified, the test is flagged as failed, and Mother2 proceeds to the next test.

This follows the natural sequence of the requirement specifications: state, stimulus, response, final state.

We have reduced the elements of the requirements into collections of elementary inputs and outputs arranged in time. These collections of inputs and outputs are grouped into test "phases": initial state, stimulus, response and terminal state.

Consequently, there are only two basic operations that the Mother2 fixture needs to perform in order to test our real-time system:

1. Set an input to the system under test to a specific value within the time interval specified.
2. Check an output from the system under test to see that it achieves a specific value within the time interval specified.

These two fundamental operations translate into another two basic Mother2 operations:

1. "tset": set a value to an input in a specified time window.
2. "twaitfor": wait for an output to take on a specified value in a time window.

In addition to these two basic operations, information had to be grouped into what we called "timegroups". For instance:

- *stategroups:* state groups were groups of Mother2 commands that would be re-tried (at the operators discretion) if they failed. stategroups were used to achieve the initial state. stategroups are delimited with "statestart" and "stateend" keywords.
- *timegroups:* these are time critical groupings of tsets and twaitfors that may not be retried if they failed. Input and output predicates fall in this domain, as do terminal state checks.

An Example Requirement and Its Generated Test

Here is an example of a requirement, it's associated Prolog database, and the generated test:

Requirement:

```
req(ready_12,ready,output_sensor(blocked),nothing,
    fault_jam, t2sec).
```

Ground base definitions:

```
system_input(output_sensor).
system_input(reset_system).
system_input(system_pwr).

system_output(front_panel).
system_output(fault).
system_output(centronic).

output_sensor(X) :- member(X , [blocked, notblocked,
    release]).

time(push_time, 0, 100, 'ms' ).
time(free_time, 100, 100, 'ms' ).
time(pwr_time, 200, 100, 'ms' ).
time(t20min, 10000, 1200000, 'ms').
time(t10_10,10000,10000, 'ms').
time(t02sec, 0, 2000, 'ms').
```

State list definitions:

```
initial_state_list(ready, [
    [reset_system, 'push', push_time],
    [reset_system, 'free', free_time],
    [system_pwr, 'on', pwr_time],
    [front_panel, '"AReady-- "', t20min]
 ] ).

terminal_state_list(fault_jam,
 [
    [fault, 'low', t02sec],
    [centronic, '"Paper_Jam SET"', t10_10]

 ] ).
```

Generated Test:

```
//
// Requirement: ready_12
//
// State: ready
//
// Stimulus: Input output_sensor takes on value blocked
//  at 0 ms
//
// Response: nothing
//
// Requirement to be met within the time period
// starting at 0 ms and ending at 2000 ms.
//
// Next state: fault_jam
//
//

teststart ready_12

statestart
tset 0 100 SUTReset push
tset 100 100 SUTReset free
tset 200 100 SUTPwr on
twaitfor 10000 1200000 LCDLine2 "AReady-- "
stateend

timestart
tset 0 1000 MediaEject blocked
timeend

//
// Check final state
//

timestart
twaitfor 0 2000 Fault low
twaitfor 10000 10000 Centronics "Paper_Jam SET"
timeend

//
// ************* End of Test *************
//

testend
// testend: ready_12
```

IV. Evaluation of Effectiveness

In the process of evaluating our next generation system, we ran the original MOTHER test fixture in parallel with the Mother2 fixture. This comparison was intended to evaluate and compare the effectiveness of Mother2. Mother2 was allocated a subset (approximately 80%) of the requirements to test in parallel with the MOTHER test fixture.

Turnaround time for these tests was 30 clock-hours on a single MOTHER fixture, and 10 hours for two Mother2 fixtures, which would translate to approximately 20 hours for a single Mother2 fixture. Therefore, we may consider Mother2 more efficient than the original MOTHER.

All defects found by the MOTHER fixture were also found by the Mother2 fixture (for the 80% of the requirements that were in the common requirements subset). This was a great confidence booster for us.

On one occasion, a wide-ranging timing change was required (a timing change which affected a large number of requirements), and this timing change was made in the MOSS requirements specification in less than an hour. The original MOTHER tests required about week of modification before the tests were properly revised.

However, not everything worked well. We discovered that in building our input predicates, it would have been nice to use other predicates as "building blocks." In short, we would have liked to write predicates by telling MOSS to use other predicates as departure points, and to incorporate some additions.

The Prolog language is difficult for requirements writers. Prolog is a somewhat arcane language, and we need to liberate our requirement writers from having to know Prolog.

We are in the process of incorporating the following changes to our Mother2/MOSS facilities:

1. We are writing a front-end to MOSS which will allow us to generate Prolog databases from a much looser requirement specification form.
2. We are in the process of installing facilities in MOSS which will allow us to build predicates as aggregates of other predicates.
3. We are in the process of trying to translate criteria laid down by Jaffe and Leveson, et al. into Prolog clauses. This will automate the analysis of our requirement specifications.

I hope to report back about the problems and successes of these efforts at a future conference.

V. Acknowledgments

I would like to take a moment to acknowledge the people who were essential in making this paper possible.

First and foremost, my manager, **Raul Krivoy**. Raul's abiding faith in our abilities and his undying commitment to quality have been the key to our successes. Thanks also to my friend and colleague, **Gene Welborn**, who is the coauthor, codesigner and coimplementor of the Mother2 portion of our system. **Harry Ford** translated lots and lots of requirements into the new formats,[1] built the knowledge bases, and managed to keep both his sanity and humor. **Richard Waugh** managed to translate vague descriptions of what we were trying to accomplish into concrete implementations of hardware interfaces. **Mike Rogers'** improvements to the MOTHER fixture gave us an ever-improving standard of quality to try and beat with the Mother2 fixture. Last, but not least, my wife **Jan Maybee**, who proofread the initial and final drafts and entertained our two-month old daughter, **Annie**, while I wrote this paper.

VI. References

[1] Maybee, Joe, *MOTHER: A Test Harness for a Project with Volatile Requirements.* Proceedings of the Pacific Northwest Software Quality Conference, Portland Oregon, 1991.

[2] Maybee, Joe, *True Stories: A Year in the Trenches with MOTHER*, Proceedings of the Pacific Northwest Software Quality Conference, Portland, Oregon, 1993.

[3] Jaffe, Matthew S, et al. *Software Requirements Analysis for Real-Time Process-Control Systems*, IEEE Transactions on Software Engineering, Vol. 17, No. 3, pp 241-258, March 1991.

[4] Ross, Peter, *Advanced Prolog*, Addison-Wesley, New York, 1989.

[5] Coelho, Helder and Cott, Jose C., *Prolog by Example*, Springer-Verlag, Berlin, 1988.

[6] Knuutila, Timo, *Efficient Prolog Programming*, Software-Practice and Experience, Vol. 22(3), pp 209-221, March 1992.

1. Actually, Harry translated ALL of the requirements used by MOSS and Mother2.

An Object-Oriented Approach for Testing Software from GUI

P. Nesi and A. Serra
Dipartimento di Sistemi e Informatica, University of Florence, Italy
ASIC S.r.l. Via Stefano Clemente 6, 10143 Torino, fax +39-11-4371916, Italy

Abstract

This paper describes the experience of introducing the object-oriented paradigm in an automated execution tool, as an alternative to the traditional script languages, currently used in capture & playback (C&P) tools.
A non-invasive C&P approach can be very useful for testing applications in distributed environments and, in general, for testing applications without modifying their run-time environment. In the software life-cycle, the phases of C&P are performed after the application design. Being that this period is close to the deadline of delivering, the time needed to test the application is considered as a high, and frequently unacceptable, cost.
In this paper, a new approach for non invasive C&P testing technique is proposed. This is strongly based on the object oriented paradigm.
A special new board for image grabbing and pattern matching, and a new language to specify the tests have been implemented.
The main goals of this new approach are (i) the reduction of testing time by supporting the reuse of tests (coded using a particular language) at each level of abstraction, and (ii) the anticipation of the testing phase by overlapping it with the last phases of software life-cycle.
The object oriented approach described in this paper is, with respect to the traditional capture & playback, descriptive and data-driven instead of procedural and code-driven.

Keywords: *testing, capture and playback, non-invasive, object-oriented language, real time, distributed applications.*

1 Introduction

One of the most powerful methods for testing software applications with man-computer interactions, is the so-called Capture and Playback approach [1]. It is based on two distinguished phases, which are the Capture and the Playback. During the Capture, each computer-user interaction is collected, that is all the messages which are displayed on the monitor (presentation of messages, windows, text, etc.), all the keys pressed coming from the keyboard and all the mouse activities. In this way, the histories of computer interactions are collected, and stored in a form of script file.
These histories of man-computer interactions can be re-proposed to the computer interfaces (simulating the presence of the keyboard and the mouse) for simulating the presence of the user itself. After each simulated stimuli, the responses of the computer can be tested to verify the application answers on the video signal, according to its correct behavior (which is supposed to be known).
The history of sequences and the sequences of operations that must be performed for testing the applications, are usually specified with an ad-hoc programming language, in a script file (in ASCII form). This allows direct modification of these programs, following the syntax and the semantics of the script language [1].

It is known that, in the taxonomy of automatic testing tools, there are software-based and hardware-based C&P tools. We also use the term "invasive" for the software-based techniques, and "non-invasive" for the hardware-based approaches.
The software approach consists of a resident program which runs (capturing and playing back) on the same machine on which the application under test is running. The hardware approach consists of dedicated hardware which is capable of grabbing the signals passing through the cables linking the monitor, the mouse and the keyboard to the computer body itself. In this case, a second computer, usually called HOST, is needed to control the processes of C&P on the System Under Test (SUT). The tool architecture is shown in Fig. 1 below.

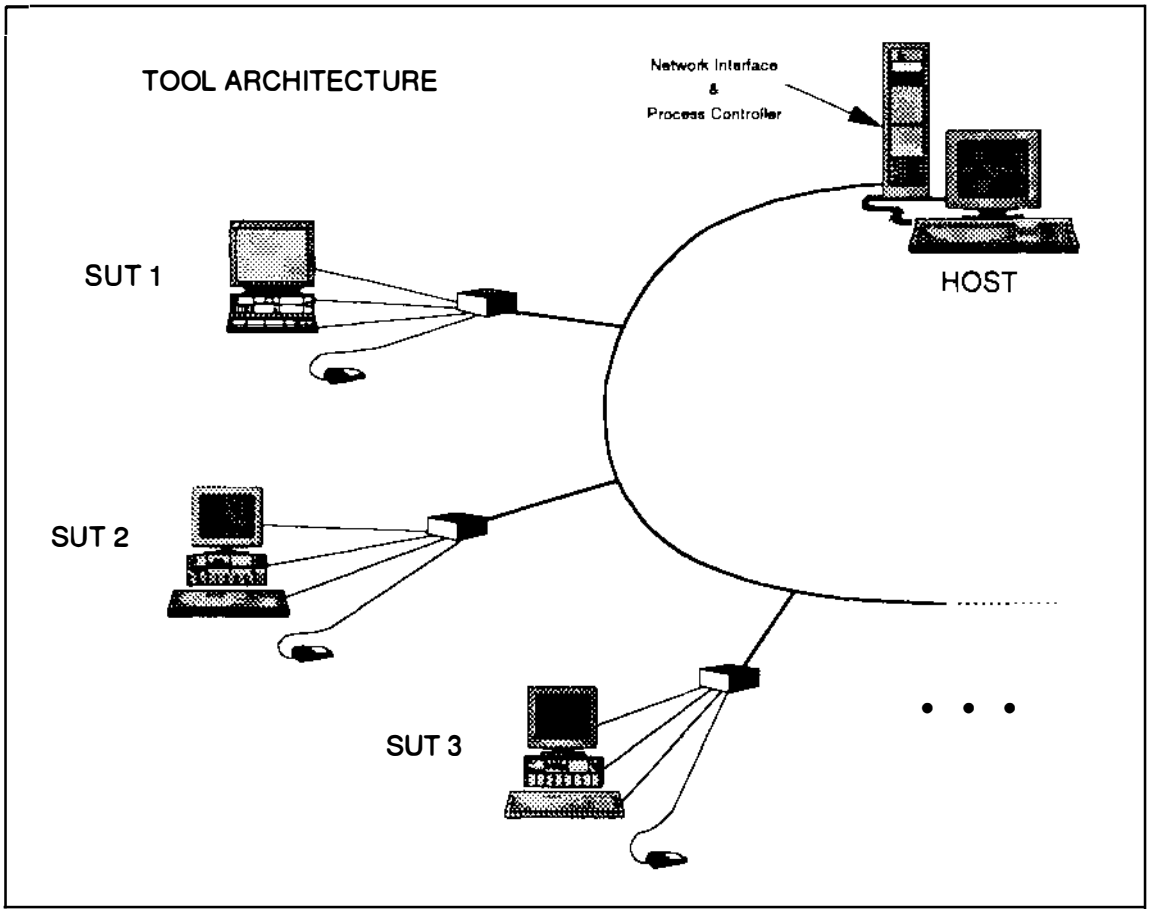


Fig. 1: *Non-invasive approach to Capture & Playback capabilities for testing distributed and/or multiple applications. The emulation boards (video grabbers) are connected to the HOST by means of a dedicated network.*

This last approach is much more robust with respect to the first that, being invasive, obviously, changes the real context where the software under test runs. This can affect the performances of the software under test and all testing results. Some failures could be due to the presence of the tool itself and some errors may not be revealed.
The non-invasive approach can be very useful for executing the tests in real-time and for testing distributed environments where several computers execute the same or different applications, possibly

running under different operating systems, at the same time. On the contrary, the invasive approach is surely unsuitable for testing (i) time-dependent operations, (ii) distributed applications and (iii) applications very sensitive to the operating system environment conditions. The better performances of a non-invasive approach with respect to the invasive one is paid with an increment of cost and complexity. In fact, in the non-invasive approach, dedicated hardware is needed. On the other hand, the improvement of performance and the possibility of real-time testing of distributed applications are unfeasible with invasive approaches.

The C&P approach is usually applied by companies which are focused on software development to maintain control of the product quality or by the end-users to control the quality and to assess the new version of software product already in use. In both cases, the phases of C&P are performed after the design and formal test of the application. This fact, for the companies which are focused on software development, makes C&P strongly unsatisfactory since the time which is necessary to perform the tests is often past the delivery date, due to the delays accumulated during the software development. For this reason, the time which is needed to test the application by means of a C&P, is considered as a high and frequently unacceptable cost. This is particularly important on the first market release rather than on subsequent versions of the same product. This is due to the fact that, during the first time of testing by means of a C&P approach, all the screen elements (in the following called *entities* or *gadgets*) such as windows, dialog boxes, buttons, scrollbars, list boxes, etc. (see Fig. 2), mouse activities and keyboard sequences must be captured, while for a new version of the same application, only a part of the work performed in the capture phase must be done again, modifying the scripts according to the changes occurring in the application itself.

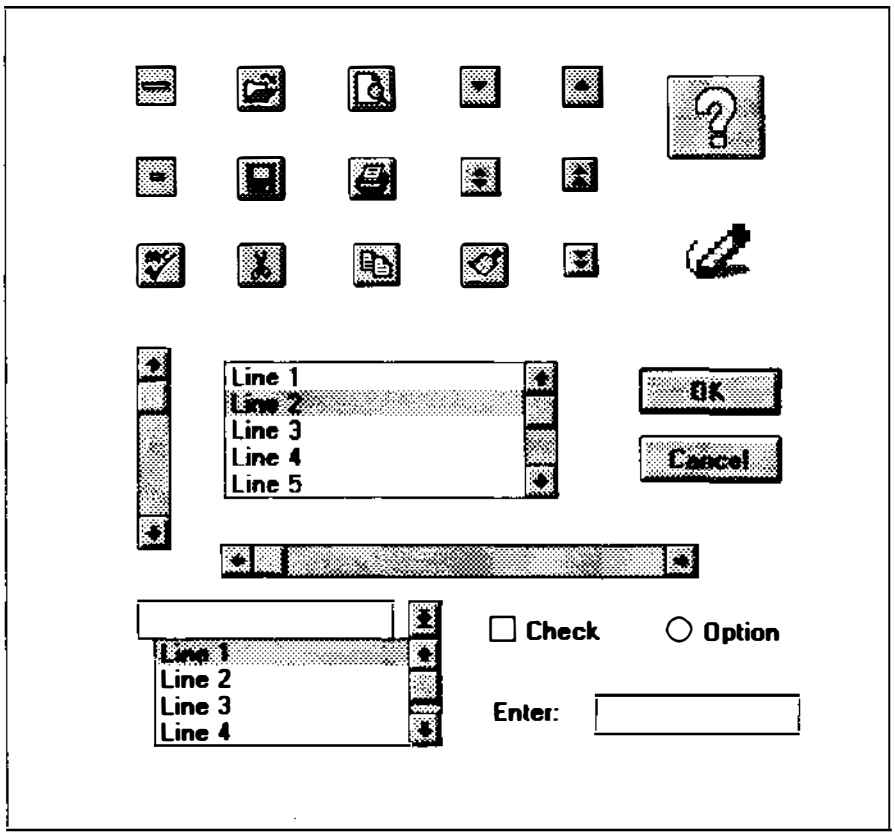


Fig. 2: *Examples of visual entities (gadgets)*

As remarked in many studies, one of the main limits in the first generation of capture & playback tools, when applied in Graphical User Interface (GUI)-based applications, regards the maintainability of recorded scripts.
When used in a context of regression testing activities, these tools require absolutely stable (at the pixel level) graphic interfaces. Any minimum change in the user interface, such as the shape of an icon, makes the recorded scripts unusable to the new version of the software under test. This means that, in practice, a continuous activity of re-creating old scripts is necessary; and the regression testing is no longer an automatic process.
Another problem is that script programs are not easy to understand nor easy to be modified. Even if the script is an ASCII file, the modification with a standard editor should be an easy-to-do activity but is not. This problem is due to the fact that human operations described in the script (mouse movements, screen checking, etc.) are "physical" and not "logical" items.
The new generation of software-based capture & playback tools introduces smart events and solves some of the problems described above. Hardware-based tools in the market still maintain the same "physical" approach in the script description languages.
In this paper, an evolution of the non-invasive C&P approach, implemented by ASIC [1], is proposed. The main goal of this approach is to develop a hardware-based testing tool which solves the problems related to script maintainability and comprehensibility, while preserving all the advantages of the non-invasive solutions.
This new approach is strongly based on the object paradigm at both hardware and software levels of the system.

The entire application is object-oriented. A new language, called LOOT (Language Object Oriented for Testing), to describe the tests, has been developed.
The object-oriented paradigm allows the representation of relationships among the main entities, or gadgets, which are present in the user interfaces such as windows, buttons, patterns, etc., as it has been done in many object-oriented user interface proposed in literature [4], [5], [7], [8], [11]. In addition, the object-oriented paradigm supports reusing already defined objects and thus reducing the cost of maintenance and expansion. In fact, one of the goals of the approach is to reduce the time of capture. Therefore, the main goals of this new approach are (i) the reduction of testing time, and (ii) the overlapping of the testing phase with the last phases of Software Life-Cycle.
The paper is organized as follows. In Section 2, the main aspects of the object-oriented definition and implementation of a non-invasive testing tool for C&P are reported, with a discussion about the main drawbacks of the traditional approaches. In Section 3, the architecture of the testing tool is proposed. In Section 4, some examples about the Language Object-Oriented for Testing, LOOT, are reported in order to show its main capabilities. Conclusions are drawn in Section 5.

2 Object-Oriented-Based Capture & Playback

The approach proposed in this paper has been motivated by the fact that most of the classical C&P approaches have several drawbacks. In particular:

1 Testing a new version of previously tested application, it can happen that a slight change in the visual profile of a dialog box (or windows, etc.), such as the displacement of a gadget (a button, a scrollbar) of the dialog box is *recognized* as a severe error even if the semantic of the dialog box (or window, etc.) is unchanged (see Fig. 3).

2 Testing a new version of previously tested application, it can happen that a slight change in the visual profile of a dialog box (or windows, etc.), such as the addition of a new gadget (or menu) is *not recognized* as a severe error even if the semantic of the dialog box (or window, etc.) has been *substantially* changed. Moreover, in this case, the old test, coded in the script, is performed on the new version of the dialog box (or windows). The verification of differences about the shape of a visual entity must be explicitly coded in the script of the test.

3 In the case of. testing a new version of previously tested application, it can happen that a slight change in the visual presentation such as the displacement of the window when it is open (or dialog box) is *recognized* as a severe error even if the semantics of the application are unchanged and the graphic entities are the same. This can happen even when a different environment is used during two different sections of testing the same application (of the same version).

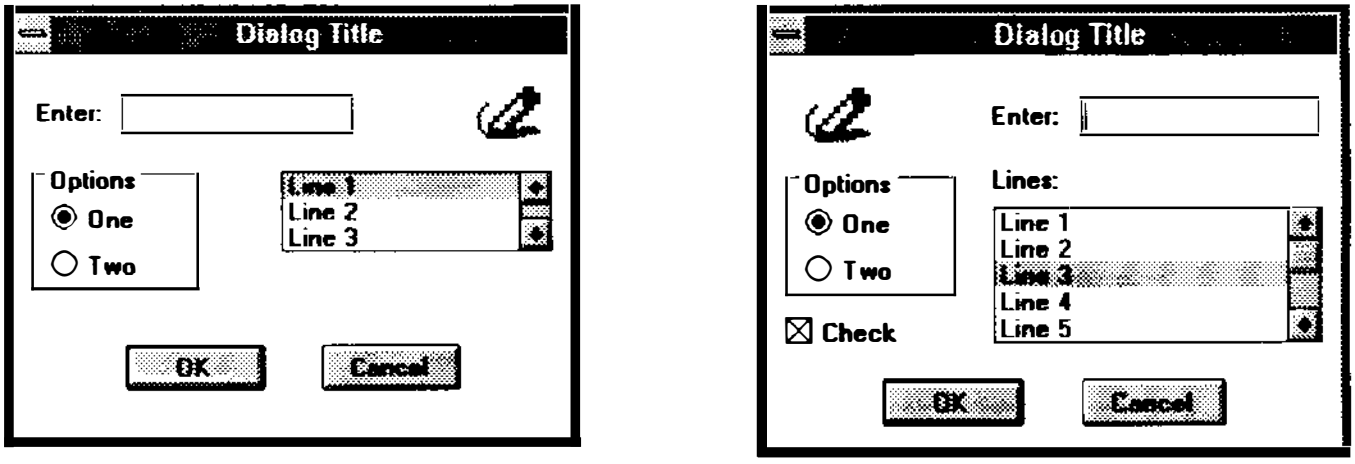


Fig. 3: *Semantically equal dialog boxes. These are usually sources of severe errors in some traditional C&P approaches.*

4 During the capture, a considerable number of patterns (visual - i.e. 2D, textual - i.e. 1D, etc.) are collected and associated with the entities (menu, button, window, mouse motion, etc.) of the application under test. In this way, in the playback phase, the presence of these patterns can be verified. Consider a window-based application of medium size. During the capture phase, it can be very easy to have to collect 100-200 patterns. The tool maintains this information by generating a script file describing the history of the capture. This script file is substantially the same file that will

be used for testing the application during the playback. The information collected is very difficult to reuse for testing other applications even if the patterns are the same of those of many other applications. However, the reusing of this information could shorten the time of capturing. This fact is more evident considering that: (i) a company usually tends to reuse the software during the application development (for example, the dialog box for selecting files); (ii) most of the companies follow standard methods for building user interfaces in the sense that the semantic of certain visual entities and actions are standardized (position of certain selections on the menu, adoption of certain combinations of keys for closing an application, selecting an item in a list etc.); (iii) there exist international standards for the definition of the actions allowed on user interfaces;

5 Though script files can be automatically generated, a manual capture is often mandatory. This is very difficult since the syntax and semantics of the script language are too closely related to the low level details of the visual aspect of the user interface of the application. This means that it is impossible to generate the script knowing only the generic behavior, the main structure of the script file for testing could be generated from the early phases of application development.

These problems are mainly due to the fact that, traditional script languages are usually strictly related with the low-level details of the application, such as the position of graphic elements, the patterns to be searched, etc. Moreover, the script languages are also too procedural-oriented and, thus it is difficult to maintain the structure of the application. This means that the scripts do not model the relationship among the visual entities and other events which can appear on the screen, and the relationships between a window and its menu, items and subitems, or between a button and the possible mouse actions which can be performed on it, and are not described by using the traditional script languages for C&P. In addition, if the information related to the visual entities (e.g., pattern, buttons, icons) and the corresponding actions (which can to be performed on an entity) are maintained in a unique structure, then a powerful mechanism for reusing the old tests is obtained. This concept is in accordance with the object-oriented paradigm, by which the single entities of an application can be modeled with a unique class containing both data and behavioral aspects [6]. The object-oriented paradigm (OOP) also has many other mechanism which are very useful for modeling the applications and for providing a support for reusing the visual and textual entities of the applications together with their allowed operations (i.e., entity behavior) [9].

In order to define a fully object-oriented system for Capture and Playback, both hardware and software components of the SUT must be in accordance with the object-oriented approach. As regards the hardware components, in the non-invasive testing systems, a board to grab and analyze the video screen, in real time, is usually present. The screen patterns grabbed during the capture, are compared in the phase of playback with specific areas of the current screen in order to *verify* their presence.

In order to guarantee the real-time testing, the verification is directly performed by the hardware. It allows the verification of the presence of given patterns at given image coordinates. This is a strong limitation because, for slight modifications of the screen of the SUT, the process of verification fails. For example, a simple displacement of a gadget (i.e., buttons, icons, text, etc.) in a dialog box is confused with the absence of the gadget. This problem can be circumvented by software, reiterating the process of verification in each pixel of a given area, thus searching for a pattern in a given sub-frame of the screen. It should be noted that a software-based solution of this problem is completely unfeasible if real-time testing is required, thus a hardware real-time search is mandatory. In particular, dedicated hardware, based on programmable gate array and fast correlators, has been developed by ASIC. That hardware is capable of searching image patterns, in real-time, in the whole screen.

Therefore, the hardware supports the object-oriented management of the problem. In fact, if a dialog box D1 contains a certain gadget G1, its presence must be recognized directly by asking the hardware where G1 is in D1, Once G1 has been found, then a requested operation on that gadget can be sent directly to that gadget. Therefore, the instruction used to test G1 is semantically reduced to **test G1 of** D1. In this way, the script files can be independent to the visual position of the visual elements on the screen. Only the relationships among entities are needed. In this case, the script file can describe the testing procedure at a higher level that traditional script files.
The object-oriented paradigm is present in the software architecture at various levels. In particular:

1 For defining a Language Object-Oriented for Testing (LOOT). The language describes at a high level the operations which must be performed to the test application. It has also the capability of describing the test at various levels of abstraction, in order to reach the final version for refinements, possibly during the application development. In addition, it is capable of defining new entities and reusing already defined ones.

2 For the definition of the tool for C&P. Being the tool itself is object-oriented, a high confidence is conferred to the system since most of the primitives which are available at the level of the language are also inside the tool for generating the script files in the LOOT language.

3 For the modeling of entities which are used for testing the application. These are also stored into an object-oriented database. For each entity (button, dialog box, icon, etc.) both static and dynamic descriptions are stored. Static descriptions correspond to the visual entity pattern, while dynamic descriptions are the possible operations which are allowed on that entity.

This allows the reduction of Capture time by reusing objects and classes and by beginning to specify the tests of the application even if the final visual aspect of the application is not available. The reduction of Capture time is obtained by reusing already defined entities. The reuse can be obtained at various level of abstraction, thus it results in both time saved for capturing patterns, and/or time saved for describing the procedures of testing of already tested parts of applications, such as dialog boxes, etc.

3 Testing Tool Architecture and Application Modeling

The testing tool includes a real-time object oriented kernel supporting the concurrence at various levels. With this kernel, the concurrence can be at an equivalent level for both objects and methods of the same class. This is indispensable since the testing tool must be capable of keeping under control several SUTs at the same time (see Fig. 1 in Sect. 1). The emulation boards are connected to the HOST by means of a dedicated network. Through this network, asynchronous messages can arrive at several SUTs thus the kernel must be capable of reacting in real-time to these stimuli.
Strict deadlines for stressing distributed applications are usually imposed. Therefore, the kernel must be characterized by real-time capabilities. To this end, the object-oriented kernel of the tool CASE TOOMS [2] (i.e., the kernel of the real-time language TROL [3]), has been adopted. It consists of (i) a low level interface between the object-oriented system of classes and the operating system (OS/2 2.1, or SUN Solaris 2.3), and (ii) a set of classes like Thread, Path, Temporal, Constraint, Clause, etc.

In order to provide the basic elements for modeling the application under test around the real-time kernel, a lot of classes have been defined, such as Pattern, Font, Button, Keyboard, Mouse, Screen, Window, Dialog boxes, TextInput, Text, Icon, OnOffButton, RadioButton, ScrollBar, Menu, etc. These classes have been organized in a specialization hierarchy exploiting the object-oriented capabilities of inheritance and polymorphism (see Fig. 4). Among these classes, elementary drawable objects have been also modeled, to provide a support for modeling more complex graphic entities.
Observing the hierarchy reported in Fig. 4, it should be noted that, the class hierarchy proposed is quite different with respect to other hierarchy used for modeling graphical user interfaces endowed with windowing systems, such as CommonView, Zinc++ , MS Visual C++, etc. This is due to the fact the hierarchy proposed models the graphic environment from a different point of view, which is the testing of the applications by manipulating the visual entities from outside, rather than visualizing the entities for representing certain actions for getting certain information as in windowing systems.
The main objects of the testing system are instantiated from the classes specialized from the ExtEvent class, which are: Keyboard, Mouse, Time and Screen. These are the classes which model the physical entities of the application under test. Therefore, for each application there exists one object for each of these classes. Note that, these classes are internally concurrent and, thus, are capable of satisfying asynchronous requests with respect to the current operations, such as managing exceptions.
In this hierarchy the class Application is also reported, and, from this, classes, such us Window, ScreenTextual, etc., have been specialized. This means that an application can be either built on a graphical user interface (endowed with a windowing system) or on a classical ASCII interface. According to the OOP, the specialization mechanism defines that subclasses inherit both Attributes and Methods from the superclasses. Therefore, such inherited features are not reported in the definition of specialized classes:

```
Class root
{
Attributes:
        char *Name;      // object's name
        .....
Methods:
        Set_Name (char n[]);
        char * Get_Name();
        .....
}

Class Application public : root
{
Attributes:
        Dimension dim;      // screen dimension (dx, dy)
        .....
Methods:
        Open ();
        Close ();
        Dimension GetDimension()
        .....
}
```

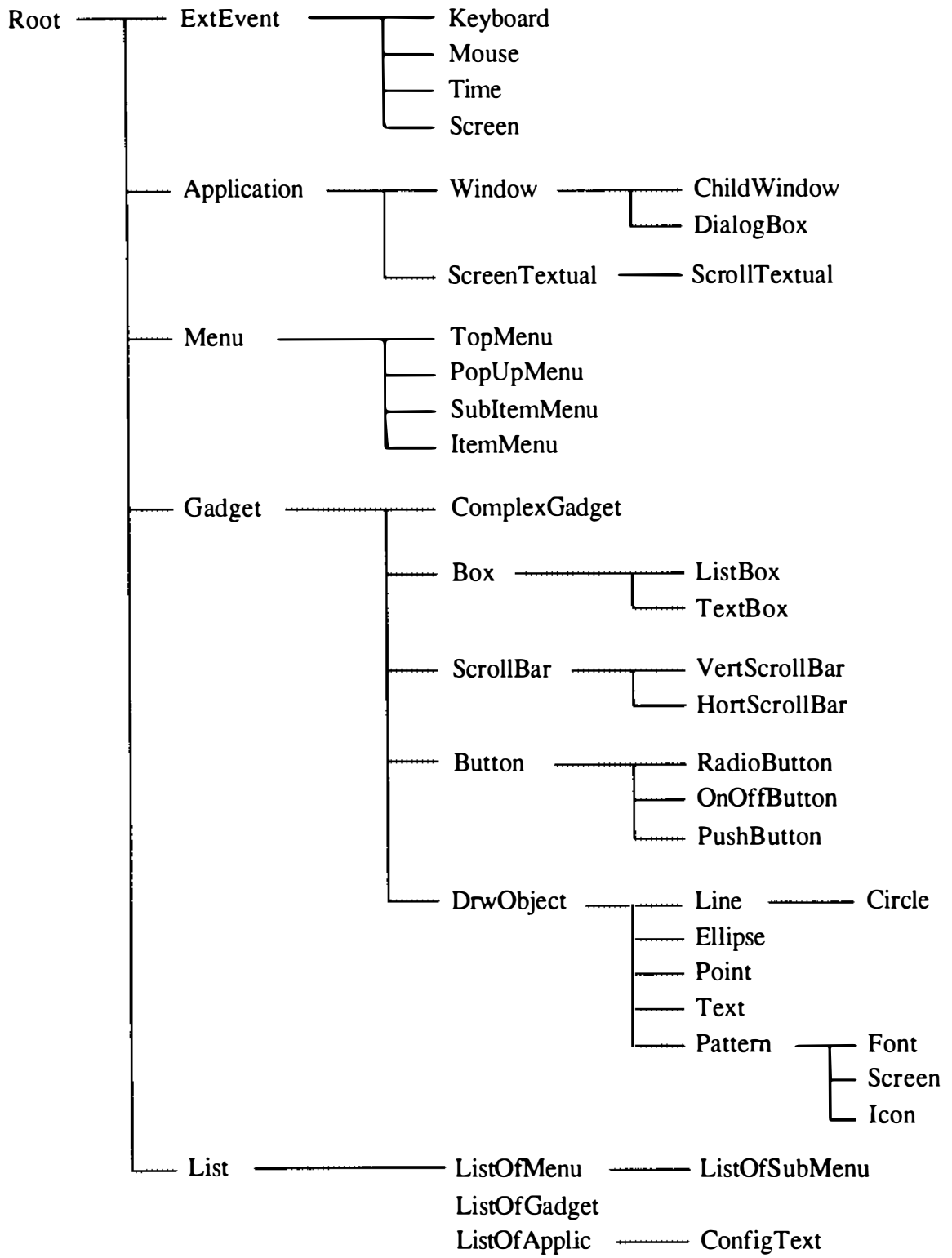


Fig. 4: *Class tree reporting the classes of the testing tool which are used for modeling the applications under test (a part).*

During the testing of a set of applications on several SUTs, the tool collects the information corresponding to each application in the instance of the class Application, which in turn are collected by an object in the class ConfigTest (specialization of the class ListOfApplication).
At the level of the testing tool, an application is modeled by specifying it structural hierarchy starting from an instance of a class of the Application sub-hierarchy. A window application is modeled as an object of class Window. This in turn contains an object of the class ListOfGadget and of the class ListOfMenu, etc. Therefore, by using polymorphism, instances of the sub-hierarchies Gadget and Menu can be collected in the corresponding lists, respectively:

```
Class Window public : Application
{
Attributes:
        Pattern id;         // pattern of identification
        ListOfMenu TheMenu;
        ListOfGadget TheGadget;
        Position pos;       // position on the screen
        ..............

Methods:
        Window (Pattern);
        Set_id (Pattern);
        Pattern Get_id ();
        Localize ();        // Set position "pos" by calling hardware facilities
        Position Get_Position ();
        Move (int, int);
        Size (int, int);
        Add_Gadget (Gadget * );
        Add_Menu (Menu *);
        Remove_Gadget (Gadget *);
        Remove_Menu (Menu * );
        ..............
}

Class ChilWindow public : Window
{

Attributes:
        Window * father;
        ..............
Methods:
        ..............
}
```

Classes belonging to the Window sub-hierarchy have the attribute id which is the pattern of identification for the window. This allows the testing tool to find the position of the window on the screen by calling the image grabber board inside the method Localize ().
The ListOfGadgets (TheGadgets) maintains the structural relationship among the elementary components belonging to the window and the application itself, while the ListOfMenu describes the menu facilities of the Window. Each menu may be logically related or not to the opening of a DialogBox or to a ChildWindow. This information is maintained at the level of menu by means of Pointers. The referenced DialogBox or ChildWindow are in turn defined in the same way. In this way, the application is hierarchically described inside the testing tool. It should be noted that the structural description is independent of the low-level details comprising the details about the application behavior. Low-level details, such as the position of a window or the presence of a particular pattern, can be automatically recovered or added later).

Therefore, the hierarchical structure of an application can be described long before the delivery of the final version of the application itself. In particular, it can be done in the early phases of the software life-cycle. The description of an application hierarchy can be saved into the object database for its future reuse. The same mechanism is performed to describe the structure of Dialogbox, therefore the future reuse of these complex entities is strongly facilitated with respect to a procedural approach of testing tools. Analogous mechanisms are used to define complex gadgets by using a set of simple Gadgets This is made possible by means of the class ComplexGadget which is a Gadget containing an object of the class ListOfGadget.
The specific details about the applications structure and behavior are described only when the final version of the application under test is available. It should be noted that two different applications presenting the same structural hierarchy having different detailed behavior can exist. In addition, two versions of the same application can have the same structural description while presenting changes of behavior. Therefore, these two aspects, static (structure) and dynamic (behavior) are managed in two different ways in the testing tool, but both can be reused and maintained in relation to each other to ensure congruence.
The hierarchical organization is the main reason for the decrease of Capture time, allowing the reuse of previously captured and described parts of applications (e.g., DialogBox, ChildWindow, Button, Icon, etc.). As it has been pointed out, this organization also allows the specification of the application in the early phases of the application development.
In order to make easier the building of the structural and behavioral descriptions of a whole Application, as well as those of DialogBox, Button, etc., these descriptions can be specified by means of a formal language called LOOT (Language Object-Oriented for Testing). This language allows the description of the instances of the leaf classes of the hierarchy presented in Fig 4. For example, the details about the attribute values of a ListBox, id (pattern of identification), dimension, color, etc. can be specified.
Moreover, in order to confer a high degree of flexibility to the language, new classes as conceptual specializations of window, ChildWindow, DialogBox, etc. can be defined as depicted in Fig 5.

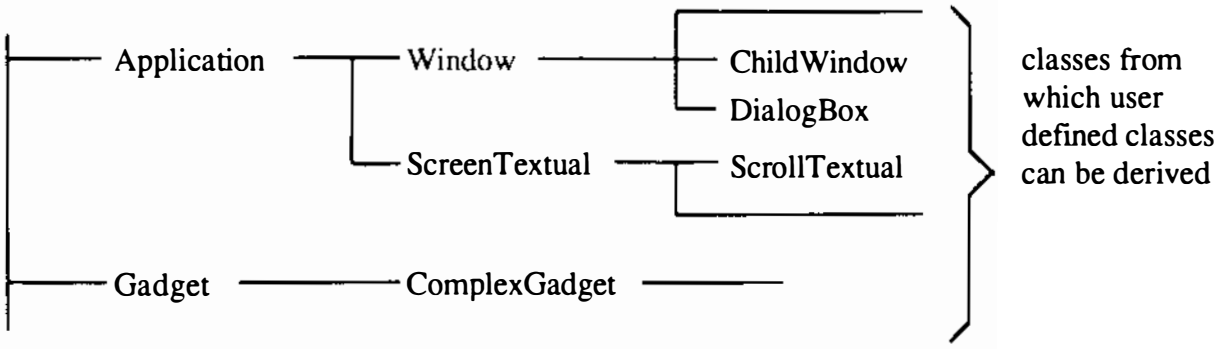


Fig. 5: *Classes which can be conceptually specialized from the user*

From the point of view of the testing tool, the user derived classes are sort of templates which in turn can be further specialized. For example, see Fig. 6 (such as in TOOMS/TROL [2], [10]). These user defined classes are hierarchically organized and stored into the object databases. Therefore, the specialization among the user defined classes improves the reusability of the defined applications. These user defined classes also support the process of instantiation. This allows the description of very complex applications, with minimal effort.

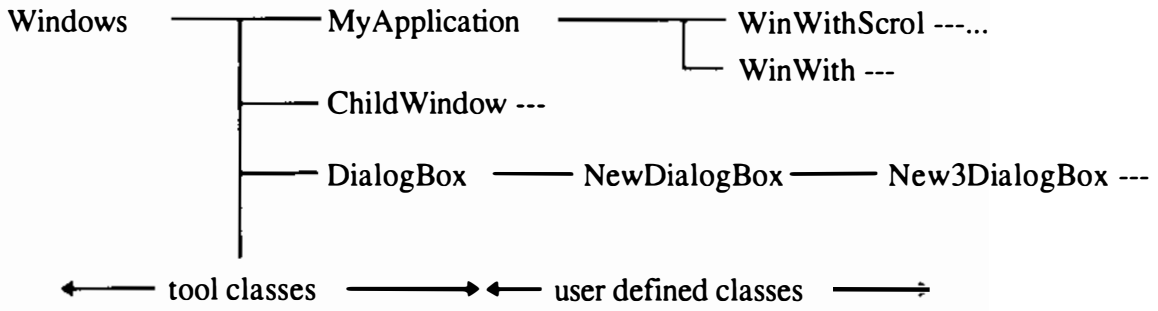


Fig. 6: *Example of conceptual specialization of user defined classes, user sub-hierarchies*

For the user defined classes, specific methods, called operations, can be defined. These operations can be used to describe the elementary actions which can be performed on these entities in order to test them.
In this way both structural and behavioral aspects of the user defined classes can be stored in a single chunk and reused in future.
It should be noted that, many standard windowing systems, such as MS-Windows, PM OS/2, Motif, etc. have the capability of defining some visual details of the application by using particular resource languages, such as the Resource or User Interface Language (UIL). In the future, the testing tool proposed will be able to convert this information for building classes inside the testing tool. This will further reduce the capture time.

4 Language Object-Oriented for Testing (LOOT)

The object oriented paradigm allows the definition of the structure and behavior of new abstract objects by means of the concept of class and method. According to this paradigm, classes describing the entities under test must belong to the testing procedure and, the so called elementary operations of test belong to classes as discussed in the previous section. This point of view transforms the traditional concept of a sequential script file in which the low level details are distributed with the high-level ones and in which the behavior of each entity cannot be clearly identified and reused.

Following this new point of view, the script file is transformed in a sequence of class declarations followed by a very short sequence of high-level operations. This is due to the fact that the details about the test of the major entities (e.g., windows objects, dialogbox objects etc.) are encapsulated inside the operations of the classes describing them. These in turn use the operations defined for the smaller objects (e.g., Buttons, Icons, etc.) during their test.

For these reasons, LOOT allows both the definition of new classes (conceptually derived from the testing tool classes as depicted in Fig. 5 and Fig. 6) as well as the instantiation of objects from these classes and from the elementary classes of the testing tool (see Fig. 4). The definition of a new class is performed by describing both the structural (internal data, i.e., attributes) and behavioral (allowed operations) aspects of the class. For example the definition of class NewDialogBox, comprises two TextBox and two PushButton operations and is conceptually derived from the class DialogBox, and it is specified as:

```
Class NewDialogBox specialization DialogBox
Gadgets:
        Name, Surname :  TextBox;
        Close,  Cancel    :  PushButton;
Operations:
        NewDialogBox (na : String, su : String, cl : PushButton, ca : PushButton)
                {    // operation of instantiation
                Name. Set_id(str2pattern ( na ));     // identification set
                Surname. Set_id ( str2 pattern (su ));     // identification set
                Close= cl;
                Cancel= ca;
                }
         WriteText ( t1 :  String,  t2 :  String)
                        {
                        Name.Write ( t1);    // call the method Write () of class TextBox
                        Surname. Write (t2);
                        }
         ReadText (t1 : String, t2 String)
                        {
                        Name.Read (t1 );    // call the method Read () of class TextBox
                        Surname.Read (t2);
                        }
         CheckText (na: String, su: String)
                        {    // checking the presence of known Name and Surname
                        tmp1, tmp2: String;    // temporary objects
```

```
                    Name.Read (tmp1) ;
                    Surname.read (tmp2) ;
                    if (tmp1= =na and tmp2= =su) return (1);
                    else return (0);
                    }
        Close()   // close the dialog box, by clicking on the icon Close, saving strings
                    { Close.Click ();   // call the method Click of class PushButton }
        Cancel()  // close the dialog box, by clicking on the icon Cancel, without saving
                    { Cancel.Click ();  // call the method Click of class PushButton }
        .....
        .....
        .....
Events:
        .....
        .....
end;
```

The above class inherits from the class DialogBox the standard operations such as Move(), Size(), Get_Position(), Localize(), etc. (most of these are directly inherited from the class Window). Among the user defined operations, CheckTest () has been defined in order to verify the presence of known strings in the corresponding TestBox. The class definition for describing the structure and the elementary operation of an Application, can be defined in the early phases of the software life-cycle. In fact, frequently structures of the dialog boxes are the first things to be defined.
After a class definition, instances of this class can be defined and used (see Fig. 7). On these instances, both the operations inherited (defined in the fundamental classes of the testing tool) and defined in the class description itself, are allowed. The process of instantiation consists of describing the details of the class attributes (i.e., Gadgets and Menu). This is performed by following the rules defined in the operation of instantiation of the class, for example:

```
CANCEL (Patt_ca), CLOSE (Patt_cl) : PushButton
. . .
A_NewDialogBox ( "Name" , "Surname", CLOSE, CANCEL ) : NewDialogBox
```

Where the objects CLOSE and CANCEL are two already defined PushButton operation with their patterns, and which could be used in many other dialog boxes.

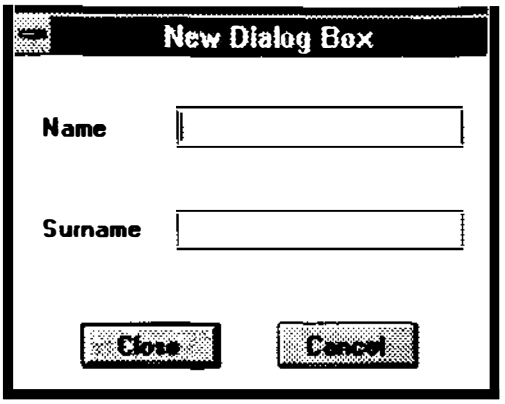


Fig. 7: *Visual aspect of an object instantiated by the class NewDialogBox*

More complex classes can be defined by means of a process of specialization, such as the class New3DialogBox which has been obtained by specializing the class NewDialogBox (see Fig. 8):

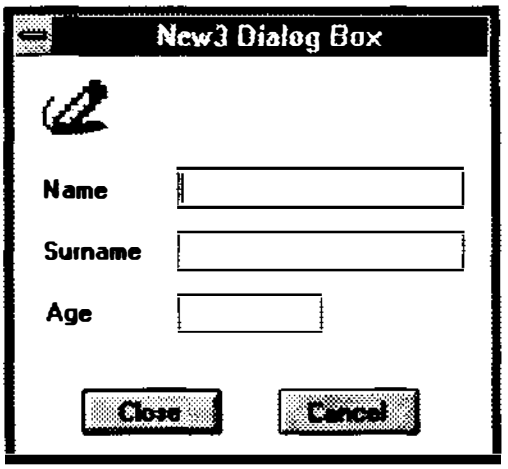


Fig. 8: *Visual aspect of an object instantiated by the class NewDialogBox.*

```
Class New3DialogBox specialization NewDialogBox
Gadgets:
        Age : TextBox;    // new gadget
        pat  :  Pattern;    // constant pattern
Operations:
        New3DialogBox (na : String, su : String, ag: Integer,
                        cl : PushButton, ca : PushButton)

        {  // operation of instantiation
           Name.Set_id (str2pattern (na));          // identification set
           Surname.Set_id (str2pattern(su));        // identification set
           Age. Set_id (integer2pattern(ag));       // identification set
```

```
        Close= cl;
        Cancel= ca;
        pat= str2pattern("New3DialogBox");    // static initialization
        pat.Set_Position (30, 45);            // static initialization
    }
    ......
    ......
Events:
    ......
    .....
end;
```

This new class inherits from the class NewDialogBox all its Gadgets, operations and Events. According to OOP Operations, Gadgets and Events (that will be discussed later) can be overwritten and overloaded respectively, following the rules of monotonic inheritance. This specialized DialogBox has been obtained by adding two new Gadgets: **Age** which is a TextBox; and the Pattern **pat**.
In Fig. 8, an example of object instantiated from that class is given.
It should be noted that it is only a particular case that the instances presented in Fig. 7 and Fig. 8 present the same values for the attributes inherited (in the sense of dimension of TextBoxes, type of Texts associated with the TextBoxes, position of Gadgets in general, type of PushButtons, etc.). In fact, all dialog boxes having three TextBoxes, two PushButtons and one Pattern, placed in same way, could be regarded as instances of class New3DialogBox and, thus, they could be stored in the Object Database and subsequently recovered by means of a conceptual navigation.
Once the class definitions are performed, the class which represents the application under test (for example, as a specialization of class Window) and the structure of the application has also been defined.
At this point, the main script can be written describing at a high-level the testing process:

```
// class definitions
.............
.............
// instantiation
theapp: MyApplication;
tmpDB : DialogBox;
FB : FoundBox;      // window confirming the entry found and containing the
                    // the telephone number etc...
NFB : NotFoundBox;  // window affirming that the entry was not found
.............
.............

// program
theapp.Open ();
theapp.OpenFile ("MyAgend");

tmpDB = theapp. Searching();    // request for opening the DialogBox for searching
                                // it is of type NewDialogBox
```

```
tmpDB. WriteText ("Mark", "Smith");
tmpDB. Close ();

When (Screen.Appear (FB)) do
        FB.Get ();
        .....
        .....
        if (FB.Telephone ()) then
        .....
        else
        .....
        endif
        .....
        .....
        for i = 1 to 2 do /* loop */
        .....
        .....
        .....
        endfor;
        .....
        .....
        FB.OK();
endwhen;

When (Screen. Appear (NFB)) do
        NFB.Get ();
        .....
        .....
        NFB.OK ();
endwhen;

.....
.....

theapp.Close ();   // close the application "theapp"
.....
.....
```

It should be noted that, by default, a number of active instances are defined, in particular: Mouse, Keyboard, Screen, and Time. These are the only allowed instances of the homonymous classes of the testing tool. The service of these objects can be requested (by sending messages) in every position of the script and in every operation defined for the classes. For example, to move the mouse in the position (10.30), write Mouse.Move (10,30) or Mouse.Move (P1) if P1 is a Point with coordinates (10.30).

To find a pattern on the screen:

```
P1 : Position;
apattern : Pattern;
.....
.....
P1 = Screen.Find (apattern);
```

For setting a timeout, a timer, from the object Time, must be requested:

```
T1 : Timer;
.....
T1 = Time. Start (DOWN, 5);    // timer of 5 seconds
.....
When (Time.TimeOut (T1) ) do
.....
.....
endwhen;
```

In the above example, the construct When-endwhen has been used to define the actions which must be executed when asynchronous exceptions are detected. These exception handlers can also be defined inside the class body under the field Events. In this case, if an exception is redefined inside a class, and if its occurrence reaches the system during the execution of the operations of that class, then the script specified for the exception inside the class is executed instead of the global one. The exception can be defined only on the basis of the operations allowed for the *active instances*. In fact, the construct When-endwhen is defined at level of ExtEvent in the testing tool.
In the following, a part of the class definition of class MyApplication is shown. It should be noted that within the definition of class operations, the attribute variable pos is used to request the absolute displacements of the mouse:

```
Class MyApplication specialization Window
Menus:
        TheTopMenu : TopMenu;
Gadgets:
        VS : VertScrollBar;
        HS :  HortScrollBar;
Operations:
        MyApplication (.....)
        {
        .....
        .....
        .....
        }
        Point GetVertScrollBarPosition () { return (VS. GetPosition ()); }
        Point GetHortScrollBarPosition () { return (HS. GetPosition ()); }
        SetVertScrollBarPosition (p: Point)
```

```
        {
        Mouse.Move (pos+p);
        Mouse.ClickLeft ();
        }
SetHorScrollBarPosition (p: Point)
        {
        Mouse.Move (pos+p);
        Mouse.ClickLeft ();
        }
SelectAMainItemMenu (value : String)
        {
        p: Point;
        p= TheTopMenu.WhereIs (value);
        Mouse.Move (pos+p);
        Mouse.ClickLeft ();
        }
SelectASubItemMenu (itemvalue: String, subitemvalue: String)
        {
        p, p2: Point;
        im: ItemMenu;
        im= TheTopMenu.WhoIs (itemvalues);
        if (im.HasSubItem () ) then
                p2=im.WhereIs (subitemvalue);
                Mouse.Move (pos+p2);
                Mouse.ClickLeft ();
        endif;
        }
        .....
        .....
Events:
        .....
        .....
end;
```

In addition to what has been shown in the previous examples, the constructs of Repeat-until, While-do-endwhile, Do-while-endwhile, and Switch-do-case-else-endswitch are present in LOOT. Moreover, the definition of procedures and functions is also allowed.

In the LOOT language, there is a particular construct to specify which operations must be performed on the different SUTs connected to the dedicated network. This is particularly important in testing distributed applications. With the construct OnNode, the part of LOOT code which must be executed on a set of SUTs is specified.

For example, with:

```
OnNode (n1, n2, n3)
{
.....

Where (Screen.Appear (.....)) do
.....
endwhere;

.....
}
OnNode (n4)
{
.....
.....
}
```

it is specified that the LOOT code reported inside the first pair of braces is executed on the SUTs named n1, n2, n3, while the other is executed only on SUT n4. Obviously, the presence of a given SUT in the construct is mutually exclusive. In the body of an OnNode construct, the construct Where is used to define little changes of the test behavior depending on the occurrence of particular exceptions. This can be useful if the hardware of the declared SUTs are different or to test particular working conditions.
In the Capture phase, the testing tools help the user to define the classes and to inspect the Object Database for reusing already defined classes and objects. Presently, the class definition is textually performed, but work is in progress to define a suitable visual language to help the user in its work.

5 Conclusions

An object-oriented non-invasive C&P approach has been presented. It is suitable for testing applications in distributed environments and for testing applications without modifying their run-time environment. The approach proposed is strongly based on the object-oriented paradigm at both hardware and software levels. "Hardware" in the sense that a new object-oriented image grabber board for capturing and searching patterns in real-time has been implemented. "Software" in the sense that the entire application of C&P is Object-Oriented, and a new language called LOOT (Language Object-Oriented for Testing) for describing the script sequences of testing has been developed. The language allows the definition of the structural hierarchy of the application under test, the description of application behavior, and the specification of test procedures. All these descriptions can be saved and reused. Also the testing tool defined is object-oriented according to the LOOT language. The main features of this new approach are (i) the reduction of testing time by supporting the reuse of old tests (in the form of programs written in LOOT language) at each level of abstraction, and (ii) the anticipation of the testing phase overlapping the last phases of software life-cycle.

Bibliography

[1] ASIC, "User's Manual Hardware Simulator", tech. rep., ASIC S.r.l, Via S. Clemente, 6, 10143, Torino, Italy, 1993.

[2] G.Bucci, M.Campanai, P.Nesi, and M.Traversi, "An Object-Oriented CASE Tool for Reactive System Specification", in Proc. of 6th International Conference on Software Engineering and Its Applications (sponsored by: EC2, CXP, CIGREF, and SEE), (Le CNIT, Paris la Defense, France), 15-19 Nov. 1993.

[3] G.Bucci, M.Campanai, P.Nesi, and M.Traversi, "An Object-Oriented Dual Language for Specifying Reactive Systems", in Proc. of IEEE International Conference on Requirements Engineering, ICRE'94, (Colorado Spring, Colorado, USA), 18-22 April 1994.

[4] G.Bucci and P.Nesi, "Impiego di tecniche visuali per la programmazione di controllori industriali", L'ELETTROTECNICA rivista dell'associazione Elettrotecnica ed Elettronica Italiana, Vol.78, June 1991.

[5] B.Cox and B.Hunt, "Object, Icons, and Software-ICS", Byte, pp.161--176, Aug. 1986.

[6] A.DelBimbo and P.Nesi, "Blackboard-Based Concurrent Object Recognition Using and Object-Oriented Database", in Proc. of the IEEE International Phoenix Conference on Computers and Communications, IPCCC'92, Scottsdale, AZ, USA, pp.172--180, April 1-3 1992.

[7] D.Hu, "Object-Oriented Environment in C++ -- A User-Friendly Interface". 14th Ave. Portland, Oregon, USA: MIS Press, Management Information Source, Inc., 1990.

[8] M.A.Linton, J.M.Vlissides, and P.R. Calder, "Composing User Interfaces Using InterView", IEEE Computer, Vol.22, pp.8--22, February 1989.

[9] B.Meyer, "Eiffel: a Language and Environment for Software Engineering". Prentice-Hall, Englewood Cliffs, 1988.

[10] P.Nesi, "An Object-Oriented Language and Compiler for Reactive Systems", tech. rep., RT 3/93 Dipartimento di Sistemi e Informatica Facolta di Ingegneria, Università di Firenze, Florence, Italy, 1993.

[11] M.A.Tarlton and P.N.Tarlton, "Pogo: A Declarative Representation System for Graphics", in Object-Oriented Concepts Databases and Applications (W.Kim and F.H.Lochovsky, eds.), pp.151--176, New York, USA: Addison-Wesley Publishing Company, ACM Press, 1989.

A Risk-Based Approach to Regression Testing

James S. Collofello
Mohammed Fitri Abdullah
Computer Science and Engineering Department
Arizona State University
Tempe, Arizona 85287-5406
(602)965-3190
collofello@asu.edu

Abstract

Regression testing has been receiving increasing attention from both academia and industry. This paper introduces the principles of risk management into the field of regression testing. One fundamental problem in regression testing today is balancing the need to test changes (change testing) as well as to test to make sure that the system still meets its requirements (confidence testing). A selective regression test strategy is proposed which tries to address this dilemma by creating a prioritized list of test cases based on risk. The focus of this paper is on system level regression testing although the approach is applicable to other facets of testing.

Keywords: testing, regression testing, risk, metrics

Biographical Sketches: Jim Collofello is a Professor in the Computer Science Department at Arizona State University. His research interests include software testing, software quality assurance, software process management and software maintenance. During his 15 years at ASU, Dr. Collofello has authored over 50 publications in the software engineering area and worked on numerous research contracts with local industry. Dr. Collofello has also been a quality assurance consultant for several large corporations.

Mohammed Fitri Abdullah was a graduate student in the Computer Science Department at the time of this research.

1.0 Background

Software maintenance is an error prone endeavor. Collofello and Buck [4] in their investigations found approximately 50% of the errors uncovered in maintenance are errors introduced courtesy of maintenance activities. Regression testing is the final validation process in software maintenance prior to the release of a software product. It involves testing the modified system to make sure the introduced changes are correct and have not introduced any side-effects in order to reestablish our confidence in the system. The IEEE definition [9] for regression testing is:

"Selective retesting to detect faults introduced during modification of a system or system component, to verify that modifications have not caused unintended adverse effects, or to verify that a modified system or system component still meets its specified requirements."

It is clear from the above IEEE definition that the focus of regression testing is on both *change testing* as well as *confidence testing*. One of the fundamental problems in regression testing today is balancing the need for *change testing* as well as *confidence testing*

Ideally, all previous tests should be run during regression testing. This, however, is not a cost effective strategy. Hence, several strategies have been proposed for regression testing [5,6,7,8,10]. All of the proposed strategies advocate selecting a subset of previous tests based upon some set of criteria as well as the new features.

A fundamental issue that has been omitted in current regression testing research is the realities being faced by software practitioners in the industry. Issues like meeting market windows and deadlines, as well as the ever increasing shortages of resources have forced us to perform regression testing smarter. A strategic approach is required in industry to take into account these real-time issues. A risk-based approach is proposed in the next section to address the regression testing requirements as well as balance these realities.

Before embarking on the proposed risk-based approach, it is worthwhile to analyze current regression testing activities. Figure 1 provides an overview of the typical regression test process. From the modified system, we perform ripple effect analysis to evaluate the impact of the introduced modifications. We then update our tests to be consistent with the modifications. The next activity consists of selecting the tests to be executed. Test selection currently is based upon the modification information, the software under test, and what is termed as "testers intuition". This is inadequate considering the additional information available at this point. Retesting activities consist of actually performing the testing, analysis, and error reporting tasks. Test requirements describe the output goals we are striving for in a testing effort while constraints are the limiting factors being faced in a testing effort. The test requirements and constraints are used for establishing the entry and exit requirements.

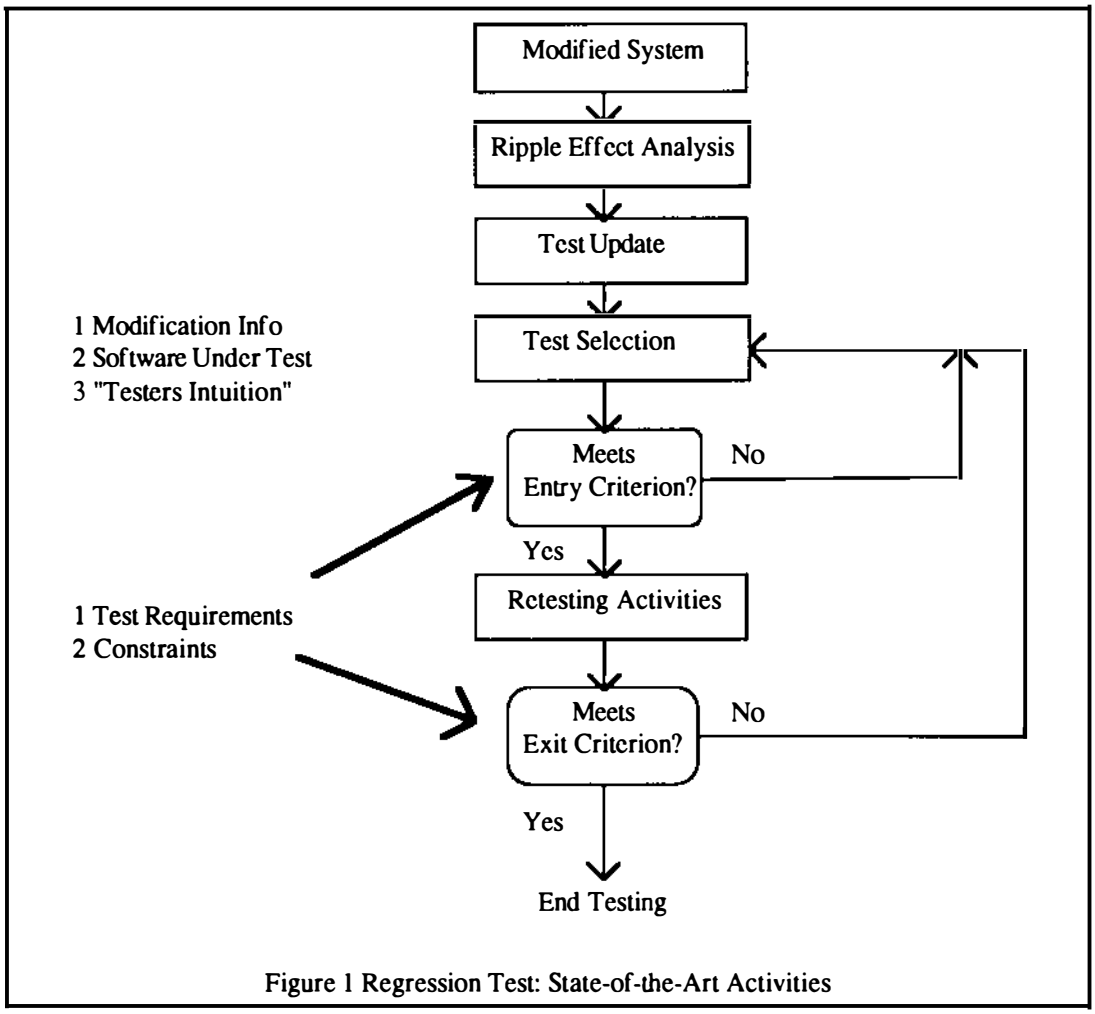


Figure 1 Regression Test: State-of-the-Art Activities

In order to understand our risk-based approach, it is necessary to define risk. Webster's dictionary defines "risk" as "the possibility of loss or injury." In regression testing, "risk" can be defined as "the probability and impact of potential problems to the system". Boehm [3] defines the concept of "risk exposure" (also called "risk factor" or "risk impact") to be the fundamental concept of risk management. Boehm goes on to define risk exposure (RE) by the following relationship.

$$\mathbf{RE = P(UO) * L(UO)}$$

"P(UO)" is the probability of an unsatisfactory outcome, and "L(UO)" is the loss to the parties affected if the outcome is unsatisfactory. "UO" means "unsatisfactory outcome" which may mean different things to developers, customers, users, and maintainers. From a regression test point of view, "unsatisfactory outcome" means not detecting errors in the system that may have an adverse impact on the system, customer, or both. Risk exposure is the probability of unsatisfactory outcome multiplied by the loss of the unsatisfactory outcome.

2.0 Risk-Based Regression Testing Approach

The proposed risk-based regression testing approach balances the need for *change testing* and *confidence testing*, as well as the realities being faced in the industry. Figure 2 depicts the proposed risk-based framework for regression testing. Regression test activities that are common to figures 1 and 2 are briefly explained while a more thorough explanation is given for the risk-based specific activities.

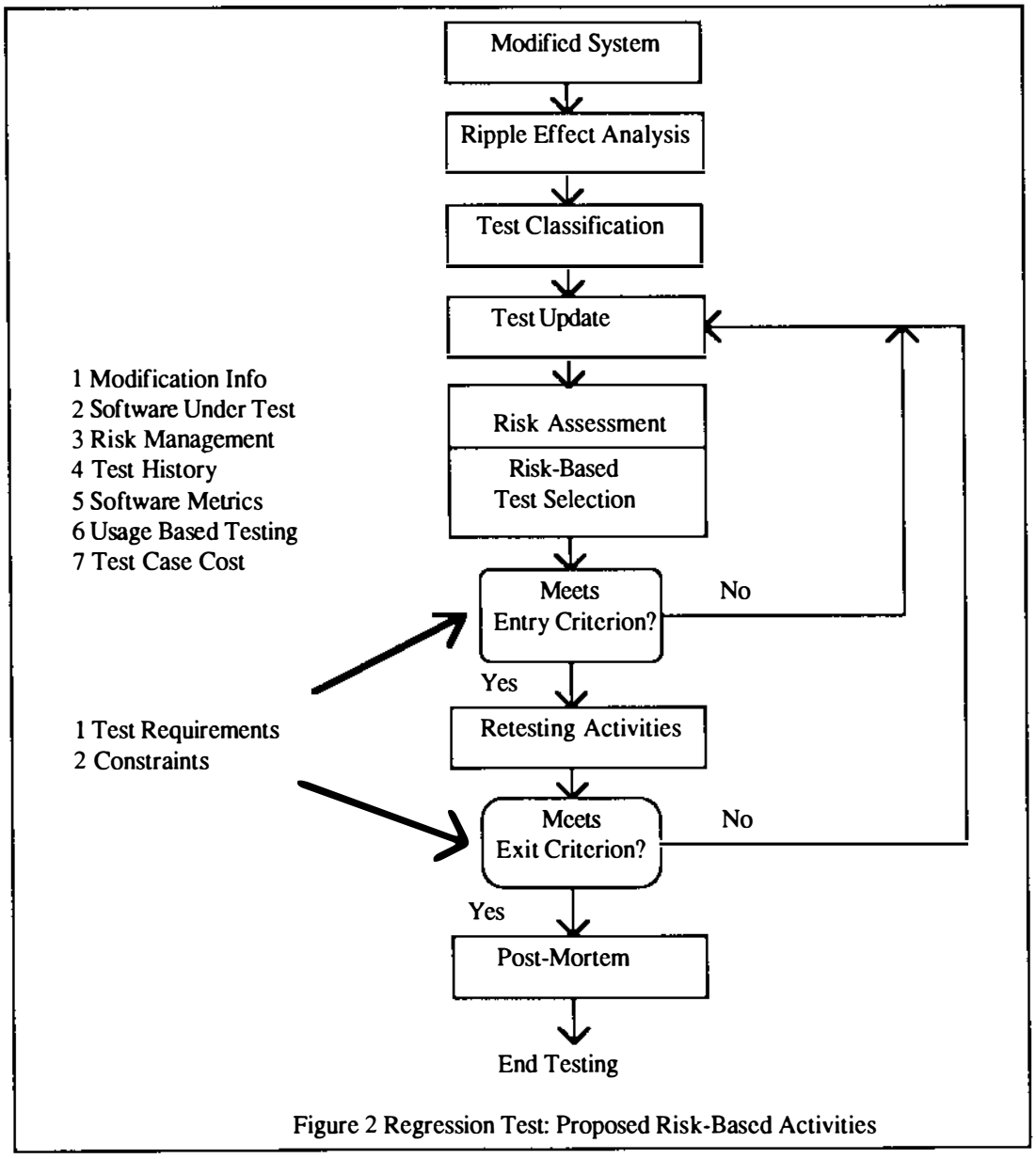


Figure 2 Regression Test: Proposed Risk-Based Activities

Modified System :

The modified system refers to the system that has undergone maintenance. It is important here to find out the type, location, and size of the modifications before commencing regression test activities. All of this information will determine how extensive the regression test effort will be.

Ripple Effect Analysis:

Ripple effect analysis is where control flow, data flow, variable set/use, and parameter passing analysis is performed to find the areas affected by the modifications performed. The more thorough the ripple effect analysis, the higher the degree of confidence we will have from our regression test efforts. If we do a poor job on ripple effect analysis, we may have to increase our "confidence" test suite to make up for this deficiency. Inapt ripple effect analysis is like testing in the dark without knowing which areas of the system have been effected.

Test Classification:

Leung and White [10] described a scheme to classify test cases based upon the type of modification a system has been subjected to and its impact to the test cases. This scheme basically identifies test cases as being reusable, needing update or obsolete. It is recommended that each test case be annotated with the attributes below to facilitate the test selection process:

1) origin - functional or structural.
2) methodology - methodology applied (i.e.Boundary Value).
3) rationale - the rationale behind the test case (i.e. for functional - to test a particular feature).
4) level - unit, integration, or system.

These attributes provide a good overview of a particular test case while at the same time differentiating one from the other. Each test case also has an associated cost. This cost can be of tremendous assistance in the test selection phase when balancing test selection with the available resources. Test case cost can be broken down into the following:

1. automation - the automation level of the test case whether it is fully automated(0), semi-automated(1), or not automated(2).
2. total elapsed time (Te)- the total elapsed time in minutes to complete a test. This includes human and machine time to set-up, retest, and analyze a test.
3. total human time (Th)- the total time in minutes spent by a tester to actually set-up, retest, and analyze a test. Th time is a subset of Te time.
4. maturity - whether the test continues on upon encountering a failure (0) or stops upon encountering one(1).
5. specials - does the test require a dedicated system or network (1) to complete the test or can it be run on common use systems (0).

Test Update:

In the test update phase, all of the test cases identified as being impacted by the modification must be updated regardless of our selection strategy. Test update and test selection should be kept as two separate issues. Test case update may include modifying some test cases, deleting some, and adding some new ones as classified in the previous activity. The test case attributes and test case cost for each test case should also be modified to reflect the updates.

Test Requirements and Constraints:

Test requirements describe the output goals of the testing effort. This information is normally described in a test plan. Example test requirements may include coverage goals or MTBF objectives.

Constraints are the limiting factors that drive the whole regression test process. Example constraints include schedules and market windows that must be met for business reasons. Hence, the availability of time, personnel, and machine resources limit the retest effort that can be spent. For this reason the constraints described above can be viewed as control factors in regression testing. What is desirable is to achieve the best possible testing effort given the control factors. In other words, the maximum return on the constrained investment.

Risk Assessment and Risk-Based Selection:

These two activities are combined here due to their similarity. The test selection stage consists of deciding whether to run all of the test cases or perform selective revalidation. Leung and White [11] describe the conditions when a selective strategy is more economical than the retest all strategy. Since the proposed risk-based approach is a selective approach, this paper assumes that a decision has been made to adopt a selective strategy.

The test classification done previously supports the identification of test cases that are directly impacted by the changes. All of these effected test cases should be selected. A set of "confidence" test cases should also be selected to make sure that the unaffected areas are indeed unaffected as our ripple effect analysis may not be 100% full proof. The amount of test cases to be selected for the "confidence" test suite depends on the criticality of the product as well as the effectiveness of the ripple effect analysis. The amount of testing is also constrained by time and resources. Thus, a risk-based approach is needed to guide the test selection process.

As mentioned previously, risk exposure is the result of the probability of unsatisfactory outcome multiplied by the loss of the unsatisfactory outcome. The practice of risk management consist of two primary steps [3]: risk assessment and risk control. The proposed approach only utilizes risk assessment. Boehm [3] defines risk assessment to consist of three separate activities as shown below.

1. risk identification - produces lists of project specific risk items likely to compromise a project's success.
2. risk analysis - assesses the loss probability and loss magnitude for each identified risk item.
3. risk prioritization - produces a ranked ordering of the risk items identified and analyzed.

The risk identification activity for the proposed scheme involves viewing the lack of reexecuting a test case as a risk item. This leads to the following definition of risk-based regression testing:

"Selective retesting based upon the probability and impact of potential problems (risk), to verify that the modification of a system or system component have not

introduced faults or causes unintended adverse effects given the resource constraints."

The risk exposure of not executing a test case will be calculated as the product of the impact of loss and the probability of loss of the feature addressed by the test case.

The impact of loss of a feature can be defined as the sum of the following identified factors. Maximum values for each of the factors in their composition of the impact factor are shown in brackets. The justification for the initial factors as well as their maximum values was largely based on practitioner experience [1] and is subject to change and refinement.

1. Criticality of the feature to the product (0.4)
2. Criticality of the feature to the customer (0.4)
3. The feature usage (0.2)

The first factor deals with the criticality of a feature to the product as a whole. The second factor deals with the criticality of a feature to the customer; that is to say that if a particular feature is not operational, it will make the customers irate and may cause the company to lose money as well possibly shrinking its customer base. This is where it is observed that "old" features are considered more important than "new" features as the customers are more dependent on the "old" features to run their businesses as compared to the "new" ones. The first two factors are very closely related in that if a defective feature causes the whole product to become unusable, it also would cause customers of the product to become irate. The second factor was added to take into account the features of the product the customer perceived as critical to its operation. That is why a distinction was made for coming up with the two separate factors The third factor deals with the usage of a particular feature by customers or by the product itself. The more heavily used a feature is, the more important it is to the whole product as well as to customers.

The probability factor can be defined as the sum of the following identified factors. Again, the justification for coming up with the initial factors as well as the maximum values in the composition of the probability factor was largely based on practitioners experience [1] and is subject to change and refinement.

1. Ripple Effect (0.4)
2. Test History (0.3)
    i. Test ran in prior phases of the release? (0.15)
    ii. Test ran in prior releases of the product? (0.15)
3. Software Metrics (0.3)
    i. Error prone features for the current release (0.15)
    ii. Error prone features for the overall product (0.15)

The first factor deals with the changes introduced to the system that can cause probable ripple effects to other parts of the system. The higher the probability of ripple effect, the higher the risk probability of errors occurring in the effected features. The first factor is perhaps the determining factor on where to intensify the regression test efforts.

The second factor deals with the test history of the product. Often, the same test cases are run in integration level test as well as in system level test, except perhaps the environments may be different. Part i) of the second factor deals with this phenomenon of running the same test in the different levels of testing. If a test case

was not run in previous levels, the probability of an undetected error not caught if the test is not selected is higher compared to if the test case was run in prior levels. Part ii) of the second factor takes into account information concerning test cases being run in previous releases of the product and was brought to light by Adams [2]. Typically, in the maintenance life cycle, several releases of the product are built in-house either for correcting errors found in the previous release, to add additional features, or both. If a test case was executed for the previous release and the test passed, the probability risk factor for that test case is lower for the next release compared to that of a test case that was not run in prior releases, or an error was detected in prior releases and the test case has not been retested since.

The third factor deals with taking into account the metrics of error prone features. Part i) deals with features that are error prone for a particular release. The information can come from inspections results, and testing performed from prior levels (unit and integration). Part ii) deals with the features that are error prone for the entire life cycle of the product including errors reported in the field.
The risk exposure of not executing a test case has a maximum value of 1. The risk-based testing approach requires calculation of risk exposure for each test case. The test cases can then be prioritized based on risk exposure. The prioritized list of test cases enables one to maximize the return on investment of regression testing. The goal of this scheme is to develop a table which prioritizes the test cases with risk factors in descending order together with their associated test case cost. This facilitates the selection of high risk test cases within resource constraints. The columns for the accumulated times for elapsed as well as human time factors can be used to draw the line on which tests can be executed given the time constraints. If we have very limited time, we draw the line higher in the table, and conversely draw the line lower in the table if we have more time in our schedule for regression testing. Table I shows an example of what the end product of risk-based test selection might look like.

Entry/Exit Criteria:

Entry and exit criteria must be defined for the actual execution of the regression tests. The entry criterion consists of completing test case selection. The exit criterion for regression testing consist of satisfying the test requirements. This may involve the selection of additional test cases.

Retesting Activities:

Retesting activities consist of actually performing the testing, analysis, and error reporting tasks. All of the test cases should be performed as specified during test selection.

Post-Mortem:

As depicted in Figure 2, a post-mortem activity should be performed at the end of the regression test cycle. This helps identify activities that were done right and those that are in need of improvement. The post-mortem should be based on information about errors found in the field and, thus, missed by the regression tests. This information can be used to modify probability and impact factors and, thus, increase the effectiveness of future regression testing activities.

Table I Example of Risk-Based Prioritized List

Test Case	RE	P	L	Te	ATe	Th	ATh
1. Test#4	0.9	0.9	1.0	30	30	10	10
2. Test#9	0.81	0.9	0.9	45	75	15	25
3. Test#3	0.81	0.9	0.9	35	110	10	35
4. Test#7	0.8	0.8	1.0	60	170	20	55
5. Test#8	0.72	0.8	0.9	75	245	25	80
6. Test#2	0.63	0.7	0.9	40	285	20	100
7. Test#1	0.54	0.6	0.9	60	345	25	125
8. Test#5	0.4	0.5	0.8	40	385	15	140
9. Test#6	0.4	0.8	0.5	45	430	20	160
10.Test#10	0.32	0.8	0.4	60	490	30	160
etc.	etc.	etc.	etc.	etc.	etc.	etc.	etc.

RE -Risk Exposure/Factor
P -Probability of Loss
L -Impact of Loss
Te -Elapsed time in minutes to run a particular test
ATe -Accumulated elapsed time(minutes) of the selected tests
Th -Human time in minutes to run a particular test
ATh -Accumulated human time(minutes) of the selected tests

3.0 Some Experience With This Approach

A case study was performed on a large distributed application that had undergone maintenance with this risk-based approach with a focus on the applicability of this approach by practitioners in the industry [1]. The case study included performing the different regression test activities as specified in figure 2. It was observed that the risk-based prioritized list of test cases that was produced did in fact address the need for both *change testing* as well as *confidence testing*. The prioritized list also can be used as justification to management when the lack of resources allocated for regression testing will greatly add significant risk to the entire project.

4.0 Conclusion and Future Research

A risk-based approach is defined in this paper with a focus on system level testing although it can be applied to other facets of testing. The approach balances the needs for *change testing* as well as *confidence testing* while taking into account real-life constraints of the availability of resources in the regression test process. The proposed risk-based approach should be seen as a framework for future risk-based testing.

The initial impact and probability factors identified in this paper should be seen as baseline factors. More research is needed to estimate the impact and probability factors utilized by this approach. Other possible research areas include adopting the risk-based approach to other facets of testing (i.e. testing in the development phase).

References

[1] M.F. Abdullah, "A Risk-Based Approach to Regression Testing," Master's thesis, Arizona State University, May 1993.

[2] M.K. Adams, "The System Test Risk Assessment Measure," *Software Test, Analysis, and Review (STAR) Conference*, May 1992, pp. 1.275 - 1.300.

[3] B.W. Boehm, "Software Risk Management: Principles and Practices," *IEEE Software*, January 1991, pp. 32-41.

[4] J.S. Collofello and J.J. Buck, "Software Quality Assurance for Maintenance," *IEEE Software*, Sept. 1987, pp. 46-51.

[5] K.F. Fisher, "A Test Case Selection Method for the Validation of Software Maintenance Modifications," *IEEE COMPSAC 77 Int. Conf. Procs.*, Nov. 1977, pp. 421-426.

[6] K.F. Fisher, F. Raji, and A. Chruscicki, "A Methodology for Retesting Modified Software," *Proc. Nat'l Telecomm. Conf.*, CS Press, Los Alamitos, Calif., 1981, pp. B6.3.1-B6.3.6.

[7] J. Hartmann and D.J. Robson, "Approaches to Regression Testing," *Proc. Conf. Software Maintenance*, IEEE Computer Society Press, 1988, pp. 368-372.

[8] J. Hartmann and D.J. Robson, "Techniques for Selective Revalidation," *IEEE Software*, Jan. 1990, pp. 31-36.

[9] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std. 729-1983 (IEEE Press), 1983.

[10] H.K.N. Leung and L. White, "Insights into Regression Testing," *Proc. Conf. Software Maintenance*, IEEE Computer Society Press, 1989, pp. 60-69.

[11] H.K.N. Leung and L. White, "A Cost Model to Compare Regression Test Strategies," *Proc. Conf. Software Maintenance*, IEEE Computer Society Press, 1991, pp. 201-208.

Model-Assisted Probability-Testing: A New Approach to Operational Testing of Software

Andy Podgurski

Computer Engineering & Science Dept.
Case Western Reserve University
Cleveland, Ohio

Abstract

Model-assisted probability-testing (or MAP-testing) is a new approach to the operational testing of software, in which data about executions is collected and analyzed computationally to reveal *unusual conditions* that might cause or indicate failure [?]. Human testing effort is then be directed toward carefully evaluating "interesting" executions. Through the use of model-assisted sampling techniques, MAP-testing can provide accurate and economical estimates of software reliability as well as reveal defects. We have experimented with a form of MAP-testing based on the ideas of *stratified sampling*. It involves partitioning program executions into many strata by automatic *cluster analysis* of execution profiles. Stratified random sampling was used together with a basic stratified estimator, and this often produced significantly more accurate estimates than simple random sampling when the same overall sample size was used with both sampling designs. Stratified sampling was effective in these experiments because clustering program executions based on their dynamic control flow often had the effect of *isolating* and/or *concentrating* executions that failed.

Biographical Sketch: Andy Podgurski is an Assistant Professor in the Computer Engineering and Science Dept. at Case Western Reserve University in Cleveland, Ohio. He received his M.S. and Ph.D. in Computer Science from the University of Massachusetts at Amherst in 1985 and 1989, respectively. His research interest is software engineering methodology, especially techniques for software analysis, validation, and reuse.

An Application of Genetic Algorithms to Software Testing and Reverse Engineering

William B. McCarty
G. Thomas Plew

Azusa Pacific University
901 East Alosta Avenue
Azusa, CA 91702
(818) 815-5311 voice
(818) 815-5323 fax
E-mail: mccartyb@class.org

Abstract

Although exhaustive methods for the testing of finite-state machines have long existed, such methods are not always appropriate or feasible. We describe Eugene, a program that uses a genetic algorithm for testing finite-state machines. Because of the use of a genetic algorithm in preference to an exhaustive method, Eugene is not strictly limited to the testing of software units that can be expressed as finite-state machines. We describe the design of Eugene and present data on Eugene's performance in testing actual finite-state machines.

Keywords and Phrases: finite-state machine, software testing, genetic algorithms, reverse engineering.

Biographical: Bill McCarty has been involved in the construction of software since 1970, when he programmed in FORTRAN on the IBM 1130. He is an associate professor at Azusa Pacific University, where he teaches software engineering in the M.Sc. Applied Computer Science program. Bill is a Doctoral candidate at The Claremont Graduate School. His research interests include software metrics, software process, software testing and probabilistic algorithms.

Tom Plew is a professor at Azusa Pacific University, where he also teaches software engineering. He previously directed one of the first training centers for the blinded adult. He holds a Ph.D. in Instructional Technology from Indiana University. His research interests include artificial intelligence, expert systems, neural networks, software process and software quality.

1. Introduction

The finite-state machine (FSM) , as the most elementary of the familiar automata-theoretic formulations of computation [Minsky, 1967], holds a special place in software development. Many software units are, or could be, implemented using the FSM as a control structure model [Witt et al, 1994]. This is particularly true of modern reactive systems, where the simplicity and versatility of the FSM facilitate the construction of reliable and verifiable software components performing a variety of computational tasks.

Recent years have seen an emphasis on techniques for producing reliable software without the traditional degree of reliance on execution-based testing. Such techniques and methodologies as cleanroom software engineering, proofs of program properties, software inspections and formal methods of specification have contributed to our improved understanding of quality software development processes. However, few, if any, software development organizations have entirely abandoned the use of execution-based testing, which remains an important software process element.

This paper addresses the need for improved software testing techniques for testing software units implemented as FSMs. We apply Genetic Algorithm (GA) techniques in implementing Eugene, a program for generating test sequences for FSMs. Eugene capitalizes on the ability of GAs to efficiently explore large search spaces.

The structure of this paper is as follows. First, we survey some previous research on FSMs and GAs. We then present the design of Eugene and data on Eugene's performance. Finally, we offer conclusions and possible directions for further research.

2. Testing Finite-State Machines

Since modern computing hardware is built on the FSM model, software programs fit the FSM model as well. Often, the FSM model is made explicit in the design process. Many structured and object-oriented methodologies recommend the FSM model (e.g., [Hatley and Pirbhai, 1988], [Rumbaugh et al, 1991], [Yourdon, 89]). Protocols, concurrent systems, system failure and recovery subsystems, system configuration control, and distributed data bases are given by Beizer [1990] as typical applications of FSMs to software construction. For many other systems, a non-FSM model may be simpler to understand and manipulate, though in principle an equivalent FSM model could be constructed.

Beizer [1990] presents a discussion of FSMs and how software implemented using the FSM model can be tested. Beizer emphasizes state testing, a method which depends on knowledge of the number of states in the FSM and which aims at exercising every state transition of the FSM. Other recent work on testing of FSMs includes the work of Sidhu and Leung [1989], that of Rivest and Schapire [1993] and that of Chow [1978].

Chow [1978] presents a method for test data selection in testing software designs modeled by FSMs. Chow's approach is based on access to the FSM's design specification, i.e., it is a white-

box method. He assumes the FSM is 1) completely specified, 2) minimal, 3) starts with a fixed initial state, and 4) that every state is reachable. Chow's method is exhaustive and is analogous to path testing of ordinary software units (cf. [Beizer, 1990]).

Chow's method involves three steps: 1) the maximum number of FSM states is estimated, 2) test sequences are generated based on the (possibly incorrect) design specification, and 3) the responses to the test sequences are (manually) computed and verified. The test sequences are composed from two subsets, P and Z. P is a set of input sequences that will move the machine from its initial state to any chosen state. Z consists of input sequences that can distinguish the behaviors associated with each pair of states in a minimal FSM. By composing sequences from P that force the FSM into a selected state and from Z that assess the correctness of the FSM, each state of the FSM can be verified. Verification of the FSM consists of verification of each of its component states.

Chow estimates the bound on the number of sequences required to fully test an FSM as $n^2 \times k$, where n is the number of states in the FSM and k is the number of input symbols, i.e., the size of the input alphabet. The bound on the total length of all test sequences is given as $n^3 \times k$. Chow points out that, in an average case, the number of test sequences can be lower.

Chow reports the application of his method to three actual systems. One of the systems required some transformation to reduce the associated design into a FSM. Chow's results are summarized below.

Table 1. Chow's FSM Test Cases

FSM States	FSM Symbols	Test Sequences Required	Average Test Sequence Length
7	18	400	4
5	4	48	3
11	5	135	4

In summary, Chow's method requires that the design to be verified be expressed as an FSM, or at least be transformable into an FSM. It is a white-box method geared to verifying designs, rather than software units. Where the necessary assumptions obtain, the results given by Chow's method are provably correct.

In section 4 of this paper we present a different approach to testing software units, rather than designs, based on the FSM paradigm. Our method is probabilistic, rather than provably correct, but it is relatively efficient, easy to implement and is not strictly dependent upon the existence of an FSM model of the control structure of the software unit. First, however, we present some necessary background material on GAs.

3. Genetic Algorithms

Genetic algorithms (GAs) mimic biological systems in using the principles of population diversity and natural selection to explore large search spaces. GAs have been surveyed by Michalewicz [1992] and by Koza [1993]. Both authors also present an extension to GA, known as genetic programming, wherein GA techniques are applied to the construction of computer software. Genetic programming is one form of program induction, the process of searching the state space of possible computer programs for a program that produces a given desired output when presented with a given input.

GAs have been applied to a wide variety of problems. Koza (ibid.) surveys a number of such applications including optimal control, planning, sequence induction, symbolic regression, automatic programming, strategy discovery, empirical discovery and forecasting, symbolic integration and differentiation, discovery of inverses and identities, induction of decision trees and evolution of emergent behavior. A recent issue of *IEEE Expert* focused on GAs, including reports of GA research by Schultz, Grefenstette and De Jong [1993] and by Uckun, Bagchi, Kawamura and Miyabe [1993]. Common to these applications is a search space so large that the so-called combinatorial explosion precludes effective use of traditional algorithmic techniques.

GAs operate by iteratively transforming a population of individuals in such a way that desirable properties, measured by a fitness function, become progressively prevalent. At some point the properties are deemed adequate and the computation is terminated. However, the computation proceeds probabilistically and there is no guarantee of convergence.

Individuals are represented by chromosomes that encode their properties. Chromosomes may be simple bit strings or more complex data structures such as trees. The elementary components of chromosomes are termed genes and are commonly represented as integers or floating point numbers. The fitness function associated with the GA maps a chromosome's value into a number, the chromosome's fitness, that measures the desirability of the chromosome's properties.

A GA has associated with it one or more transformation functions that map chromosomes into new chromosomes. Common transformations include mutation, which alters the value of a gene, and permutation, which alters the sequence of a chromosome. Biologically, such operations would be considered asexual, since only a single chromosome is involved.

An important class of transformation function is the crossover function, which maps a pair of chromosomes into a new chromosome. Typically the resulting chromosome takes its first n genes from one "parent" and the remaining m genes from the other "parent." GA chromosomes may be restricted to some particular length (i.e., number of genes) or their length may be allowed to vary. Usually, to facilitate implementation, some fixed upper limit is chosen.

Beginning with an initial population, usually generated randomly, GAs successively operate on the chromosomes of the population using the transformation functions, such as mutation, permutation and cross over. Each pass over the population is termed a generation and results in the establishing of a new population. In a typical generation, selected chromosomes with low fitness

are removed from the population. Others are copied from the old population to the new. Still others are subjected to one or more transformations and the resulting chromosome is inserted into the new population.

Usually chromosomes are eligible to participate in transformations according to a probability that is proportional to their fitness. Over time the average fitness of the population is intended to improve and the fitness of the best individuals is intended to achieve some threshold level. When the threshold level is reached, the computation terminates.

In the following section, we present Eugene. Eugene is a software program for testing or reverse engineering of FSMs. Eugene uses a GA to explore the state space of the FSM, which is usually quite large.

4. Eugene: a GA for Testing FSMs

Eugene is a GA for testing or reverse engineering FSMs. Eugene requires that the FSM to be tested is accessible by a subroutine call. In addition, Eugene requires that 1) the FSM start in a fixed initial state, 2) the FSM be resettable to its initial state, and 3) each state of the FSM is reachable. Sometimes it is possible to test an FSM that does not satisfy these assumptions by transforming the FSM into an FSM that does. For example, if one or more states of the FSM are not reachable the FSM can usually be decomposed into two or more FSMs that individually satisfy the reachability assumption. The resulting FSMs can then be tested separately.

Since Eugene does not have access to design information for the FSM (i.e., Eugene is a black-box method), it operates by searching for input-output pairs. Based on responses of the FSM to inputs Eugene generates and sends, Eugene produces a set of input sequences that tend to uncover as many input-output pairs of the FSM as possible.

Since Eugene is a GA, it operates probabilistically. There is no assurance that, for a given FSM, Eugene will converge to a globally optimal sequence. Eugene can be programmed to halt after a predetermined number of generations. For the FSMs examined to date, Eugene generally gives good results within the space of 100 generations of 100 chromosomes. Results nearly as good are generally obtained much earlier in the run, usually in less than 20 generations.

If the FSM is not minimal, the FSM induced by Eugene will nevertheless be minimal. Moreover, if the module being tested is not implemented as an FSM, but its behavior is consistent with that of an FSM, Eugene will induce an FSM. Thus, Eugene's lack of provable correctness of result is complemented by its ability to undertake a wider variety of problems than formal techniques such as those of Chow (ibid.).

There are several possible ways in which Eugene can be employed. One way is to choose the individual of highest fitness in the final population. No other sequence in the entire run was found capable of detecting more input-output pairs of the FSM than this individual. The sequence, along with the actual output values produced by the FSM, can be used to induce a minimal FSM. In testing, this computed FSM can be compared to a design specification. In reverse engineering, the computed FSM becomes the design representation of the subject FSM.

Alternatively, an entire set of relatively "fit" chromosomes may be collected as a test data set for the FSM. Although the individual chromosome of highest fitness discloses no fewer input-output pairs than other chromosomes, the set of input-output pairs it discloses may be disjoint with respect to that disclosed by one or more chromosomes with the same, or even lesser, fitness. This set of chromosomes may be presented to the FSM as a test data set and the behavior of the FSM then compared with predictions based on the design representation.

A similar use of multiple chromosomes allows a reverse engineering operation to check for possible premature termination of the GA. If one or more of the high-fitness sequences induce an FSM not compatible with those induced by less fit sequences, the GA should be allowed to examine additional generations. This ability has not been programmed into Eugene, but would not be difficult. Note, however, that the probabilistic nature of Eugene's GA algorithm offers no guarantee that additional iterations will resolve the inconsistency.

History

A predecessor of Eugene, called Tinker, was implemented based on the SGPC of Walter Tackett and Aviram Carmi [1993]. Their program is essentially a C language implementation of the LISP code given in Koza's book (ibid.). Because of the efficiencies made possible through the use of C, their program is reported to run significantly faster than the original LISP code.

Tinker did not survive a conversion from SunOS to Solaris 2, which would have required changes to the Tackett and Aviram code we had ported to SunOS. We determined that this was a suitable opportunity to rebuild Tinker without dependence on code written by others. This afforded the opportunity to simplify the code considerably since many general-purpose facilities provided by the Tackett and Aviram code were not needed for our application.

A new program, christened Eugene, was coded in ANSI C using the Linux operating system (SLS and Slackware distributions) running on an IBM-compatible 486SX or 486DX2 PC. Despite the greatly reduced hardware costs associated with a PC as compared to a Sun workstation (SPARC station 2), we have been favorably impressed by the performance of the Linux implementation, which has afforded us greater flexibility in working with Eugene.

Structure and Operation of Eugene

Eugene uses a simple fitness function that encodes the dual goals of maximizing the number of input-output pairs discovered and minimizing the length of the input test sequence (parsimony). Typical to genetic algorithms is a fitness function that decreases in value as better fitness is achieved. We subtract from 100000 the product of 1000 and the number of input-output pairs found. We then subtract the number of genes in the chromosome. This simple function generally provides rapid convergence in early generations to a population that discloses many input-output pairs, as well as convergence in later generations to more compact input sequences that are equally effective in disclosing input-output pairs.

Eugene supports a number of genetic operators we thought suited to the problem of testing FSMs. An evaluation of their effectiveness is found in the following section. Operators include

asexual reproduction (copying), bisexual reproduction with chromosome cross over, mutation of the value of a single gene, insertion of a random gene and deletion of a single gene. Operations are selected randomly, based on input parameters, and applied to randomly selected individuals. Two distinct forms of deletion are provided in Eugene: one deletes a random gene within a chromosome; the other (the "snip" operator) deletes the last gene. Similarly, there are two forms of insertion: one inserts a gene at a random point; the other (the "extension" operator) appends a new gene on the end of a chromosome.

We have generally obtained satisfactory results running Eugene for 100 generations of 100 individuals. Thus, a run examines 10,000 individuals in total. We generally constrain the length of a chromosome (i.e., input sequence) to 200 genes. Although Koza (ibid.) recommends relatively larger populations (and a not quite proportionally smaller number of individuals), our computing resources were not adequate to examine the desired set of cases under such circumstances. We therefore employed two techniques described by Koza, to obtain faster convergence.

Holland [1975] recommends a fitness-proportionate selection strategy in which individual chromosomes are selected for reproduction based on their fitness scores. In Eugene, we have biased our selection method by boosting the probabilities associated with the relatively more fit members of the population. This technique is known as over selection. Our early experience with Eugene indicated that over selection improved convergence rates without sacrifice of final fitness scores. We did not formally test this hypothesis. The experiments reported herein all used over selection.

Another technique for improving convergence is elitism, described by Koza (ibid.). Elitism involves always retaining the cumulatively best individual, which might otherwise be discarded through operation of proportionate selection. Again our early experience indicated that elitism improved convergence rates without harm to final fitness scores, but we did not formally test this hypothesis. The experiments reported in the next section all used elitism.

Tinker's selection mechanism was considerably simpler, and more efficient in implementation and execution, than that employed in Eugene. Interestingly, Tinker's convergence rates were also better than those of Eugene until over selection and elitism were added to Eugene. Tinker divided the population into three categories based on fitness. A low-fitness category, the size of which was parametrically determined, was unconditionally removed from the population. A high fitness category, again parametrically determined, was copied unchanged into the new generation. A third category, along with the high fitness category, was allowed to participate in bisexual reproduction with cross over. Individuals serving as parents were randomly selected from within this group.

The discontinuous probabilities reflected in Tinker's selection mechanism, like those of over selection and elitism, can result in a GA homing in on a local maximum and failing to find a true global maximum. Further research is needed to determine the extent, if any, to which this tendency affects convergence in problems such as those addressed by Tinker and Eugene.

5. Results and Conclusions

Eugene was run over a three-day period using as input a FSM generated randomly. The FSM included 10 states, five input symbols and 50 transitions. The FSM was larger, therefore, than two of the three tested by Chow (ibid.) and was about the same as the third. A second series of runs was made using a different randomly generated FSM; the results were not obviously different from those obtained using the first FSM. The long run times and lack of a fast, dedicated computing facility precluded further analysis of Eugene's sensitivity to the input FSM.

Eugene was parameterized to generate sequences from an alphabet consisting of 25 symbols and was constrained to produce chromosomes of 200 genes or less. Thus, the space explored by Eugene consisted of 25^{200} possible sequences, or about 3.9×10^{279} sequences. Each generation consisted of 100 individuals and a total of 100 generations were examined.

Several different cases were run for a total of 100 times each. The best, worst and average fitness were reported for each generation and the best-of-run individual was printed every 10 generations.

The so-called Base Case specified the mutation operator with probability 1%, the deletion operator with probability 50%, extension operator with probability 1%, and bisexual reproduction with a probability of 90%. The high incidence of the deletion operator was intended to aid in locating parsimonious sequences. Note however, that this high level of occurrence of the deletion operator was found to be counter-productive. The various cases examined and the associated results are described in table 2.

Recall that low fitness values indicate the best result. Most of the cases correctly found the 20 input-output pairs of the input FSM, as shown by associated fitness values from 80,000 to 80,999. The number of items in the input sequences for these cases ranged from 180 to 198. [The high population case found a chromosome with only 179 genes, but examined over twice as many individuals as the other cases.]

Table 2. Case Summary

Case	Mutation	Snip	Delete	Extend	Insert	Breed	Fitness	Rank
Base	1%	0%	50%	1%	0%	90%	81,111	6
High Breed	1%	0%	50%	1%	0%	95%	81,125	8
High Extract	1%	0%	50%	1%	0%	90%	80,186	2
High Mutation	5%	0%	50%	1%	0%	90%	80,198	5
High Snip	1%	2%	50%	1%	0%	90%	80,193	4
Low Breed	1%	0%	50%	1%	0%	85%	80,191	3
Low Deletion	1%	0%	40%	1%	0%	90%	80,180	1
Low Extension	1%	0%	50%	0%	5%	90%	81,128	10
Low Mutation	1%	0%	50%	1%	0%	90%	81,125	8
High Population	1%	0%	50%	1%	0%	90%	80,179	N/A

Inspection of the best-of-run individuals disclosed many genes with values outside the range accepted by the FSM. Continuing the run beyond 100 generations would possibly have improved the parsimony of the chromosomes, but the rate of convergence to minimal chromosomes was not rapid overall. The first dozen or so iterations usually found nearly optimal chromosomes in terms of input-output pairs but the remaining iterations did not quickly decrease the number of genes in the best-of-run individual.

The rate and quality of convergence varies according to the parameters used. It seems possible to significantly improve the performance of this GA by proper choice of parameters; but how optimal parameters might be determined in advance is not obvious.

The lengthy run times mentioned above are the result of the number of cases run. Individual runs complete a single generation in less than 2 seconds on an Intel 486 PC-compatible.

In summary, Eugene appears to be a viable tool for inducing FSMs from software code. Its run times and accuracy seem to be within the necessary limits.

6. Suggestions for Further Investigation

As described above, Eugene's predecessor (Tinker) used a simplified selection mechanism that seemed to perform better than that used in Eugene. This hypothesis should be examined more rigorously in the context of a designed experiment.

Eugene employs over selection and elitism in an attempt to obtain faster convergence. The hypothesis that these options in fact do improve convergence should be examined.

Eugene needs to be extended to include automatic data collection and checkpoint/restart. Checkpoint/restart would permit effective use of computing time that would not otherwise be available, thereby allowing examination of more cases and a greater variety of cases.

Eugene's ability to deal with software systems not representable as FSMs should be explored. An important issue is how the discovered input-output pairs might be used to describe the behavior of the software using some model other than the FSM.

Uckun et al (ibid.) report the use of a hill climbing algorithm in conjunction with a GA. This synergistic combination exploits the ability of the GA to rapidly find local optima within the search space and the ability of the non-genetic, hill climbing algorithm to converge rapidly to the precise value of each local optimum. Such a strategy might lead to improved chromosome parsimony and more rapid convergence in testing of GAs.

Potential practical application of this technology is seen in the testing of software implemented using the FSM model and in the reverse engineering of such software. To test a software unit, the unit must be equipped with a test harness that permits inputs and outputs to be coupled to Eugene. Presently, Eugene is limited to input and output of 32-bit integers. If the software unit's inputs and outputs are of non-integer data type, the harness must also map the software unit's data type onto the set of integers. Note that this is not always possible. In addition, the harness must

provide a means of resetting the software unit to its initial state. In operation, Eugene will generate sequences of inputs and present them to the software unit, after resetting it to its initial state. Eugene will monitor the resulting outputs and thereby induce the FSM structure. When Eugene halts, the reported FSM structure can be compared with the functional specification or the design specification to determine whether the unit was correctly implemented. Note that this may involve determining the equivalency of two functionally identical FSM structures.

Eugene, by design, will tend to produce a minimal FSM. If the FSM specification contains redundant states or transitions Eugene is unlikely to reproduce them. This behavior makes Eugene useful as a reverse engineering tool. Given a redundant FSM, Eugene will tend to induce a simpler, clearer FSM. Since Eugene can operate on non-FSM software, it is possible to use Eugene to induce an FSM which approximates the behavior of a non-FSM software unit. The software unit can then be re-implemented as an FSM. Exceptions that distinguish the unit's behavior from that of the FSM induced by Eugene can be trapped and handled specially. In many cases, the resulting software structure may be improved by this means.

A test of Eugene's ability to correctly induce a FSM from behavior of an actual software system is desired. The authors would welcome such an opportunity and encourage interested readers of this paper to contact them.

Bibliography

Beizer, Boris. *Software Testing Techniques*, 2nd. edition. New York: Van Nostrand Reinhold, 1990.

Chow, Tsun S. "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, May 1978, pp. 178-187.

Hatley, Derek J., and Pirbhai, Imtiaz A. *Strategies for Real-Time System Specification.* New York: Dorset House, 1988.

Holland, John H. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

Koza, John R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, Mass.: The MIT Press, 1993.

Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs.* New York: Springer-Verlag, 1992.

Minsky, M.L. *Computation: Finite and Infinite Machines.* New York: Prentice Hall, 1967.

Rivest, Ronald L., and Schapire, Robert E. "Inference of Finite Automata Using Homing Sequences," *Information and Control*, No. 103, 1993, pp. 299-347.

Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; and Lorensen, William. *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey: Prentice Hall, 1991.

Shultz, Alan C.; Grefenstette, John J.; and De Jong, Kenneth A. "Test and Evaluation by Genetic Algorithms," *IEEE Expert,* October 1993, pp. 9-14.

Sidhu, D.P., and Leung, T.K. "Formal Methods for Protocol Testing: A Detailed Study," *IEEE Transactions on Software Engineering* SE-15, No. 4, April 1989, pp. 413-426.

Tackett, Walter Alden, and Carmi, Aviram. "SGPC: Simple Genetic Programming in C", version 1.1. Available via Internet: contact gpc@ipld01.hac.com.

Uckun, Serdar; Bagchi, Sugato; Kawamura, Kazuhiko; and Miyabe, Yutaka. "Managing Genetic Search in Job Shop Scheduling," *IEEE Expert,* October 1993, pp. 15-24.

Witt, Bernard I., Baker, F. Terry, and Merritt, Everett W. *Software Architecture and Design: Principles, Models, and Methods.* New York: Van Nostrand Reinhold, 1994.

Yourdon, Edward. *Modern Structured Analysis.* Englewood Cliffs, New Jersey: Yourdon Press, 1989.

Disclaimer

Prof. G. Thomas Plew contributed significantly to the ideas presented in this paper. However, owing to a recent illness, he was unable to review the paper prior to its submission in final form. Responsibility for errors and omissions in this paper is therefore that of Prof. McCarty and not of Prof. Plew.

State-Based Software Testing

Tomas Vagoun

Information Systems Department
College of Business and Management
University of Maryland
College Park, MD 20742
Email: tomv@src.umd.edu

Alan R. Hevner

Salomon Brothers/HRCP Chair of Information Systems
College of Business Administration
University of South Florida
Tampa, FL 33620-7800
Email: ahevner@cis01.cis.usf.edu

Abstract

The efficiency of software testing can be improved by partitioning the input domain. In this paper we examine how partitions of the input domain can be established within the Cleanroom software engineering methodology by utilizing the state box system design. The improvements are demonstrated with a simulation and a case study.

Keywords: software testing, Cleanroom software engineering, state-based testing.

Biographical: Tomas Vagoun is a doctoral candidate in the Information Systems Department, University of Maryland. With a background in computer science, his research is in the area of software quality and software testing techniques. His research is supported in part by IBM.

Biographical: Alan R. Hevner is a professor in the Information Systems and Decision Sciences Department at the University of South Florida. He has published, lectured, and consulted extensively in areas of system analysis and design, software quality, and database design. This research was performed while he was a member of the Information Systems Department at the University of Maryland.

1. Introduction

The proliferation of information systems into all aspects of our lives forces the software engineering community to re-examine the theories and practices used to control software quality. Chronically, a large percentage of software projects is completed with costs over budget, with low quality, and exceeding the estimated development time.

In the area of software quality assessment a number of techniques can be used. Software testing is the most prevalent. Testing techniques can themselves be roughly characterized as applying functional or structural strategies. The functional strategies regard software as a black box and the main emphasis is on testing the user-level functions. Typically, tests are selected according to known or expected software operational profiles. The main advantage of functional strategies lies in their closeness to user-oriented quality assessment needs. Statistical reliability measures, such as the Mean Time To Failure (MTTF) rate, can be calculated. The disadvantage of such strategies stems from providing little or no information about what occurs inside the software during testing.

On the other hand, in structural strategies tests are designed specifically based on software's internal characteristics. For example, during coverage testing, tests are created to exercise every execution path. As a result, structural strategies provide good internal assessment of the software's quality. However, this view of quality is removed from the needs of users who are interested in the overall software reliability.

In light of the dilemma between functional and structural testing, researchers as well as practitioners recognize the need to base testing strategies on a combination of both approaches. Our objective is to present an improved testing strategy based on the system's state design. Past research has established the use of system's state design as a viable approach in testing improvement, see for example [Hamlet 1992] and [Rapps and Weyuker 1985]. In this paper, we present the underlying theory of state-based software testing and identify the benefits of this testing strategy over the current practices. Our findings are supported by a case study that demonstrates these advantages.

2. Cleanroom Software Engineering

Our research in state-based testing is performed in the context of Cleanroom software engineering. However, the principles of state-based testing apply to any disciplined software engineering approach. To establish our research context, an overview of Cleanroom is presented in this section.

Cleanroom software engineering develops software systems under rigorous quality control. The goal is to help developers design and implement highly-reliable, near zero-defect software. The emphasis is on denying the entry of defects during the development, hence the name "Cleanroom." The Cleanroom development process is shown in Figure 2.1. Cleanroom development involves a specification team, a development team, a certification team, and a

documentation team. The specification team prepares and maintains the specification and then specializes it for each development increment. The development team designs and implements the software. Group inspections verify the correctness and completeness of the software design. The certification team compiles, tests, and certifies the software's reliability. The complete system is developed by being evolved through a series of tested and certified executable product increments. A comprehensive overview of Cleanroom is found in [Dyer 1992]. We shall briefly describe the Cleanroom process by discussing its central concepts.

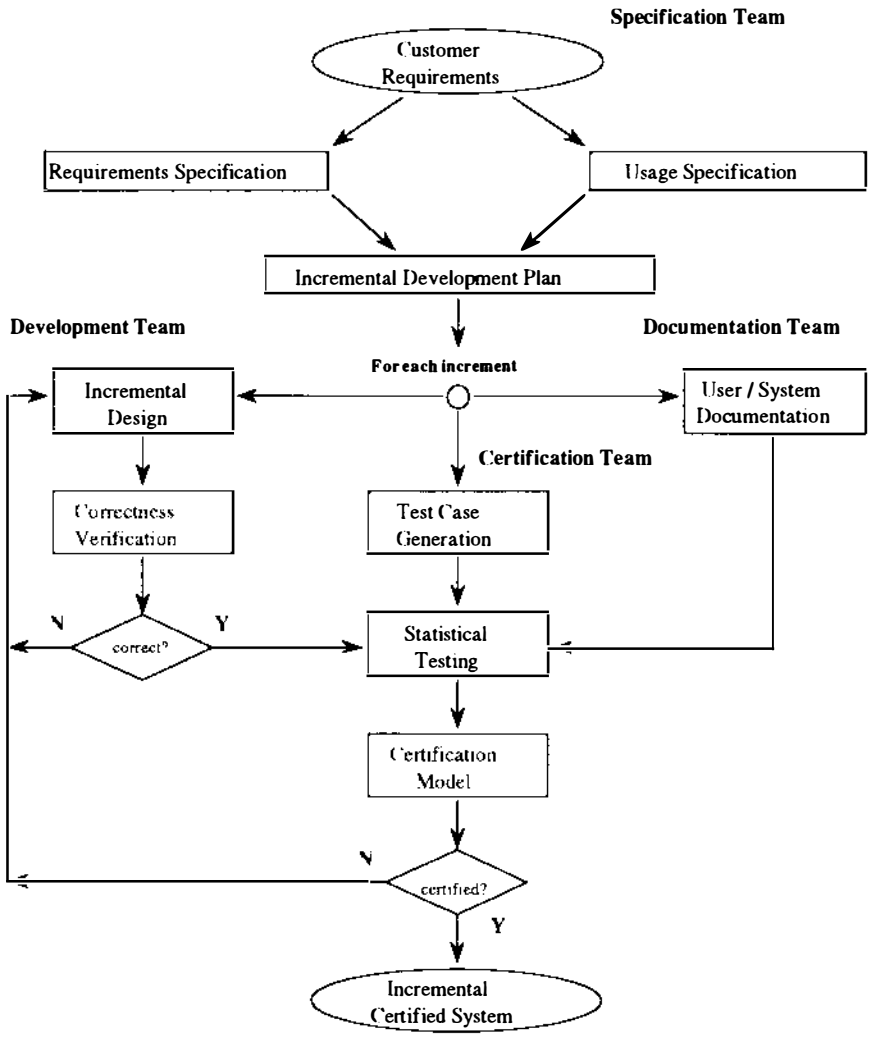


Figure 2.1 Cleanroom development process.

2.1 Incremental development

This concept is a focal point of Cleanroom development, supporting prototyping, configuration management, version control, incremental verification, and incremental testing. The development process is divided into manageable increments, each of which defines a complete end-to-end system with added functionality over previous increments. Each new increment is

integrated with the system's most current incremental version. Thus, potential design inconsistencies are resolved early in the development. Each successive increment builds onto the hierarchy until the system is eventually completed.

2.2 Box structure analysis and design

The analysis and design for each increment follow a structured set of methods based on box structures [Mills 1988]. This technique models system components as data abstractions in three increasingly detailed forms:

1. Black box - gives an external description of the system in terms of stimulus histories and responses. This is the most general and implementation-free view.
2. State box - gives additional information by showing what data (state) are needed for the system's function
3. Clear box - shows the interaction and the control flow among the next-level black boxes.

Figure 2.2 depicts the relationships between the black box, the state box, and the clear box. The expansion steps represent the process of creation and designing and the derivation steps represent the process of verification.

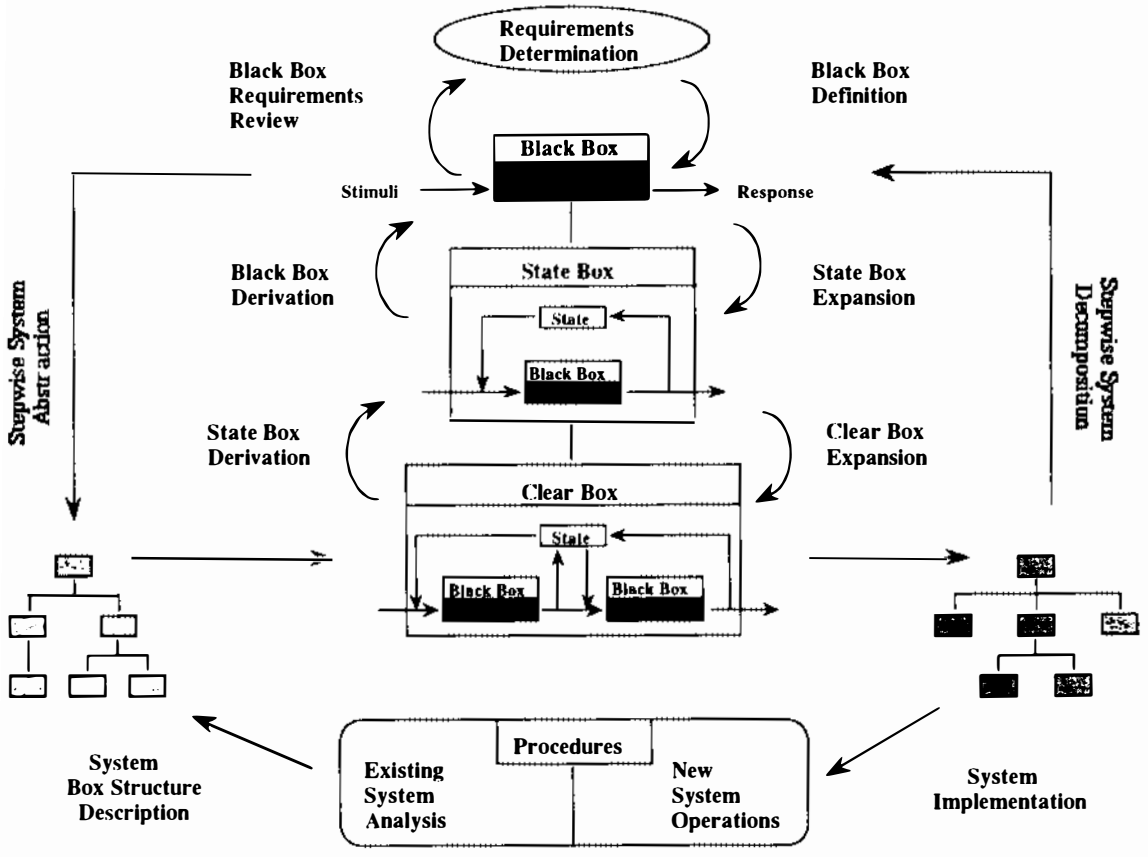


Figure 2.2 Box structure expansion and derivation.

The box structure hierarchy provides the representation needed for top-down Cleanroom incremental development. It combines the data- and procedure-oriented specification, capturing relevant design information. No bridges are needed to transform information between development phases.

2.3 Functional verification of correctness

Once the design is complete, the development team expands the clear box at each level into code that fully implements the defined functionality specified by the black box for this level. Two types of correctness verification are used. During the design activity, each expansion from black box to state box and from state box to clear box can be verified immediately for consistency by reversing the design process and deriving the state and the black box. If the derived boxes match the original boxes, then the design can be verified as consistent. Following each expansion, the development team uses functional verification to develop a proof that the design correctly implements the required specification. The proof strategy is divided into small parts that scales up to a proof for a large program.

2.4 Reliability certification

In parallel with the development team, the certification team uses the requirements specification and the usage specification to build a set of test cases. Once the increment is designed and implemented, it is integrated with previous increments, and statistical testing is then performed [Poore et al. 1993]. The reliability of the implemented system is analyzed via MTTF.

2.5 Cleanroom results

The Table 2.1 illustrates results from some of the completed Cleanroom projects [Linger 1994] (KLOC = thousand lines of code):

YEAR	PROJECT	RESULTS
1988	IBM COBOL Structuring Facility 85 KLOC, (PL/I)	IBM's first Cleanroom product Certification testing failure rate: 3.4 errors/KLOC 7 errors in first 3 years of use
1991	IBM AOEXPERT/MVS 107 KLOC, (mixed languages)	Certification testing failure rate: 2.6 errors/KLOC No operational errors from Beta test sites
1992	NASA Satellite Control Project 170 KLOC, (FORTRAN)	Testing failure rate: 4.2 errors/KLOC
1993	U of Tenn.: Cleanroom Tool 20 KLOC, (C)	Certification testing failure rate: 6.1 errors/KLOC
1993	Ericsson Telecom AB Switching Computer OS32 350 KLOC, (PLEX, C)	Testing failure rate: 1 error/KLOC 70% improvement in development productivity 100% improvement in testing productivity

Table 2.1 - Results of Cleanroom projects.

3. State-Based Testing

Software's state refers to the collective values of software's variables between successive system invocations by the user. Our state-based testing strategy combines elements from both functional and structural testing approaches. The focal point of state-based testing is the software's state box representation; that is what is meant by "state-based." The state box is the transitional representation of a design from the black box to the clear box. Its main purpose is to show the states of the system which are to be preserved between successive stimuli upon which the system is to act. The key notion is that only the states between successive, user-level stimuli are being considered, rather than all possible states software can be in during its operation.

As various functions of the software are executed they change the software's internal state by modifying values of state variables. The values of some state variables are typically preserved between the successive uses of the software and eventually the values of state variables are used as a part of the input during the future executions of the software's functions. Exposing the uses of state variables provides a mechanism for partially ascertaining the correctness of the implementation without the additional procedural details represented in the clear box. In this manner, the state box is a source of information about the structure and the internal behavior of the software without overburdening the testing process with the implementation details of the software.

A state box is defined as a generalization of the state machine [Mills 1988]. In the state box, a transaction is represented as a transition function from the set of existing states and the set of stimuli to the set of responses and new states: (*stimulus, old state*) $\rightarrow$ (*response, new state*). A more detailed treatment of box structures is given in [Mills et al. 1987] and [Mills et al. 1986].

3.1 Extended state box

One of the first outcomes of our research was to extend the currently limited formalism of the state box. The definition of our extended state box allows us to derive additional constructs. The extended state box is briefly presented here.

Definition: a state box, SB, is a 6-tuple, $SB = (F, \alpha, T, V, G, K)$, where:

- **F** is the set of black box functions which are the state box specifications;
- α is the encapsulation function that transforms black box functions into state box transactions;
- **T** is the set of state variables;
- **V** is the set of initial values of T;
- **G** is the set of state box transactions that implement specified black box functions; each transaction is of the form: (stimulus, current state) $\rightarrow$ (response, new state); and
- **K** is the set of all state box statements; each statement is either a concurrent assignment, next level black box invocation, or if-then-else construct.

Based on the extended state box we define the following two constructs:

State box slice: a SB slice is a subset of state box statements that are invoked by a specific (input, current state) combination; and

State box trace: SB traces extend SB slices through all levels of software design along the box usage design hierarchy.

Definition: a state trace, *ST*, is a 4-tuple, $ST = (U, E, Rd, Wr)$ where:

- **U** is the set of state box statements from state box slices which belong to state boxes on a state expansion path through the box usage design hierarchy;
- **E** is a union of Boolean conditions which determine when the state trace is invoked;
- **Rd** is the set of state variables which are Read by statements in the trace; and
- **Wr** is the set of state variables which are Written to by statements in the trace.

One of the purposes of specifying state traces is to create partitions with high density of failure-causing inputs. The other purpose is to provide additional information about the correctness of the software during testing in terms of the values of state variables before and after a test.

First, it is assumed that in software there are different defects with different probabilities of being uncovered; that is, defects have different sizes. A defect which is uncovered with many test inputs has a large size, whereas a defect which is uncovered with only few test inputs has a small size. The size also represents the defect's contribution to the overall software's failure rate; larger defects contribute in a larger proportion.

The second assumption made is that in software, there are defects that are uncovered only after a certain sequence of inputs is executed. This assumption corresponds to the notion that some software failures are due to an accumulation of undesirable state data.

The effect of input domain partitioning is to increase the probability of selecting those inputs or sequences of inputs that can potentially uncover a defect. By isolating those inputs that correspond to functions that use the same state variables from the rest of the input domain, the probability of selecting a failure-causing sequence increases.

4. State-based Partitioning

Partitioning has been used in a number of fields to improve the efficiency of statistical analyses. Typically, partitions are constructed in such a way that any element in a partition represents the characteristics and behavior of all the other elements as well. In testing, partitioning of the input domain can lead to certification with fewer number of tests because a small number of tests selected from a partition can represent an entire subset of inputs. In [Podgurski et al. 1993], for example, the use of this idea has been demonstrated. In this research, partitioning of the input domain is based on the state box and state traces. In this approach, we utilize information about the level of binding among state traces when the same set of state

variables is accessed. Such partitioning increases the density of failure-causing inputs and consequently software certification can be performed with a reduced number of tests.

First, we report how we used simulation to study the properties of the testing (and defect removal) process under varying combinations of the number of stimuli, sizes of test cases, and the number of defects. Since obtaining the exact probabilities for all but the most simple cases would be very complex and laborious, instead, a computer program was written to simulate the testing process. The input into the simulator is a matrix that specifies four characteristics of the simulated software:

1. The software's user-level functions (software's stimuli);
2. The response for each function (correct or incorrect);
3. The set of internal state variables; and
4. Internal dependencies among functions (through reading and/or writing to state variables).

A random number generator is used to simulate random selection of functions by the user. All functions are equally likely to be selected. A function will produce an incorrect response if either the response is incorrect or if any of the state variables that it reads contains an incorrect value. At the beginning of each test case all state variables are cleared (there are n tests in a test case). Each run constitutes the execution of as many test cases as needed to find and remove all defects. Runs are executed 10,000 times and the reported number of test cases is equal to the average number of test cases it takes to remove all defects in the 10,000 runs.

4.1 Single defect

First, the simulator is employed to examine the relationship between the number of test cases needed to find two state based defects, the size of test cases, and partitioned versus unpartitioned input domain (Figure 4.1).

The top line on the graph represents a system of 20 functions, s = 20, with two independent state-based defects. The test case size is varied from n = [3..25] and the corresponding average number of test cases needed to find both defects is plotted. The bottom line represents the same system being tested but the testing strategy splits testing into two partitions, each with 10 functions and one defect per partition. From the simulation, one sees that the total number of test cases needed to find both defects is smaller for the partitioned case than for the unpartitioned one. The difference between these two scenarios depends on the test-case size and the difference decreases as the test-case size increases.

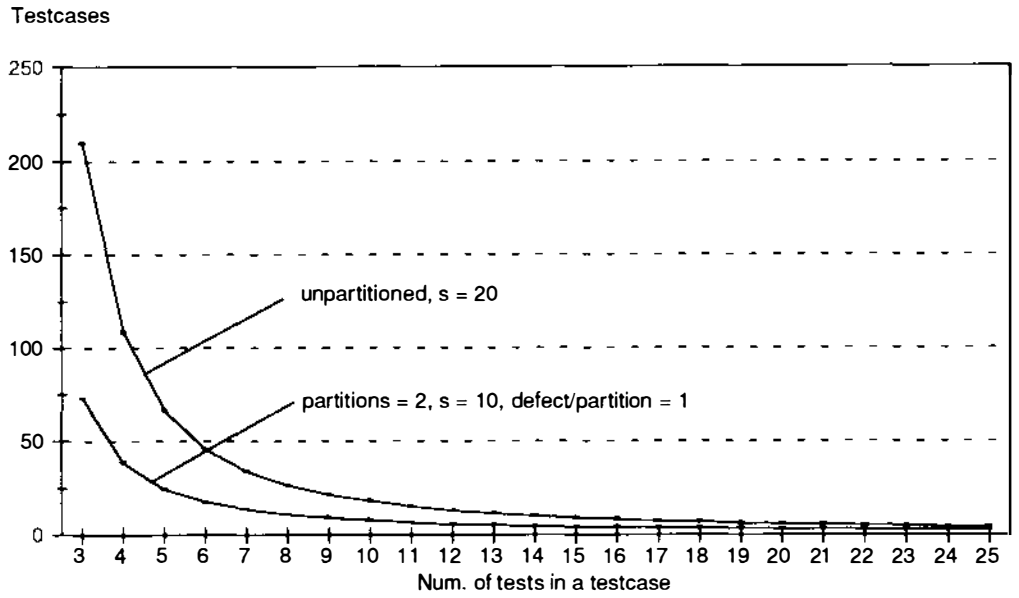


Figure 4.1 Partitioning vs. unpartitioning.

Several insights can be established. When partitioning of the input domain is done in such a way that it decreases the number of tests that are selected for a test case, the probability of selecting tests that lead to a failure increases. This reduces the total number of test cases needed. The difference in the number of test cases between the partitioned and unpartitioned scenario decreases as the test case size increases because the effects of partitioning diminish.

4.2 Multi-defect

The next simulation experiment seeks to examine the relationship between the average number of tests needed to find several defects while varying the number of partitions. The following software system description is used as the input into the simulation:

- There are 30 functions, s = {1,2,3,...,30};
- There are 10 defects: 5 response-defects, and 5 state-based defects;
- There are 5 state variables;
- Each group of functions: {1..6}, {7..12}, {13..18}, {19..24}, {25..30} contains one response-defect and one state-based defect.

In the simulation, each scenario (number of partitions) is repeated for several values of n (tests in a test case). The number of partitions is varied in the following way:

1. One partition, p1,
2. Two partitions, p2: partition A: s = {1,...,6}, and partition B: s = {7,...,30};
3. Three partitions, p3: A: {1,...,6}, B: {7,...,12}, C: {13,...,30};
4. Four partitions, p4: A: {1,...,6}, B: {7,...,12}, C: {13,...,18}, D: {19,...,30};
5. Five partitions, p5: A: {1,...,6}, B: {7,...,12}, C: {13,...,18}, D: {19,...,24}, E: {25,...,30}.

The graph in Figure 4.2 plots the average number of tests needed to find all defects in the system. In the one-partition case p1, one simulation was executed. For two partitions, the case p2, two simulations were executed (partition A and B) and the resulting average number of tests needed for each partition were added. The results of simulations for other cases were added in the same manner.

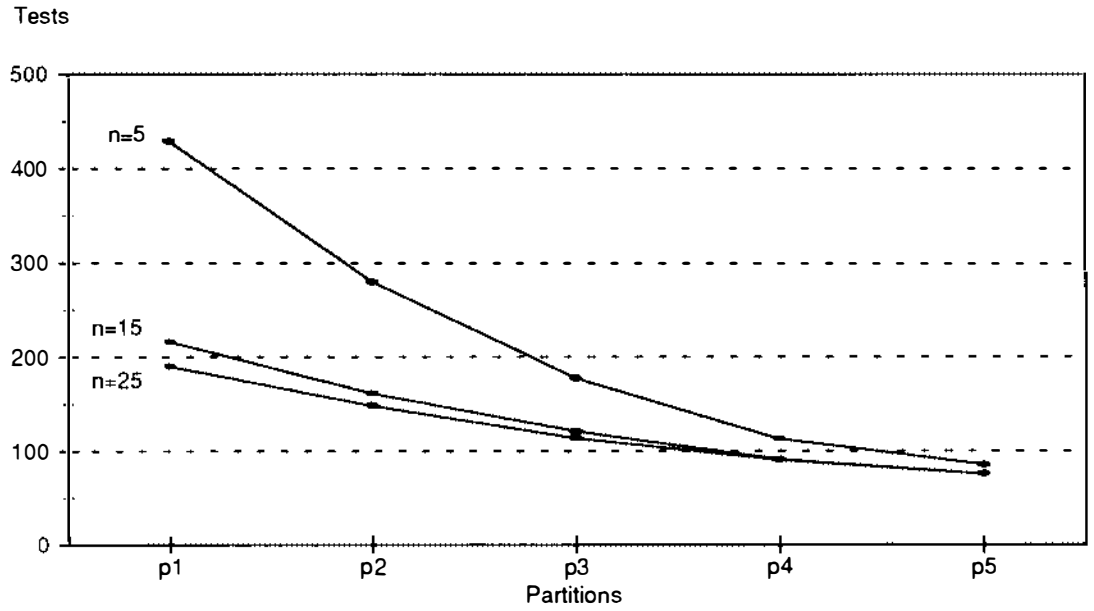


Figure 4.2 Multi-defect example: varying the number of partitions.

First, this graph repeats the conclusion from previous graph that increasing the test-case size (n) decreases the number of tests needed. Recall that after each test case completes, the state variables are re-initialized. In all cases partitioning helps to decrease the total number of tests needed. The improvement gets progressively smaller as the test-case size increases. This is due to the fact that the improvement from partitioning (increasing the probability of selecting a failure-causing input) becomes smaller.

5. Case Study

The simulation studies in the previous section illuminate some of the mathematical properties of partitioning in testing. As shown, partitioning of the input domain is beneficial whenever it increases the probability of selecting failure-causing inputs. For the failures caused by internal errors in the values of state variables, partitioning based on state traces makes it possible to increase the chance of selecting corresponding failure-causing inputs.

As important as the general properties of partitioning are, it is equally important to ascertain that a state-based testing strategy is practical. For this reason we have performed a

number of case studies to validate our approach and gain practical experience from applying the concepts. Here we report our findings from one such case study.

5.1 System's description

For this study we selected a text retrieval software package named lq-text written by Liam Quin (copyright 1993). This software is publicly available via Internet. A user can register text files with lq-text which will index the contents and save this information in its database. Then, the user can ask questions about the files and find out, for example, which of the files contain a given word, or which files contain a given phrase, and can display the text surrounding the given word. The software is written in C for the UNIX environment having some 10,000 lines of source code. The functions of the software are implemented as standalone executable programs called from a UNIX shell program which acts as the user interface. A user can access the executable programs through the shell program but also may execute the functions directly from the command line. The functions are invoked from the shell program and the behavior of each function can be further controlled by using a number of flags. Figure 5.1 represents the top level black box and the state box of the system's design:

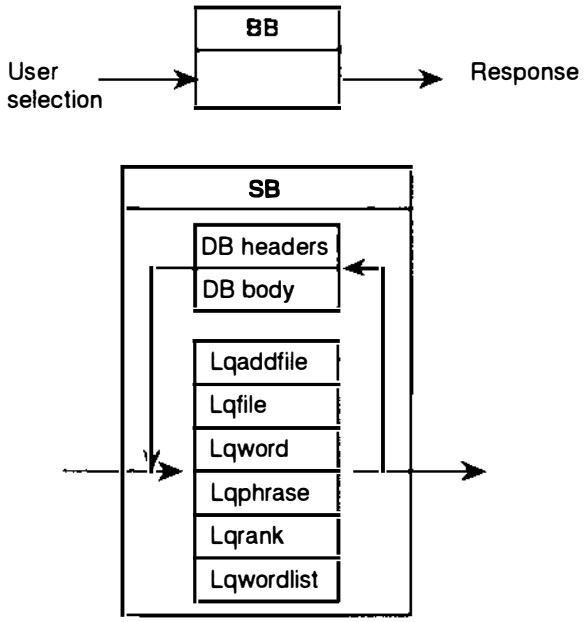


Figure 5.1 Lq-text system structure.

The functions of the software are:

- Lqaddfile - to add files to the database;
- Lqfile - to display the names of currently indexed files;
- Lqword - to get information about words in the database;
- Lqphrase - to look up phrases;

- Lqrank - to combine phrase searches and sort the results;
- Lqwordlist - to find words matching a pattern.

The state box analysis reveals that at the highest level, there are two, albeit complex, state variables. At the highest level of design, the internal database is represented in two parts: (1) the database headers which contain information about the indexed files, and (2) the body of the database which contains the indexes and other information pertaining to the content of the files. Each function can be invoked with a number of flags (options). Each option represents a state trace through the state box specification Altogether, 14 state traces were recognized as well as their accesses to state variables. This information is represented by the following state trace access table.

State Trace	State Variable DB Headers	State Variable DB Body	Function
Lqaddfile *filename*	W	W	Add and index file *filename*
Lqfile -a	R		List file names
Lqword		R	List all words
Lqword *word*		R	Get inf. about a word
Lqword -l *word*		R	No headers in output
Lqphrase *word*		R	Find a phrase
Lqphrase -l *word*		R	No headers in output
Lqphrase -m p *word*		R	Case sensitive search
Lqrank *word word*		R	Combine multiple words
Lqrank -u *word word*		R	Unsorted output
Lqrank -m p *word word*		R	Case sensitive search
Lqwordlist -p *str*		R	Find words with a prefix
Lqwordlist -e *str*		R	Find words with a suffix
Lqwordlist -g *str*		R	Find strings

This matrix was subsequently used as the basis for devising testing strategies, namely unpartitioned and partitioned ones. In more complex systems, the state-trace matrix can be also used as an input into clustering algorithms that will find natural groupings among state traces based on the number of shared state variables.

5.2 Testing

The software in this case study is seeded with defects and subjected to different testing strategies. The goal is to gain experience with state box software analysis and to verify in practice the performance of partitioning. There are two key characteristics of our testing environment:

- Automatic oracle

The oracle problem, i.e., deciding whether the output from the software is correct, is a complex and difficult subject. For our testing needs, we have dealt with this issue by writing a generic invocation script which executes the original version of a program as well as the seeded version

for each input. The outputs from both versions are automatically collected and compared. The output from the seeded version is declared incorrect if it does not match the output from the unseeded version. Any unknown original defects are present in both versions of the software, and they are omitted from consideration for the purpose of this study.

- Test case generator

In order to eliminate as much of the manual crafting of test cases as possible, we have employed a test case generator. Specifically, we have utilized the Cleanroom Certification Assistant developed at the IBM's Cleanroom Software Technology Center. This software provides a grammar-based test case generator and analysis routines to calculate software's MTTF rate throughout the testing process.

The software under consideration was subjected to three testing scenarios:

1. Unpartitioned system;
2. Partitioned system based on state variable access and operational profile;
3. Partitioned system strictly based on state variable access.

For each scenario the software was seeded with the total of four defects. Three defects represent simple function errors: (1) Lqword -l, (2) Lqphrase -l, (3) Lqwordlist -p, and one defect represents a state-based malfunction: (4) Lqaddfile. In the first three cases, the defects represent common programming errors, namely, misinterpreting a value of a flag, using a previously used variable, and omitting details from the output. In the fourth case, the function Lqaddfile saves incorrectly the given file name into the DB header. However, as there is no visible output, this error goes unreported until the user specifically lists the current file names. After each test case completes the software starts from a re-initialized (correct) state. Whenever a defect is found, it is immediately corrected and testing proceeds with a new test case.

1. Unpartitioned system

In this scenario, all state traces participate in testing and all are members of the same group. The size of each test case was set to 10 tests. Each state trace has an equal chance of being selected into a test case. Eight runs were completed, where a run corresponds to as many tests/test cases as are needed to find all four defects. On the average, it took 165 tests to eliminate all four defects.

2. State-based partitioning with operational profile

An advantage of statistically-based software testing used in this study (and used in Cleanroom in general) is the ability to provide user-oriented software quality metrics such as the MTTF rate. Although in this study we are not addressing the issue of reliability, still, the design of tests and test cases should reflect the software's expected operational profile. An operational profile specifies how the user will use the software, including the expected usage probability of each function. For a more thorough discussion on this topic see [Musa 1993]. In this second experiment, the system is divided based on how functions access state variables, but the divisions

are modified to reflect a more realistic use of the software. According to the state trace access matrix, the two partitions were established in the following way:

- P1: state traces (functions) accessing the state variable DB headers and other state traces that have a similar functionality;
- P2: the rest of state traces accessing the state variable DB body.

P1	P2
Lqaddfile	Lqaddfile
Lqfile -a	Lqword -l *word*
Lqword	Lqphrase *word*
Lqword *word*	Lqphrase -l *word*
Lqwordlist -g *str*	Lqphrase -m p *word*
	Lqrank *word word*
	Lqrank -u *word word*
	Lqrank -m p *word word*
	Lqwordlist -p *str*
	Lqwordlist -e *str*

In the unpartitioned scenario, the test case size was equal to 10. In other words, the expected usage of the system is 10 stimuli out of possible 14. To preserve this ratio, the test case size of partition P1 is set at n = 4 and for partition P2 it is set at n = 7. Total of six runs were executed with functions in partition P1 and the average of 28 tests were needed to find the defect. For partition P2, six runs were also executed and the average of 13 tests were needed to find the defects. In both partitions, the state traces had the same probability of being selected.

3. Strict state-based partitioning

Statistical software certification needs to be based on software's expected operational profile. Otherwise, the reliability measures are not relevant. However, to further show the effects of state-based partitioning, it is helpful in this case study to test the software when it is partitioned strictly based on the state. It should be noted that partitioning based strictly on state-access does not have to result in testing sequences that do not follow the operational profile. But, a congruence with the operational profile cannot be guaranteed.

According to the state trace access matrix, the system is divided into two partitions:

- P1: state traces accessing state variable DB headers;
- P2: state traces accessing state variable DB body.

P1	P2
Lqaddfile	Lqaddfile
Lqfile -a	Lqword
	Lqword *word*
	Lqword -l *word*
	Lqphrase *word*
	Lqphrase -l *word*
	Lqphrase -m p *word*
	Lqrank *word word*
	Lqrank -u *word word*
	Lqrank -m p *word word*
	Lqwordlist -p *str*
	Lqwordlist -e *str*
	Lqwordlist -g *str*

The partition P1 contains two functions, Lqaddfile and Lqfile -a, and thus it contains the state-based defect (Lqaddfile) → (Lqfile -a). The test case size was set at n = 2. The partition P1 was tested separately from the partition P2. Total of 21 runs were executed and the average number of tests needed to find the defect was 4.

The functions in partition P2 were also executed separately from P1. The test case size was left at n = 10. A total of 6 runs were executed and the average number of tests needed to find the three defects was 27.

Summary

The following table summarizes the results of this case study:

Scenario	n	s	Tests P1	Tests P2	Average total tests needed
1. unpartitioned	10	14			165
2. state-based and operational profile	$n_{P1} = 4$ $n_{P2} = 7$	$s_{P1} = 5$ $s_{P2} = 10$	28	13	41
3. state-based partitions	$n_{P1} = 2$ $n_{P2} = 10$	$s_{P1} = 2$ $s_{P2} = 13$	4	27	31

The three scenarios presented here demonstrate the effects of partitioning along a certain continuum; from no partitioning, to partitioning with operational profile, to a strict state-based partitioning. Although this case study does not produce conclusive evidence, our research suggests that efficiency of testing (the number of tests needed) is improved the most when strict partitioning is employed.

There is another advantage to partitioning based on state traces that was not addressed in this paper, but we feel it can provide useful benefits. There is a direct correspondence between test inputs and state traces. Whenever a test is executed, it is known which state trace was

invoked. If a test fails, the corresponding state trace can help pinpoint the location of the defect. This represents an improvement from the current certification approach in Cleanroom where little is known about the system's structure during testing.

6. Conclusions

Current research provides convincing evidence that partitioning the input domain during software testing decreases testing effort and provides improved defect isolation. The practical questions in applying this research are: (1) What criteria should be used for partitioning? and (2) How can we decide when partitioning will be effective in a testing environment? In this section, we will provide some guidance in answering these questions.

Partitioning of the input domain begins with a detailed understanding of user requirements to include operational profiles of how the user operates on the system [Musa 1993], [Whittaker and Poore 1993]. The discovery of natural divisions of user operations provides initial guidance on partitioning the input domain for testing. Our research contribution takes partitioning one step further to consider the use of system state. We use the Cleanroom methods of software development as the context for our research because of the formal identification of the state box in system design. (In non-Cleanroom system developments, one can use the data designs in a similar way.) By identifying the state variables in the software, we define state traces that provide clear criteria for partitioning the input domain. The experiments and case study in this paper demonstrate the improvements gained by basing partitioning on state information.

The second question is more difficult. Can we estimate the costs and benefits of performing input partitioning in software testing? As part of our on-going research, we are developing a cost model for state-based partitioning. We plan to validate the model in different testing environments; e.g., new system development, reengineering of systems, application systems, systems software, etc. Not all software testing will benefit from extensive state analysis and state-based partitioning. One reason may be because there are no clear partitions of state use. However, based on empirical evidence, such as shown in this paper, most testing should benefit significantly from state-based partitioning. Our future research direction is to define a comprehensive software testing process incorporating decision models for applying state-based partitioning.

Bibliography

[Dyer 1992] M. Dyer, *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons, 1992.

[Hamlet 1992] Dick Hamlet, "Are We Testing for True Reliability?," *IEEE Software*, July 1992, pp. 21-27.

[Linger 1994] Richard Linger, "Cleanroom Process Model," *IEEE Software*, v. 11, n. 2, March 1994, pp. 50-58.

[Mills et al. 1986] Harlan Mills, Richard Linger, Alan Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, 1986.

[Mills et al 1987] Harlan Mills, Richard Linger, Alan Hevner, "Box Structured Information Systems," *IBM Systems Journal*, v. 26, n. 4, 1987, pp. 365-413.

[Mills 1988] Harlan Mills, "Stepwise Refinement and Verification in Box-Structured Systems," *IEEE Computer*, v. 21, n. 6, June 1988, pp. 23-36.

[Musa 1993] John Musa, "Operational Profiles in Software-Reliability Engineering," *IEEE Software*, March 1993, pp. 14-32.

[Podgurski et al. 1993] Andy Podgurski, Charles Yang, Wassim Masri, "Partition Testing, Stratified Sampling, and Cluster Analysis," *ACM Software Engineering Notes*, v. 18, n. 5, Dec. 1993, pp. 169-181.

[Poore et al. 1993] J. Poore, Harlan Mills, David Mutchler, "Planning and Certifying Software System Reliability," *IEEE Software*, Jan. 1993, pp. 88-99.

[Rapps and Weyuker 1985] Sandra Rapps and Elaine Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Eng.*, v. SE-11, n. 4, April 1985, pp. 367-375.

[Whittaker and Poore 1993] James Whittaker and J. H. Poore, "Markov Analysis of Software Specifications," *ACM Transactions on Software Eng. and Methodology*, v. 2, n. 1, Jan. 1993, pp. 93-106.

Index

This is an index for all the articles in the Proceedings. Names of organizations and companies are in CAPITALS*; names of computer programs and projects are in italics. Other entries are in* roman *type.*

Q

R

S

1994 PROCEEDINGS ORDER FORM

PACIFIC NORTHWEST QUALITY SOFTWARE CONFERENCE

To order a copy of the 1994 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 _____