FOURTEENTH ANNUAL

PACIFIC NORTHWEST

SOFTWARE QUALITY CONFERENCE

October 28 - 30, 1996

Oregon Convention Center
Portland, Oregon

TABLE OF CONTENTS

Preface

G.W. Hicks

Welcome to the 14th Annual Pacific Northwest Software Quality Conference. Throughout its entire existence, PNSQC has been successful because of its attendees and volunteers and the companies they represent.

This year's conference is highlighted by two outstanding Keynote presentations. On Tuesday, Susan Dart from Dart Technology Strategies, Inc., presents an address entitled "But Why Can't I Have a Silver Bullet?" As many of us know, this is a question that seems to be asked over and over again.

On Wednesday, Watts Humphrey from the Software Engineering Institute (SEI) at Carnegie Mellon University will present "What If Your Life Depended On Software?" This is another interesting topic for all of us as software infiltrates all of our lives.

In addition to this year's fine Keynote presentations, we have invited three other well-known speakers—Judy Bamberger, Bill Hetzel, and Capers Jones—who will participate in a panel discussion everyone will want to attend.

Organizing and presenting the workshops and conference is always a challenge. We strive to keep the costs of our events affordable, which means we rely heavily on volunteers—professionals like you who want to share some of their time, skill, and experience with other software practitioners in the Pacific Northwest. There are many different levels of volunteering at PNSQC—from making phone calls, to interviewing candidates for the Software Excellence Award, to organizing events. We have a place for you, regardless of whether you can offer a little time or a lot. In 1997, several of our volunteer organizers will be "retiring," and we need your help to fill their places. Volunteering for PNSQC is a rewarding way to keep current on new trends and to network with other professionals. I ask you to consider volunteering because this is your conference, and we can't do it without you.

In closing, I would like to thank all of PNSQC's Officers, Directors, and Volunteers, along with Terri Moore of Pacific Agenda, who all worked to make PNSQC a reality for 1996.

CONFERENCE OFFICERS/COMMITTEE CHAIRS

G.W. Hicks - President/Chair
IMS, Inc.

Ian Savage - Secretary
CFI ProServices

Ray Lischner - Treasurer
Tempest Software

Dave Dickman, Program Co-Chair
Hewlett-Packard Company

Christine Olsen, Program Co-Chair
Hewlett-Packard Company

Judy Bamberger - Keynote
Process Solutions

Rick Clements - Publicity
Tektronix, Inc.

Karen King - Birds of a Feather
Sequent Computer Systems, Inc.

James Mater - Exhibits
Revision Labs, Inc.

Howard Mercier
- Software Excellence Award
Intersolv

Miguel Ulloa - Workshops
Mentor Graphics

CONFERENCE PLANNING COMMITTEE

Hilly Alexander
ADP Dealer Services

Judy Bamberger
Process Solutions

Rick Clements
FLIR Systems, Inc.

Dave Dickmann
Hewlett-Packard Company

Lauri Duff
ADP

Lynne Foster
Motorola

Cynthia Gens
Technical Solutions

Craig Hondo
IMS, Inc.

Bill Junk
University of Idaho

Karen King
Sequent Computer Systems, Inc.

Randy King
Informix

Ray Lischner
Tempest Software

James Mater
Revision Labs, Inc.

Howard Mercier
Step Technology

Fred Mowle
Purdue University

Christine Olsen
Hewlett-Packard Company

Linda Pellecchia
Tektronix

Ian Savage
CFI ProServices, Inc.

Eric Schnellman
CDP

Jim Teisher
Credence Systems Corporation

Eileen Trinnich
Hewlett-Packard Company

Miguel Ulloa
Mentor Graphics

Doug Vorwaller
Wacker Siltronics

Scott A. Whitmire
Advanced Systems Research

Barbara Zimmer
Hewlett-Packard Company

Presenters

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

Sandhiprakash Bhide
SQA Manager, Transmission Test
Tektronix, Inc.
PO Box 500, MS 50-310
Beaverton, OR 97077-0001

Kingsum Chow
Dept of Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350

Chi-Ming Chung
Dept. of Computer Science
and Information Engr
Tamkang University
Tamsui, Taipei 251 R.O.C.

Frank Cioch
Associate Professor
Department of CSE, School of Engineering
Oakland University
Rochester, MI 48309-4401

Rick Clements
Tektronix, Inc.
PO Box 1000
Wilsonville, OR 97070

Curtis Cook
Computer Science Department
Oregon State University
Corvallis, OR 97331-3202

Michael Deck
Cleanroom Software Engineering, Inc.
7526 Spring Drive
Boulder, CO 80303-5101

Leslie Dent
Synopsys, Inc.
700 East Middlefield Road
Mountain View, CA 94043

Tim Farley
3616 Palatine Avenue North
Seattle, WA 98103

Bernard Gracy, Jr.
DIAD Product Manager
United Parcel Service
2311 York Road
Timonium, MD 21093

Peter Hantos
Principal Scientist,
Corp S/W Engr Ctr
Corp Research & Technology
Xerox Corp
701 S Aviation Blvd, M/S ESAE 375
El Segundo, CA 90245

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

James Thomas Heuring
2410 137th Place SE
Mill Creek, WA 989012

Karen King
Sequent
15450 SW Koll Parkway
MS DES1-146
Beaverton, OR 98006-6063

Charles Knutson
962 NW Cypress Avenue
Corvallis, OR 97330

Glen Ledeboer
Computer Sciences Corporation
3170 Fairview Park Drive
M/C N300
Falls Church, VA 22042

Leslie Little
Aztek Engineering
2477 55th Street
Suite 202
Boulder, CO 80301

Steven March
Motorola Computer Group
Motorola, Inc.
1101 E. UIniversity Avenue
Urbana, IL 61801

Troy Pearse
Firmware Engr., Business LaserJet Div.
Hewlett-Packard Company
PO Box 15
Boise, ID 83707-0015

Sridhar Ranganathan
Manager, Beta Programs
Pure Software
1309 S. Mary Avenue
Sunnyvale, CA 94087

Jarrett Rosenberg
Sun Microsystems, Inc.
2550 Garcia Avenue
MPK 17-307
Mountain View, CA 94043-1100

Allen Sampson
S/W Qual Mgr, Msrmnt Business Div
Tektronix, Inc.
PO Box 500, M/S 39-732
Beaverton, OR 97077

Shel Siegel
Objective Quality, Inc.
1329 Fir Street
Sandpoint, ID 83864

Jeffrey Voas
Director of Research
Reliable Software Technologies Corp.
21515 Ridgetop Circle, Suite 250
Sterling, VA 20166

Ashley Wee
PO Box 6423
Bellevue, WA 98008-0423

William Wilson
GSFC, Code 300.1. Bld 32
Greenbelt, MD 20771

But Why Can't I Have A Silver Bullet?

Susan Dart, Dart Technology Strategies, Inc.

Abstract

Companies really want to improve the quality of their software development and maintenance practices so that productivity is enhanced and more bug-free software is produced. The developers and managers would like to buy software tools to enable such improvement. In fact, we'd all like tools that were "silver bullets" for our companies; that is, we'd buy the tools and install them, then, hey presto! within a few days the company would be magically transformed into a much improved place. But, unfortunately, it doesn't work that way.

This talk presents why tools cannot be silver bullets and why many companies fail in achieving any quality improvements by tool adoption. It presents lessons learned and highlights an adoption method that works for large companies doing enterprise-wide adoption of tools.

About the speaker

Susan Dart is President and CEO of Dart Technology Strategies, Inc., a consulting firm that helps companies achieve process improvement via the adoption of technology, and with a specialization in configuration management (CM). Ms. Dart has 20 years of experience in industry and academia, focusing on software tools and software development environments. She has contributed over 50 international publications and seminars, including being co-author of the next Ovum book on Configurations Management Tools. Ms. Dart participate on the United States' Federal Aviation Authority panel on developing CM for Explosive Detection Systems.

Before starting her own company, Ms. Dart was Vice President of Process Technology at Continuous Software Corporation, a CM vendor, where she developed adoption services to assist customers in achieving the best possible CM solution. Previous to that, she spent 7 years at the Software Engineering Institute (SEI) of Carnegie Mellon University (CMU), developing methods for CM and environments, earning her an international reputation in these fields. Prior to the SEI, Ms. Dart developed compilers at Tartan, Inc. and telecommunications software and standards for Telstra, Australia.

Ms. Dart has an M.S. in Software Engineering from CMU and a B.S. with Distinction from RMIT.

BUT WHY CAN'T I HAVE A SILVER BULLET?

Susan Dart
President, Dart Technology Strategies, Inc.
1280 Bison, Suite B9-510,
Newport Beach, CA. 92660. USA
Phone: (714) 224-9929
Email: sdart@earthlink.net

This is an outline of a keynote presentation for PNSQC '96, Oct.29

Executive Summary

Life isn't as easy as we want it. It requires us to be experts in many fields. People can be so difficult sometimes. The best laid plans of mice and men often go astray. If you think things are bad at your company, there's always worse ...
Or, in other words:

- Why is it so complicated to buy and deploy a good software tool?
- Why do we believe marketing literature?
- Why are enterprise-wide tools so expensive?
- Why can't vendors solve all our tool problems?
- Why don't tools meet our expectations?
- Why can't we meet our software deadlines?
- Why is our software so buggy?
- Why is it so difficult to do process improvement?

Overview

- A heavenly situation: the silver bullet
- The reality
- Why reality precludes a silver bullet
- Example: configuration management
- How can we approach a heavenly state?

A Heavenly Situation: The Silver Bullet

- Silver bullet: buy it, install it, use it
- No major effort involved
- No deep thinking required
- No adverse effects on work schedule
- No installation or customization problems
- No learning difficulties
- All our needs are met
- Magic

Personal Reality Isn't Heaven

- What to buy? (So many choices!)
- Is it available? (Was advertised, but not in stock yet)
- What alternate do I buy? (But it's a lesser quality product)
- Is it worth it? (I'd really prefer my initial choice)
- OK, I have deadlines so I'll take it (I hope it works)

- Takes a week to install and get all software working (Gee, I thought it would take a few hours)
- After two weeks, it breaks down (Great, just in time for a deadline)
- It gets returned (Am I angry? Does the sun shine in California?)
- Get another one and reinstall this one (Whew, back in business)
- Integrate it with other tools (Why do I have to be a systems administrator to do this?)
- Two months later, it just stops working (What on earth is going on?)
- Spend days trying to get through to technical support (Does anyone else exist on this planet?)
- It's a manufacturer's bug: they had bad quality control (Why don't they tell us buyers about this rather than us having to suffer through it and waste time and money?)
- Call in supplier to fix it (Is the warranty still valid?)

Corporate Reality Isn't Heaven

- Someone chooses a tool for everyone else to use (Well, Computerworld said it was the best)
- Management chooses you to deploy it (Why me?)
- Developers won't use it; they don't like it (But they haven't even tried it yet!)
- They prefer their own in-house tool with their personal customizations (Better get my resume out)
- Management demands developers use it (Dictator!)
- No money in the budget for training (Hey, our developers are smart. Real developers don't need training)
- Only a few end up using the tool (But you know it's a great tool)
- You growl at the vendor (Where were they when we needed them?)
- Vendor says the solution is to buy more products and services (Oi!)
- Management says there's no money; besides, they need you on another project (Progress?)
- The tool sits on the shelf; the company continues to struggle with low productivity, high error rate, and low product quality.

So, What Are The Problems?

- Vendors
- Tools
- Customers
- Consultants
- Government
- Academia
- "Standards"

Vendors

- Focused on selling a "black box"
- Prefer to avoid deployment problems if possible
- Stay competitive: "good enough" attitude
- Marketing hype
- Sales pitch sets incorrect expectations
- Poor responsiveness for customer support
- Lack of services for after-sales support
- Startup company limitations
- Lack of vision in management
- "Good enough" quality attitude

Tools

- different classes and functionality
- difficult installation and set-up

- non-intuitive interface
- methodology or process not obvious
- sales pitch set incorrect expectations
- doesn't scale up or perform
- too many patches or upgrades
- too generic, requires customization
- doesn't integrate with all tools
- a solution in search of a problem

Customers

- Expect vendor to solve all problems
- Lack a good set of requirements
- Rely too heavily on trade publications and Internet bulletin boards for knowledge
- Expect a "silver bullet"
- Unable to pay for extra services such as training
- Not keeping track with technology evolution
- Developer and management dichotomy
- Lack of leadership and good management
- Lack skill set or experience

Consultants

- Too much process, not enough technology
- Limited experience
- Lack of accountability
- Restricted time frame

Government

- Everything takes too long to achieve
- Politics
- Not directly helping industry
- Premature standards regulation

Academia

- Disconnect with industry and state of technology and practice
- Focus limited to small scale problems
- Transition of ideas to industry takes too long
- Education too far removed from industry needs
- Conferences typically showcases for academic research instead of real-world problem solving
-

"Standards"

- More focus on what technology can do
- Guidance on how to achieve its requirements/goals
- Slow evolution of standards
- Too many standards
- Purpose of standards: regulation, improvement, certification, validation

Process Improvement

- Different approaches to process improvement

- Top-down, formal SEPG
- Bottom-up, champion
- Just-in-time need
- Best practices institutionalization
- Tool acquisition
- How to do process improvement?
- What is the return on investment?
- Which assessment instrument/model should be used?
- Should we become ISO 9000 certified?

Technology Adoption

- What decisions need to be made, when and how do we make them?
- How to do it?
- Corporate-wide or project-specific?
- How to deal with resistance?
- Which tool?
- How to deal with legacy problems?
- Ineffective pilot project

Getting Closer To A Silver Bullet

- An Example: Configuration Management
- Evolved from a corporate expense to a corporate core competency
- Classes of tools
- Defensive procurement
- Risk-mitigating deployment

Classes of CM Tools

1. Version control
2. Developer-oriented
3. Process-oriented

Defensive Procurement

- Scope of CM solution
- Budgeting for all elements of CM solution
- Evaluating most appropriate tools
- Using the vendor
- Preparing for deployment

Risk-Mitigating Deployment

- Deployment Team
- Risk-based adoption methodology
- Designing and automating CM processes

What Can Be Done To Get Closer To The Silver Bullet?

- Vendors have tremendous experience: procurement, deployment, competition
- Better understanding of classes of technology
- Tools designed for easier adoption
- Know how to do technology adoption
- Academia and SEI collaboration with vendors
- SPIN meetings should let vendors speak
- "Standards" become technology aware
- Better management and leadership

What If Your Life Depended on Software?

Watts Humphrey, Software Engineering Institute, Carnegie Mellon University

Abstract

If your survival depended on software, you would be pretty interested in how it was developed. Few people realize that software is now or soon will be critical to the performance, safety, or security of the most advanced electronic products. In this talk, Mr. Humphrey discusses the critical nature of software and the challenges software developers face. He shows that the methods traditionally used to develop quality software are ineffective and that better processes are essential.

Two approaches to these problems have shown substantial benefits. The Capability Maturity Model (CMM) is now widely used to help organizations improve their capability. Experience with the CMM shows sharp improvements in product quality and development productivity. The Personal Software Process (PSI) introduces software engineers to disciplined personal methods. After PSP training, engineers produce more accurate plans and develop higher quality products. While the benefits vary by individual, quality improvements of five to ten times are the norm and productivity improvements average about 25%.

Mr. Humphrey concludes with a summary of steps required to introduce these methods into software organizations.

About the speaker

Watts Humphrey joined the Software Engineering Institute (SEI) of Carnegie Mellon University after his retirement from IBM in 1986. While at the SEI, he established the Process Program, led the initial develoment of the Software Capability Maturity Model and introduced the concepts of Software Process Assessment and Software Capability Evaluation.

Prior to joining the SEI, he spent 27 years with IBM in various technical executive positions including the management of all IBM commercial software development. This included the first 19 releases of OS/360. Most recently, he was IBM's Director of Programming Quality and Process. Mr. Humphrey holds graduate degrees in Physics from the Illinois Institute of Technology and Business Administration from the University of Chicago. His books include A Discipline for Software Engineering, Managing the Software Process, *and* Managing for Innovation—Leading Technical People. *He was awarded the 1993 Aerospace Software Engineering Award presented by the American Institute of Aeronautics and Astronautics. He holds five US patents.*

Article to follow slides. Reprinted with permission of IEEE Software, pages 77-84, May 1996.

Using a Defined and Measured Personal Software Process

What if Your Life Depended on Software?

Watts S. Humphrey
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Sponsored by the U.S. Department of Defense

What if Lives Depended on the Software You Produce?

How would you behave?

What would you do?

Today we act as if software were not important

- **like paint or packaging**
- **if defective, we hope someone will fix it**
- **if late, we are not surprised and wait**
- **if hard to use, we call the hot line and pay for the answer**

The World Is Changing

Software now flies our airplanes.

Software accelerates, steers, and brakes our automobiles.

Software transfers our funds.

Software now impacts almost every aspect of our daily lives.

It Is Only a Matter of Time

Even widely used software has many defects.

Software now so pervades our lives that these defects will inevitably cause problems.

It is only a matter of time until defective or hard-to-use software

- **kills or maims people**
- **seriously disrupts businesses**
- **triggers military accidents or disasters**

The Public Is Increasingly Exposed to Software

They don't like what they see.

- **It is inconvenient.**
- **It is unreliable.**
- **It is not trustworthy.**

The software manufacturers are complacent

- **Their products are routinely late.**
- **They are full of defects.**
- **They charge for help.**
- **They expect the public to adjust to their problems.**

Why Do We Tolerate Shoddy Work?

It has not yet caused us serious problems.

It has not caused our businesses to fail.

It has not threatened our livelihoods.

It has not caused us physical harm.

Must We Have a Disaster to Recognize Software As Critical?

Think about this before the disaster.

The trends are obvious.

Software now pervades every aspect of our lives.

It is only a question of time until something happens to galvanize public action.

What Should You Do?

The practices in other fields are instructive.

When people's lives are at stake
- **the workers are trained and disciplined**
- **incidents are reviewed and corrections made**
- **examples are pilots and surgeons**

For critical financial work
- **trained professional auditors are used**
- **they work to exacting standards**
- **infractions are prosecuted**
- **examples are CPAs and investment advisors**

This Is Hard Work

Software engineering is extraordinarily difficult.

- **The engineers start with a vague application definition.**
- **They produce a precise program.**
- **A computer literally interprets every instruction.**
- **Any simple mistake could cause an operational disruption or even a disaster.**

These Are Complex Products

Small programs have thousands of lines of code (LOC).

- **A line of code is an instruction to a computer.**
- **It is like a name and address in a phone book.**

Large programs have millions of LOC.

- **These lines must each be produced by hand.**
- **They must each be tested.**
- **And they must each be fixed when wrong.**

Why Do Programs Have Defects?

When people do complex tasks, they inevitably make mistakes.

- **On average, experienced programmers make a mistake about every 10 LOC.**
- **They fix about half of these when they translate the program to machine language.**
- **They fix more defects during test.**

They need to fix most defects before the program will run at all.

Defect Removal Is Expensive

With today's practices, tested programs generally still have a lot of defects.

Computers will run defective programs.

- **When they hit a defect, these programs don't do what they are supposed to do.**
- **They may even cause harm.**

The longer defects remain in programs

- **the more they cost to fix**
- **the more likely they are to cause disruption**

Visualize a Million Lines of Code

Consider a phone book for a metropolitan area.

- **The Sarasota, Florida phone book has about 250,000 names and addresses.**
- **Each name and address is like a program instruction.**
- **The Sarasota phone book is 1.25" thick.**

A 1,000,000 LOC program would be like a stack of 4 phone books.

- **Windows 95 would take 46 phone books.**
- **This would be a 5 foot stack of phone books.**

Writing Large Programs

Using today's methods

- **engineers produce every instruction by hand**
- **they test to find the defects they injected**
- **and then they correct them**

A 1,000,000 LOC program starts with 100,000 defects.

- **Engineers typically find and fix about 95% of them before they finish test.**
- **This still leaves 5,000 defects in the finished program.**
- **Any remaining defect could cause problems.**

One Way To Think About Defects

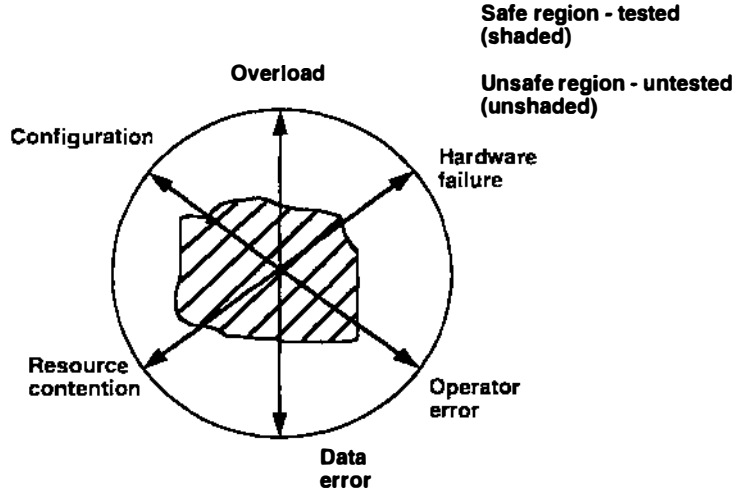
Think of defects as land mines.

- **They are hard to find.**
- **They don't cause problems until you stumble across them.**
- **You could then be in serious trouble.**

When engineers only test, it is like following a road through a mine field: you are only safe if someone has been there before.

Disciplined engineers use reviews and inspections to clear the entire field.

Stress Operating Regions



It Is Not the Engineers' Fault

Software engineering is demanding work.

- **Software engineers are among the smartest people in your organization.**
- **They are hard working.**
- **They are dedicated.**

The problem is that we do not properly train and manage software engineers.

Software Engineers Are Not Trained in Management Skills

They do not understand why they should plan and manage their personal work.

They do not know how to use standard management methods.

They do not know or practice the commitment discipline.

There Are No Quality Standards

When managers only ask about schedules, they imply that nothing else is important.

When did a manager last ask you about quality, usability, or documentation?

Does the program have to run?

- **How many defects can it have?**
- **Once it passes acceptance test, do you care if it fails?**
- **What does service cost?**
- **Do you care about the customers' opinions?**

Software Engineers Are Not Trained in Quality Methods

They think of defects as bugs.

- **Bugs are viewed as annoyances that crept in from somewhere outside.**
- **These are not bugs, they are defects.**
- **Programs with defects are defective.**

Defects are injected by engineers.

- **The same engineers should remove them.**
- **Engineers must be motivated to remove every single defect.**

What If Software Was Critical?

If the performance of your software products were a life or death matter, what would you do?

You would measure and track every defect to see how to better find and prevent it.

You would get help to ensure that you found every defect at every process stage.

You would practice writing defect-free programs with every project you do.

This Is Not Rocket Science

Plan and track your work.

Use defined and proven development practices.

Follow a rigorous quality and measurement program.

Treat the quality of every program you write as absolutely critical.

Set personal goals and manage to them.

Get the Proper Training

With the commitment discipline, engineers

- **plan and track their personal work**
- **commit only to planned work**
- **strive to meet every commitment**

With a personal quality discipline, engineers

- **use defined design, review, and testing methods**
- **strive for defect free work**
- **track and improve their quality performance**
- **set more challenging goals for the future**

Others Are Doing This Today

The Capability Maturity Model℠ (CMM℠) provides the management framework.

The Personal Software Process (PSP) provides the engineering discipline.

These are related and complementary.

- **You must do both.**
- **They reinforce each other.**
- **They depend on each other.**
- **Start with the CMM.**
- **Once started, begin PSP introduction.**

Capability Maturity Model and CMM are service marks of Carnegie Mellon University.

Results Have Been Remarkable

The CMM has been widely used in government and industry.

- **Cost and schedule performance improves dramatically.**
- **Return on investment is five times.**

The PSP has been taught to hundreds of engineers.

- **Quality improvements are five to ten times.**
- **Average productivity improvements are 20%.**

Software Process Evolution

Process Control	**5 Optimizing** - focus on process improvement
Process Measurement	**4 Managed** - process measured and controlled
Process Definition	**3 Defined** - process characterized, fairly well understood
Basic Mgt Control	**2 Repeatable** - previously mastered tasks can be repeated
	1 Initial - unpredictable & poorly controlled

CMM[SM] Improvement

The CMM provides a 5-level maturity structure.

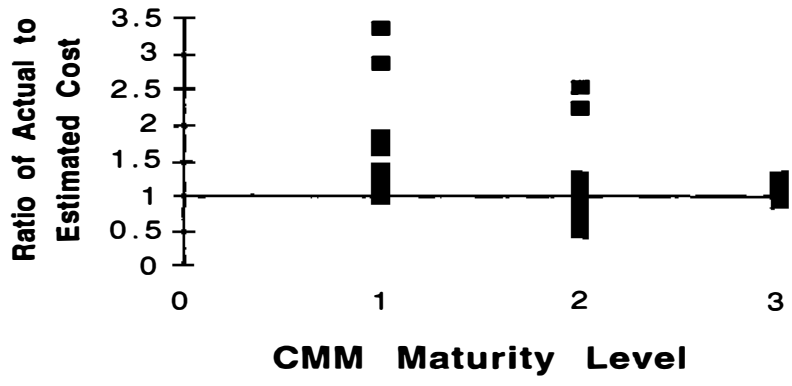
- **Organizations start at Level 1.**
- **They advance through each level.**
- **Their performance improves with each step.**

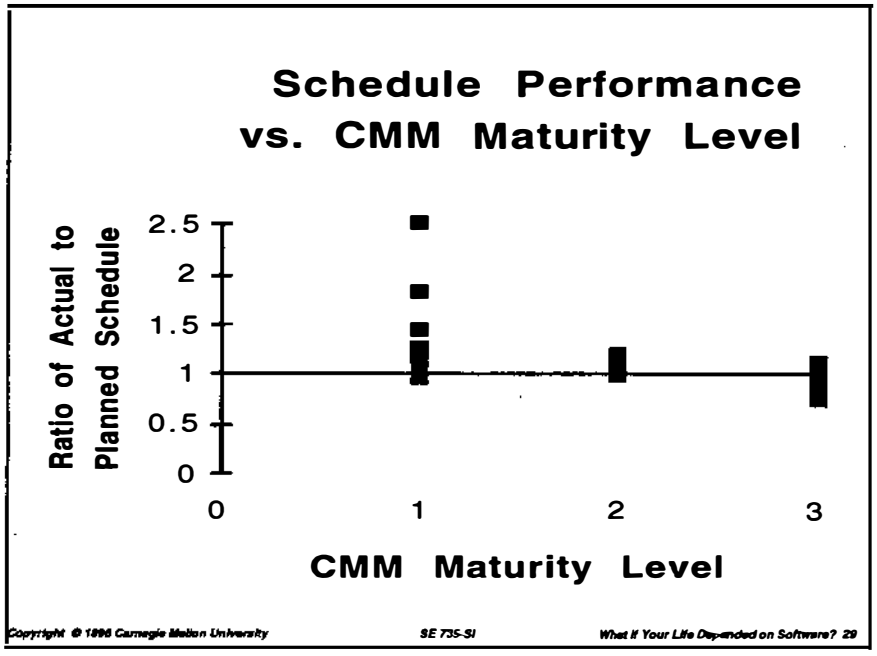
The following charts show

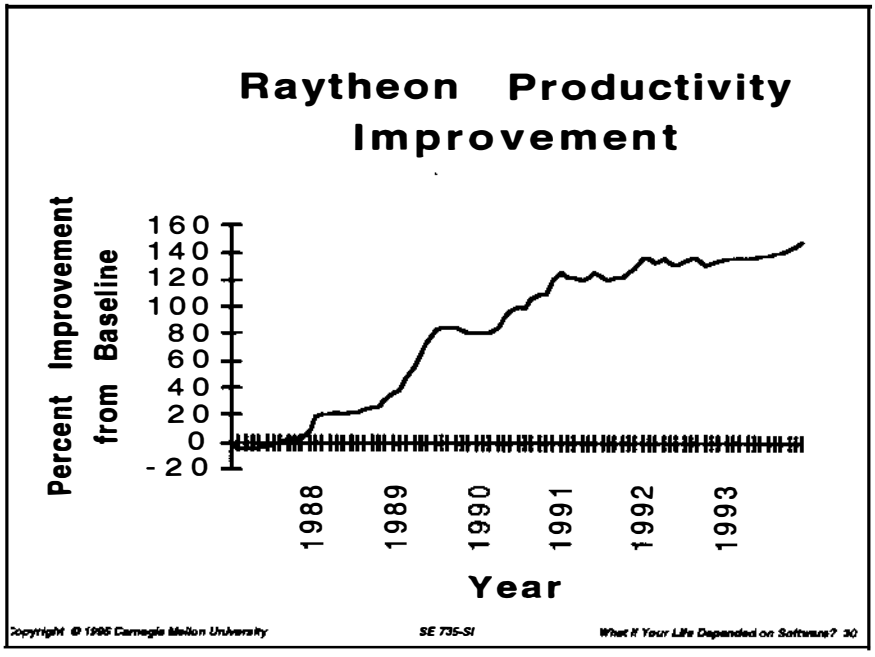
- **an Air Force study of 31 organizations**
- **productivity improvement data from Raytheon**
- **Loral (formerly IBM) data on the quality of NASA space shuttle software**

Capability Maturity Model (CMM) is a service mark of Carnegie Mellon University

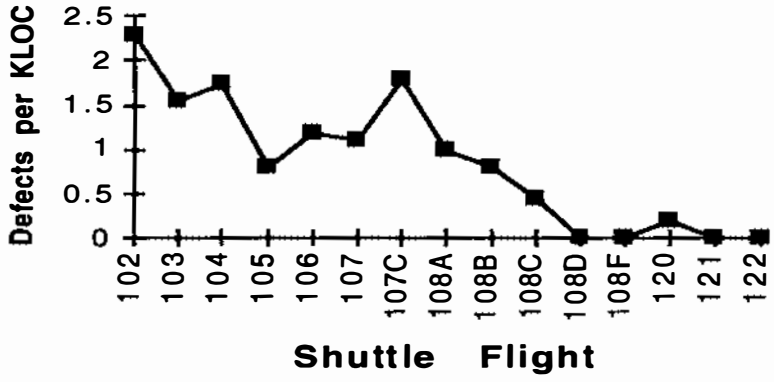
Cost Performance vs. CMM Maturity Level




Schedule Performance
vs. CMM Maturity Level
Ratio of Actual to Planned Schedule
2.5
2
1.5
1
0.5
0
0
1
2
3
CMM Maturity Level
Copyright © 1996 Carnegie Mellon University
SE 735-SI
What If Your Life Depended on Software? 29


Raytheon Productivity
Improvement
Percent Improvement from Baseline
160
140
120
100
80
60
40
20
0
-20
1988
1989
1990
1991
1992
1993
Year
Copyright © 1996 Carnegie Mellon University
SE 735-SI
What If Your Life Depended on Software? 30

Shuttle Delivered Defect Rates



Other Results Are Similar

Median results from an SEI study of 13 software organizations

Yearly cost of improvement	$245,000
Years engaged in improvement	3.5
Cost per software engineer	$1,375
Productivity gain per year	35%
Yearly reduction in time to market	19%
Yearly reduction in post-release defects	39%
Business return per dollar invested	$5.0

The CMM Is a Proven Road Map

If your organization is not using it, you should.

It is time to start.

The results will pay in the long run.

It does take
- **up front investment**
- **time**
- **strong management support**

The CMM and the PSP - 1

The Capability Maturity Model (CMM) is a management framework for improving software process capability.

The CMM provides
- **an orderly framework in which to do good engineering**
- **the infrastructure and resources to define and improve the organization's processes**

The CMM and the PSP - 2

The PSP is an individual discipline for doing superior software engineering.

It can be applied to every phase of the software process.

The PSP provides

- **the principles and methods engineers need to define, use, and improve their personal performance**
- **the knowledge, skill, and conviction to use these methods in practice**

The CMM and the PSP - 3

The CMM assumes that

- **engineers know and use effective methods**
- **engineers will participate in defining, using, and improving their processes**

The PSP assumes that there is

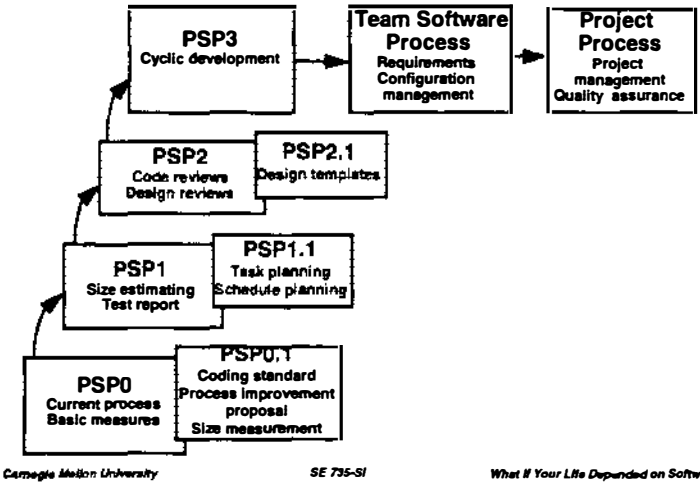
- **an orderly and managed framework within which to work**
- **the infrastructure to support a competent software engineering process**

The PSP Covers Disciplined Software Engineering

It shows engineers how to
- **define their personal processes**
- **estimate, plan, and track their work**
- **produce defect free products**

PSP results show
- **improved estimating and planning**
- **sharply fewer defects**
- **higher productivity**

The PSP Is an Evolving Process



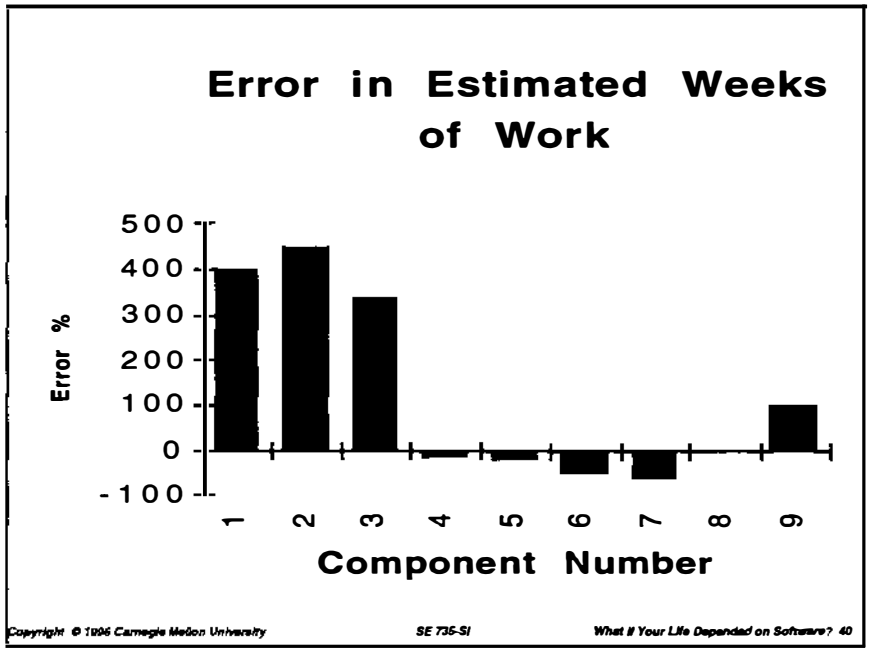
Estimating Accuracy

Three engineers at Advanced Information Services Corporation (AIS) were PSP trained.

Before training, they started developing a product and completed 3 components.

After training, they completed components 4 through 9.

Their schedule estimating error was 394.4% before training and -10.9% after training.



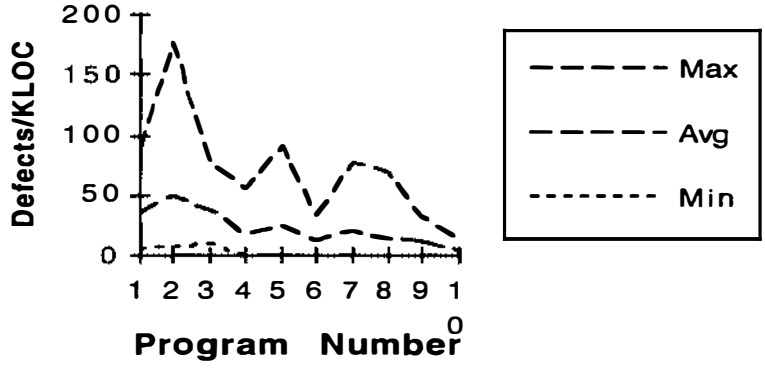
Quality Improvement

The number of defects found in test is an indication of the number of defects remaining in the product.

- **When a product is tested, a percentage of the defects is found.**
- **A percentage is also not found.**
- **Thus, when more defects are found, there are likely to be more that were not found.**

Over the 10 PSP exercises, engineers typically reduce average test defects by 5 to 10 times.

Defects Found in Test - Range



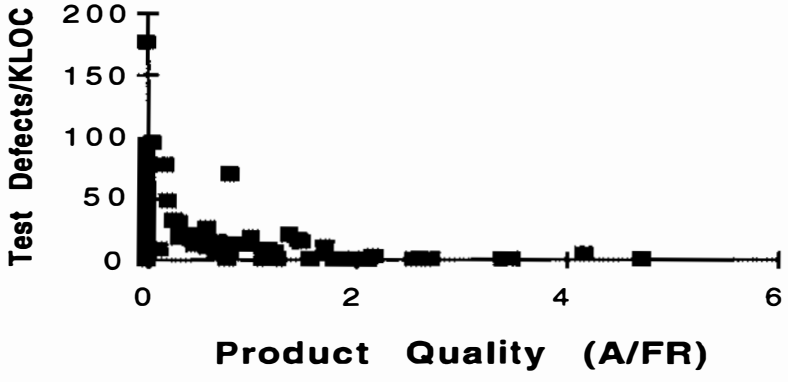
Managing Product Quality

By managing their process quality, engineers can consistently produce programs with few if any defects.

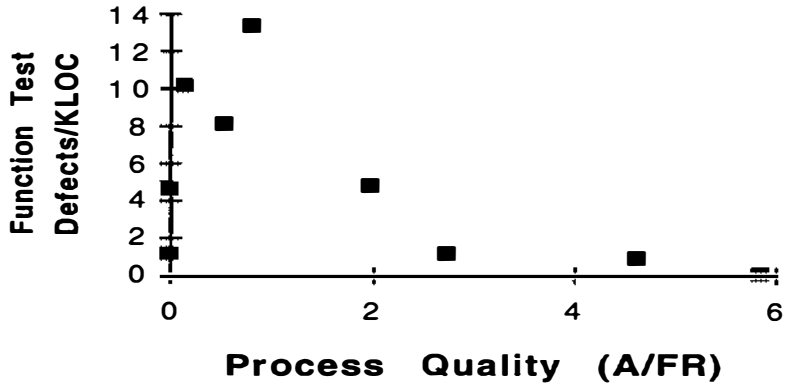
Process quality is measured in terms of A/FR, the ratio of time spent appraising divided by the time spent fixing the product.

Results show that with A/FR values above 2.0, few if any defects remain in the product.

Test Defects vs. Product Quality (A/FR)



After PSP Defects/KLOC vs. Process Quality



Productivity Improvement

In the PSP, productivity is measured in LOC per development hour.

LOC rates for 104 engineers
- **improved an average of + 20.84%**
- **had median change of - 0.58%**

While results vary by individual, average productivity is moderately improved.

This LOC per hour improvement is with substantially better quality.

Training for the PSP

The academic course takes one semester.

- **It is taught at a number of universities.**
- **The graduates are actively sought.**

The Software Engineering Institute (SEI) offers a two-week (full time) industrial course.

- **The two weeks are separated by a month.**
- **Homework is required between weeks and after the second week.**
- **A third week provides instructor training.**

Text: *A Discipline for Software Engineering.*

Why Doesn't Everybody Do It?

The software community is complacent: if it ain't broke, don't fix it.

Software education is done by scientists.

- **They view software as a technical subject.**
- **They do not teach management or disciplined engineering practices.**

Process improvement takes investment, management attention, and a lot of work.

But the Results Are Worth It

Every $1 invested in Capability Maturity Model (CMM) improvement, returns $5.

The Personal Software Process (PSP) reduces defects and improves planning accuracy at equal or better productivity.

Improvement takes time and money, but above all, it takes leadership.

And it could even prevent a disaster.

Finally, Remember Three Things

Process improvement works, both for organizations and engineers.

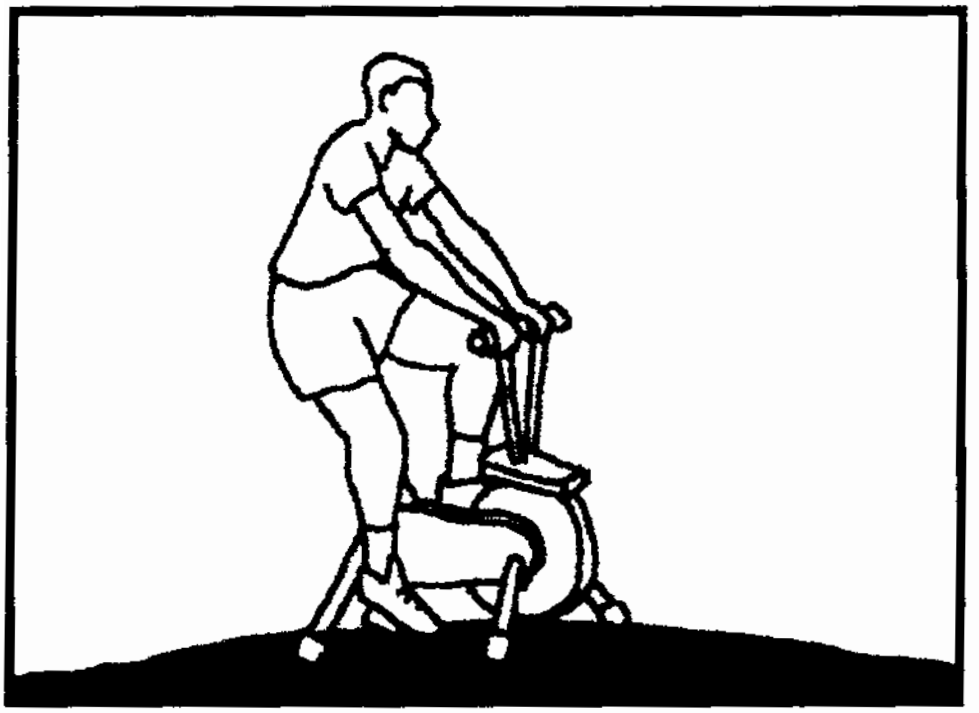
The methods are known and available.

It requires that

- **you learn the best available methods,**
- **you commit yourself to only producing quality products,**
- **you practice these methods with every project you do.**

Using A Defined and Measured Personal Software Process

Improved software processes lead to improved product quality. The Personal Software Process is a framework of techniques to help engineers improve their performance — and that of their organizations — through a step-by-step, disciplined approach to measuring and analyzing their work. This article explains how the PSP is taught and how it applies to different software-engineering tasks. The author reports some promising early results.



WATTS S. HUMPHREY
Software Engineering Institute

Fewer code defects, better estimating and planning, enhanced productivity — software engineers can enjoy these benefits by learning and using the disciplines of the Personal Software Process. As a learning vehicle for introducing process concepts, the PSP framework gives engineers measurement and analysis tools to help them understand their own skills and improve personal performance. Moreover, the PSP gives engineers the process understanding they need to help improve organizational performance. Up to a point, process improvement can be driven by senior management and process staffs. Beyond Level 3 of the Software Engineering Institute's Capability Maturity Model, however, improvement requires that engineers apply process principles on an individual basis.[1]

In fact, it was because of the difficulties small engineering groups had in applying CMM principles that I developed the PSP. Large and small organizations alike can benefit from CMM practices, and I focused the original PSP research on demonstrating how individuals and small teams could apply process-improvement methods.

In this article, I describe the PSP and experiences with teaching it to date. Thus far, the PSP is introduced in a one-semester graduate-level course where engineers develop 10

TABLE 1
PSP EXERCISES

Program Number	Brief Description
1A	Using a linked list, write a program to calculate the mean and standard deviation of a set of data.
2A	Write a program to count program LOC.
3A	Enhance program 2A to count total program LOC and LOC of functions or objects.
4A	Using a linked list, write a program to calculate the linear regression parameters (straight line fit).
5A	Write a program to perform a numerical integration.
6A	Enhance program 4A to calculate the linear regression parameters and the prediction interval.
7A	Using a linked list, write a program to calculate the correlation of two sets of data.
8A	Write a program to sort a linked list.
9A	Using a linked list, write a program to do a chi-squared test for a normal distribution.
10A	Using a linked list, write a program to calculate the three-parameter multiple regression parameters and the prediction interval.
1B	Write a program to store and retrieve numbers in a file.
2B	Enhance program 1B to modify records in a file.
3B	Enhance program 2B to handle common user errors.
4B	Enhance program 3B to handle further user error types.
5B	Enhance program 4B to handle arrays of real numbers.
6B	Enhance program 5B to calculate the linear regression parameters from a file.
7B	Enhance program 6B to calculate the linear regression parameters and the prediction interval.
8B	Enhance program 5B to sort a file.
9B	Write a program to do a chi-squared test for a normal distribution from data stored in a file.
Reports	
R1	LOC counting standard: Count logical LOC in the language you use to develop the PSP exercises.
R2	Coding standard: Provide one logical LOC per physical LOC.
R3	Defect analysis report: Analyze the defects for programs 1A through 3A.
R4	Midterm analysis report of process improvement.
R5	Final report of process and quality improvement and lessons learned.

module-sized programs and write five analysis reports. Early results are encouraging — while individual performance varies widely, data on 104 students and engineers show reductions of 58 percent in the average number of defects injected (and found in development) per 1,000 lines of code (KLOC), and reductions of 71.9 percent in the average number of defects per KLOC found in test. Estimating and planning accuracy are also improved, as is productivity — the average improvement in LOC developed per hour is 20.8 percent.

You can apply PSP principles to almost any software-engineering task because its structure is simple and independent of technology — it prescribes no specific languages, tools, or design methods.

PSP OVERVIEW

A software process is a sequence of steps required to develop or maintain software. The PSP is supported with a textbook and an introductory course.[2] It uses a family of seven steps tailored to develop module-sized programs of 50 to 5,000 LOC. Each step has a set of associated scripts, forms, and templates. During the course, engineers use the processes to complete the programming and report exercises shown in Table 1. As engineers learn to measure their work, analyze these measures, and set and meet improvement goals, they see the benefits of using defined methods and are motivated to consistently use them. The 10 PSP exercise programs are small: the first eight average 100 LOC and the last two average 200 and 300 LOC, respectively. Completing these programs, however, takes a good deal of work. While a knowledgeable instructor can substantially assist the students, the principal learning vehicle is the experience the students gain in doing the exercises.

When properly taught, the PSP

- demonstrates personal process principles,
- assists engineers in making accurate plans,
- determines the steps engineers can take to improve product quality,
- establishes benchmarks to measure personal process improvement, and
- determines the impact of process changes on an engineer's performance.

The PSP introduces process concepts in a series of steps. Each PSP step, shown in Figure 1, includes all the elements of prior steps together with one or two additions. Introducing these concepts one by one helps the engineers learn disciplined personal methods.

Personal Measurement (PSP0) is where the PSP starts. In this first step, engineers learn how to apply the PSP forms and scripts to their personal work. They do this by measuring development time and defects (both injected and removed). This lets engineers gather real, practical data and gives them benchmarks against which they measure progress while learning and practicing the PSP. PSP0 has three phases: plan-

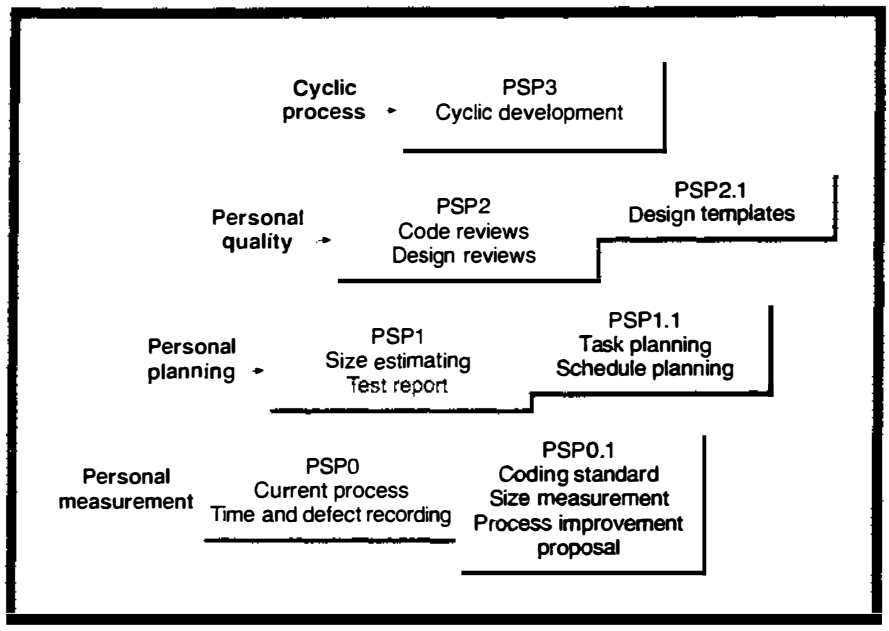


Figure 1. *PSP process evolution.*

ning, development (which includes design, code, compile, and test), and postmortem. PSP0.1 adds a coding standard, size measurement, and the Process Improvement Proposal form. The PIP form lets engineers record problems, issues, and ideas to use later in improving their processes. They also see how forms help them to gather and use process data.

Personal Planning (PSP1) introduces the PROBE method. Engineers use PROBE to estimate the sizes and development times for new programs based on their personal data.[2] PROBE uses linear regression to calculate estimating parameters, and it generates prediction intervals to indicate size and time estimate quality. Schedule and task planning are added in PSP1.1. By introducing planning early, the engineers gather enough data from the 10 PSP exercises to experience the benefits of the PSP statistical estimating and planning methods.

Personal Quality (PSP2) introduces defect management. With defect data from the PSP exercises, engineers construct and use checklists for design and code review. They learn why it's important to focus on quality from the start and how to efficiently review their programs. From their own data, they see how checklists can help them effectively review design and code as well as how to develop and modify these checklists as their personal skills and practices evolve. PSP2.1 introduces design specification and analysis techniques, along with defect prevention, process analyses, and process benchmarks. By measuring the time tasks take and the number of defects they inject and remove in each process phase, engineers learn to evaluate and improve their personal performance.

Scaling Up (PSP3) is the final PSP step. Figure 2 illustrates how engineers can couple multiple PSP2.1 processes in a cyclic fashion to scale up to developing

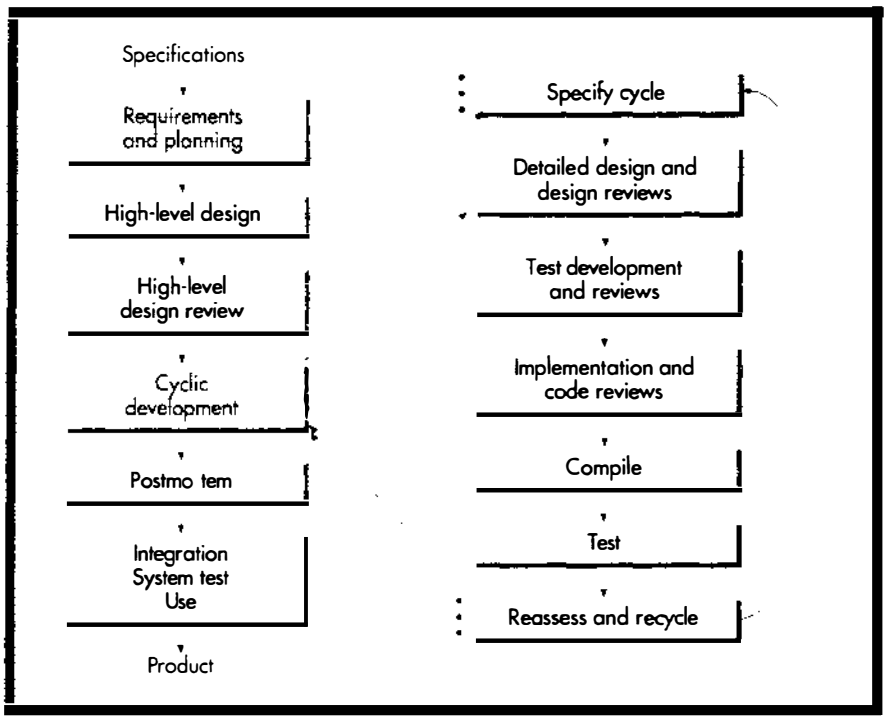


Figure 2. *PSP3 process.*

modules with as many as several thousand LOC. At this PSP level, engineers also explore design-verification methods as well as process-definition principles and methods.

PSP/CMM relationship. The CMM is an organization-focused process-improvement framework.[1,3] While the CMM enables and facilitates good work, it does not guarantee it. The engineers must also use effective personal practices.

This is where the PSP comes in, with its bottom-up approach to process improvement. PSP demonstrates process improvement principles for *individual engineers* so they see how to efficiently produce quality products. To be fully effective, engineers need the support of a disciplined and efficient environment, which means that the PSP will

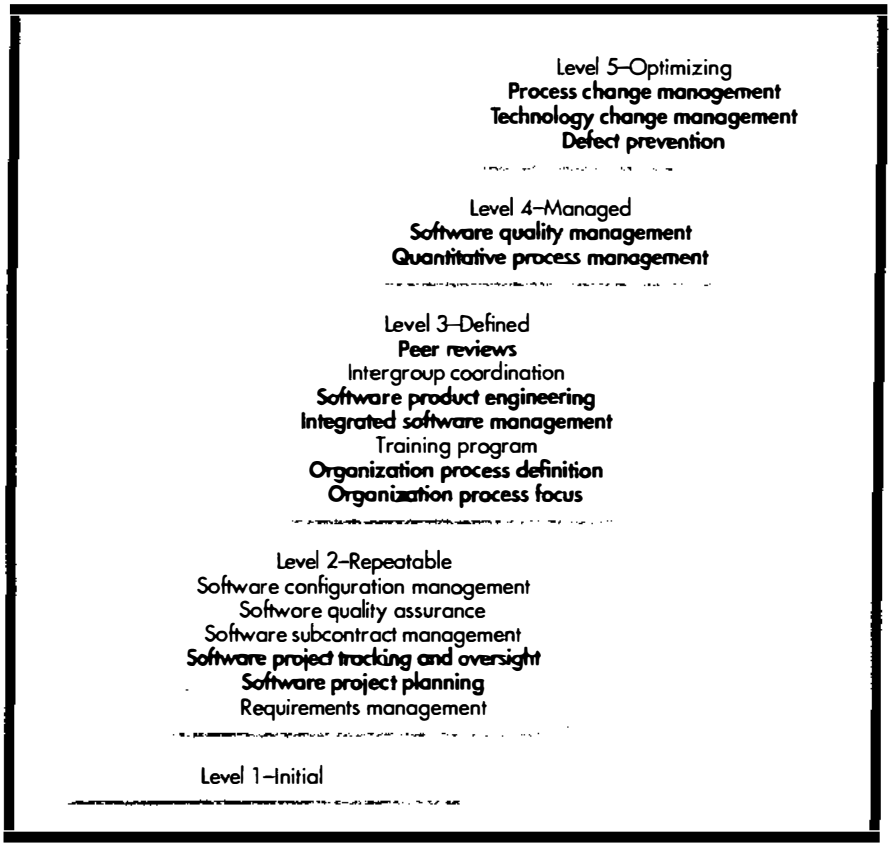


Figure 3. *PSP elements in the Capability Maturity Model, highlighted in bold face.*

be most effective in software organizations near or above CMM Level 2.

The PSP and the CMM are mutually supportive. The CMM provides the orderly support environment engineers need to do superior work, and the PSP equips engineers to do high-quality work and participate in organizational process improvement. As Figure 3 shows, the PSP demonstrates the goals of 12 of the 18 CMM key process areas. The PSP demonstrates only those that can be accommodated with individual, classroom-sized exercises.

PSP development. In the initial PSP experiments, I wrote 61 Pascal and C++ programs using a personal process as near to meeting the goals of CMM Level 5 as I could devise. I also applied these same principles to personal financial work, technical writing, and process development. This work showed me that a defined and measured personal process could help me do better work and that programming development is a compelling vehicle for introducing personal process management. A more complex process than other personal activities, software development potentially includes many useful measures that can provide engineers an objective evaluation of their work and the quality of their products.

After the initial experiments, I needed to demonstrate that the PSP methods could be effectively applied by other software engineers, so I had two graduate students write several programs using an early PSP version. Because this early PSP was introduced all at once in one step, the students had difficulty. They tried some parts of the PSP and ignored others, which meant they did not understand the overall process and could not measure its effect on their personal performance. This experiment convinced me that process introduction was important; thus, the seven-step strategy evolved.

Early industrial PSP experiments corroborated the importance of process introduction. Various groups were willing to experiment with the PSP but until these methods were introduced in an orderly phased way no engineer consistently used the PSP. In one group, for example, project engineers defined personal and team processes and committed to use them. Although a few gathered some process data and tried several methods, no one consistently used the full process. The problem appeared to be the pressure the engineers felt to complete their projects. Management had told them that using the PSP was more important than meeting the project schedules, but they still felt pressured and were unwilling to use unfamiliar methods.

Learning new software methods involves trial and error, but when faced with deadlines engineers are reluctant to experiment. While they might intellectually agree that a new practice is an improvement, they are reluctant to take a chance and generally fall back on familiar practices.

I was thus faced with a catch-22. Without data, I couldn't convince engineers to use the PSP. And unless engineers used the PSP, I couldn't get data. Clearly, to obtain industrial experience, I needed to first convince engineers that the PSP methods would help them do better work, so I decided to introduce the PSP with a graduate university course. By introducing PSP methods one at a time and with one or two exercises for each, this course would give engineers the data to demonstrate how well the PSP worked, without the pressure of project schedules.

PSP METHODS

Among the software-engineering methods PSP introduces are data gathering, size and resource estimating, defect management, yield management, cost of quality, and productivity analysis. I discuss these methods here with some examples that merge data for several PSP classes and for multiple programming languages. As you can see from the statistical analysis box on page 81, it makes sense to pool the PSP data in this manner.

STATISTICAL ANALYSIS OF PSP DATA

The analysis of variance test was applied to data for 88 engineers from eight PSP classes. Program sizes, development times, and numbers of defects found were all separately tested. The results are shown in Table A. Since F(80, 7) at 5 percent is 3.29, the null hypothesis cannot be rejected in any of the program 1 cases. For the analyses in this article, the various class data are thus treated as a single set.

Data for program 10, the last PSP exercise, was similarly examined. Here, the population examined was 57 engineers from five courses. The smaller population was used because two of the eight courses only completed nine of the exercises and one course had not completed at the time of the analysis. As Table A shows, the analysis of variance test showed that the null hypothesis could not be rejected. In this case, F(52, 4) at 5 percent is 5.63. Again, for this article, data from all the PSP classes are thus pooled for the analyses.

The analysis of variance test also examined potential performance differences caused by six different programming languages used in the PSP classes to date. Only three had substantial use, however: C was used by 46 engineers, C++ by 21, and Ada by 8. The other languages were Fortran, Visual Basic, and Pascal. Only data on C, C++, and Ada were tested. As Table A shows, the variances among individuals were substantially greater than those among languages, so the null hypothesis cannot be rejected and the data for all languages are pooled. Here, F(72, 2) at 5 percent is 19.5.

The Wilcoxon matched-pairs signed-rank test examined the significance of the changes in the engineers' performance between programs 1 and 10. The comparison was made for one class of 14 engineers for total defects found per KLOC, defects per KLOC found in compiling, and defects per KLOC found in testing. The T values obtained in these cases were 5, 1, and 0 respectively. For N=13 and 0.005 significance in the one tailed test, T should be less than 9. In all cases, these improvements thus had a significance of better than 0.005. A repeated measures test has also been run on these same parameters and all the changes were found to be significant.

Table A
Analysis of Variance — F Values

	Ratio of Variances		
Measure	**Program 1**	**Program 10**	**Languages**
Program size	2.09	2.37	0.450
Development time	1.20	1.87	1.460
Number of defects	0.65	3.31	0.042

Gathering data. The PSP measures were defined with the Goal-Question-Metric paradigm.[4] These are the time the engineer spends in each process phase, the defects introduced and found in each phase, and the developed product sizes in LOC. These data, gathered in every process phase and summarized at project completion, provide the engineers a family of process quality measures:

- size and time estimating error,
- cost-performance index,
- defects injected and removed per hour,
- process yield,
- appraisal and failure cost of quality, and
- the appraisal to failure ratio.

Estimating and planning. PROBE is a proxy-based estimating method I developed for the PSP that lets engineers use their personal data to judge a new program's size and required development time. Size proxies, which in the PSP are objects and functions, help engineers visualize the probable size of new program components. Other proxies — function points, book chapters, screens, or reports — are also possible.

The PSP estimating strategy has engineers make detailed size and resource estimates. Although individual estimates generally have considerable error, the objective is to learn to make unbiased estimates. By coupling a defined estimating process with historical data, engineers make more consistent, unbiased estimates. When engineers estimate a new development in multiple parts, and when they make about as many overestimates as underestimates, their total project estimates are more accurate. The estimating measure is the percentage by which the final size or development time differs from the original estimates.

Overall, engineers' estimating ability improved moderately during the PSP course. At the beginning, only 30.8 percent of 104 engineers estimated within 20 percent of the correct program size. For program 10, 42.3 percent did. For time estimates, 32.7 percent of these 104 engineers estimated within 20 percent of the correct development time for program 1 and 49.0 percent did for program 10.

In general, individual estimating errors varied widely. Some engineers master estimating skill more quickly than others, so it was no surprise that some engineers improved considerably while others did not. Even though 10 exercises can help engineers understand estimating methods, they generally need more experience both to build an adequate personal estimating database and to gain estimating proficiency. These data suggest, however, that by using PROBE most engineers can improve their ability to estimate both program size and development time.

Planning accuracy is measured by the *cost-performance index*, the ratio of planned to actual development cost. For the PSP course, engineers track the cumulative value of their personal CPI through the last six exercises.

Managing defects. In the PSP, all defects are counted, including those found in compiling, testing, and desk checking. When engineers do inspections, the defects they find are also counted. The reason to count all defects

TABLE 2
PSP DEFECT TYPES

Type Number	Type Name	Description
10	Documentation	comments, messages
20	Syntax	spelling, punctuation, typos, instruction formats
30	Build, package	change management, library, version control
40	Assignment	declaration, duplicate names, scope, limits
50	Interface	procedure calls and references, I/O, user formats
60	Checking	error messages, inadequate checks
70	Data	structure, content
80	Function	logic, pointers, loops, recursion, computation, function defects
90	System	configuration, timing, memory
100	Environment	design, compile, test, or other support-system problems

is best understood by analogy with filter design in electrical engineering. If you examine only the noise output, you cannot obtain the information needed to design a better filter. Finding software defects is like filtering noise from electrical signals — the removal process must be designed to find each defect type. Logically, therefore, engineers should understand the defects they inject before they can adjust their processes to find them.

A key PSP tenet is that defect management is a software engineer's personal responsibility. If you introduce a defect, it is your responsibility to find and fix it. If the defects are not managed like this, they are more expensive to find and fix later on.[1]

As engineers learn to track and analyze defects in the PSP exercises, they gather data on the *phases* when the defects were injected and removed, the *defect types*, the *fix times*, and *defect descriptions*. Phases are planning, design, design review, code, code review, compile, and test. The defect types, shown in Table 2, are based on Ram Chillarege's work at IBM Research.[3] The fix time is the total time from initial defect detection until the defect is fixed and the fix verified.

Defect trends for 104 engineers are shown in Figure 4. These are the engineers in the PSP classes for whom I have data and who have completed the 10 programming exercises. (Other engineers met these criteria but reported either incomplete or obviously incorrect results.) Of the 104 engineers, 80 took the PSP in university courses and 24 in industrial courses. Of the 80 university students, 16 were working engineers taking a night course and 28 were working engineers earning a graduate degree by returning to school full-time. Thus more than half of the engineers in this sample had worked in software organizations.

The top line in Figure 4 shows the average of the total number of defects found for each of the exercises. With program 1, the average is 116.4 defects per KLOC with a standard deviation of 76.9. By program 10, the average number of defects had declined to 48.9, and the standard deviation narrowed to 35.5.

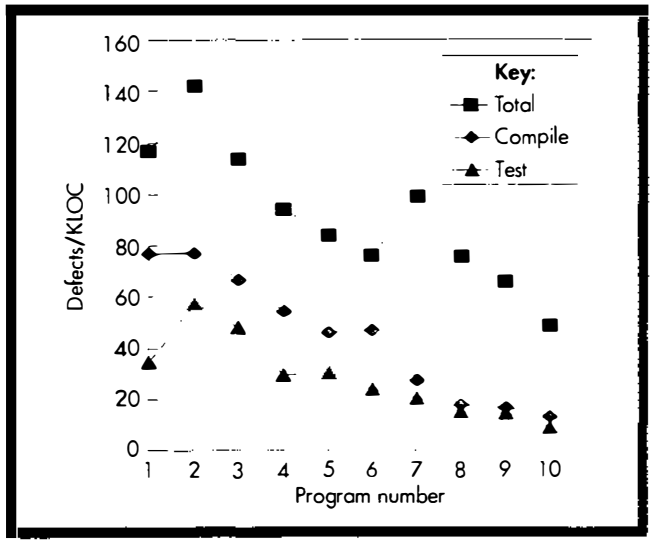


Figure 4. Defects per KLOC trend.

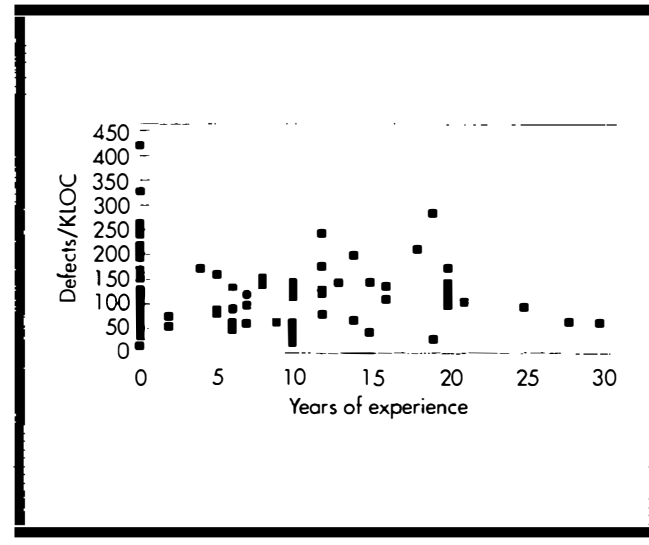


Figure 5. Defects versus experience, program 1.

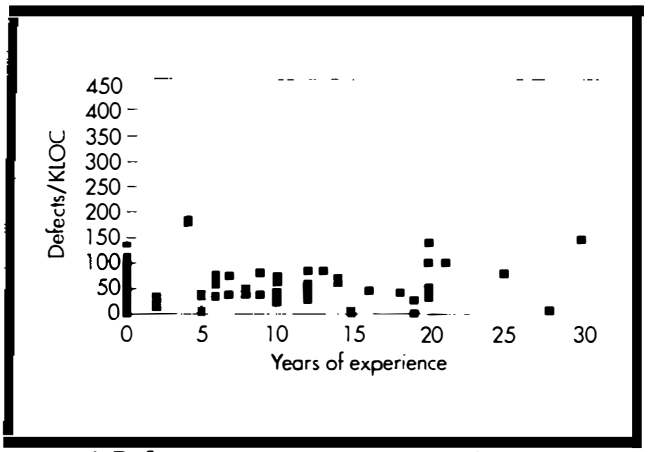


Figure 6. Defects versus experience, program 10.

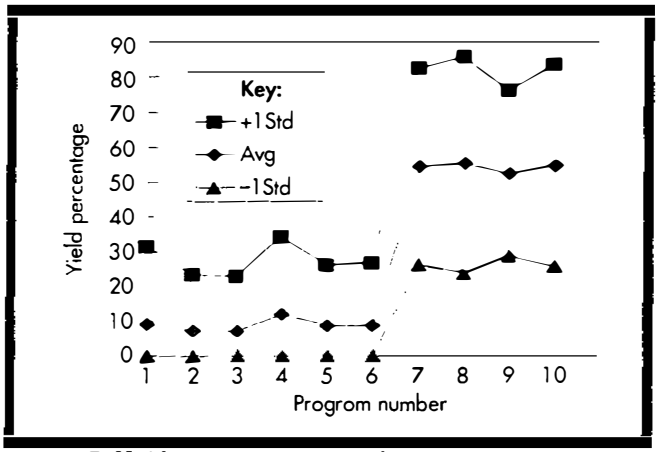


Figure 7. Yield versus program number.

The middle line in Figure 4 shows fewer defects found in compiling, from an average of 75.5 to 12.7 per KLOC, which is an improvement of about six times. The standard deviation narrowed from 58.7 to 12.7. For defects found in testing, the bottom line in Figure 4 shows reduced average defect levels, from 33.8 to 9.5 per KLOC, and reduced standard deviation, from 33.8 to 12.0.

Almost half (41) of these 104 engineers completed questionnaires, and the demographic data shows a modest relationship between defects per KLOC and years of engineering experience. While there is considerable variation in the defect rates for program 1, Figure 5 shows that the engineers with more than 20 years experience had somewhat lower defect rates than many less-experienced engineers, some of whom had low rates while many did not. As shown in Figure 6, the relationship between defect rates and experience does not hold for program 10. In fact, it appears that the less-experienced engineers learned the PSP methods better than their more experienced peers. Plots of defect levels versus both total LOC written and LOC written in the previous 12 months show no significant relationships.

Managing yield. Yield is the principal PSP quality measure. Total process yield is the percentage of defects found and fixed before the engineer starts to compile and test the program. Although software quality involves more than defects, the PSP focuses on defect detection and prevention because finding and fixing defects absorbs most of the development time and expense. When they start PSP training, engineers spend about 30 percent of their time compiling and testing programs, which probably mirrors their actual work practice. When engineers release actual modules for integration and system test, software organizations devote another 30 to 50 percent of development time in those phases,[6] almost exclusively to find and fix defects. Thus, despite other important quality issues, defect management will receive priority, at least until defect detection and repair costs are reduced.

If engineers want to find fewer defects in test, they must find them in code reviews. If they're going to review the code anyway, why not review it before compiling? This saves time they would have spent in compiling, and the compiler will act as a quality check on the code reviews. With few exceptions, however, engineers must first be convinced by their own data before they will do thorough design and code reviews prior to compiling.

In PSP, engineers must review their code before the first compile. Engineers often think the compiler's efficiency at finding syntax errors means they needn't bother finding them in reviews. However, some syntax defects will not be detected, not because the compilers are defective, but because some percentage of erroneous keystrokes will produce "valid" syntax that is not what the engineer intended. These defects cannot be found by the compiler and can be difficult to find in test. PSP data indicates that 9.4 percent of C++ syntax defects escape the compiler. If these defects are not found before compiling, they can take 10 or more times as long to find in unit test and, if not found in unit test, can take many hours to find in integration test, system test, or system operation.

The satisfaction that comes from doing a quality job is another reason to review code before compiling. Engineers like finding defects in code review, and they get great satisfaction from a clean first compile. Conversely, when they find few defects in code review, they feel they wasted their time. My personal experience also suggests that projects whose products have many defects in compile tend to have many defects in test. These projects also tend to be late and over budget.

Evidence shows that the more defects you find in compile, the more you are likely to find in test. Data on 844 PSP programs from 88 engineers show a correlation of 0.711 with a significance of better than 0.005 between the numbers of defects found in compile and those found in test. Thus, the fewer defects you find in compile, the fewer you are likely to have in test.

Reduced numbers of test defects imply a higher quality-shipped product. While it could be argued that finding few defects in test indicates poor testing, limited data show high correlation between the numbers of defects found in test and the numbers of defects later found by users. Martin Marietta, for example, has found a correlation of 0.911.[7]

Figure 7 shows the yield trends for our 104 engineers. Here, the sharp jump in yield with program 7 results from the introduction of design and code reviews at that point.

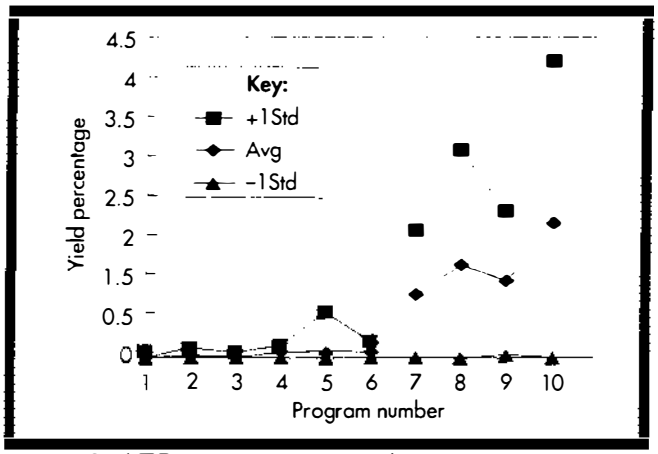


Figure 8. A/FR versus program number.

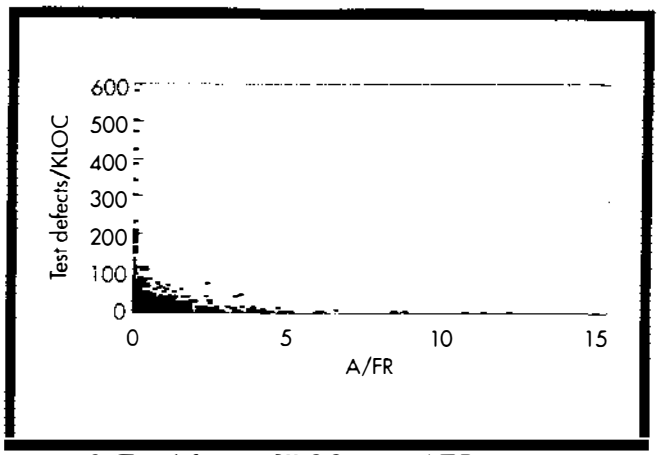


Figure 9. Test defects per KLOC versus A/FR.

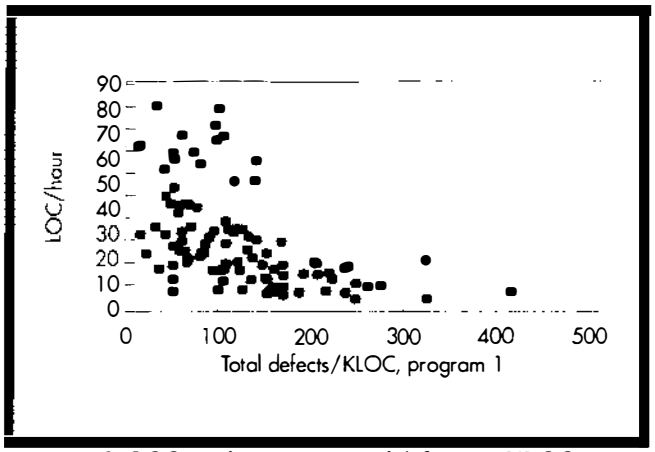


Figure 10. LOC per hour versus total defects per KLOC.

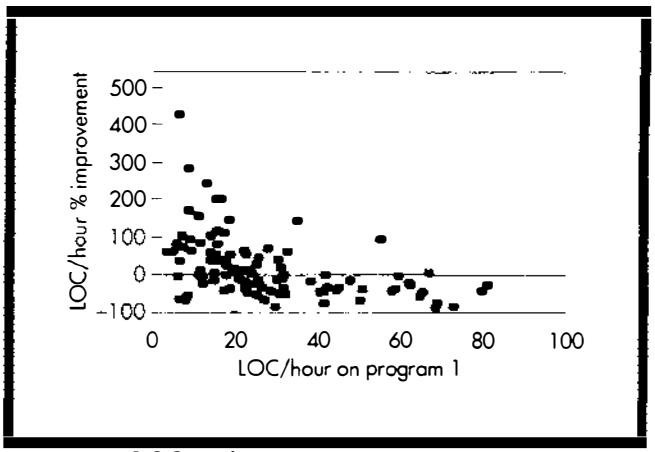


Figure 11. LOC per hour improvement.

Controlling cost of quality. To manage process quality, the PSP uses three cost-of-quality measures:

- appraisal costs: development time spent in design and code reviews,
- failure costs: time spent in compile and test, and
- prevention costs: time spent preventing defects before they occur. Prevention costs include prototyping and formal specification, methods not explicitly practiced with the PSP processes.

Another cost-of-quality measure is the ratio of the appraisal COQ to the failure COQ, known as the appraisal-to-failure-ratio. The A/FR is calculated by dividing the appraisal COQ by the failure COQ, or the ratio of design and code review time to compile and test time. The A/FR measures the relative effort spent in early defect removal. While the yield objective is to reduce the number of defects found in compile and test, the A/FR objective is to improve yield.

Figure 8 shows the improvement in A/FR for the same 104 engineers. Notably, A/FR increases with exercise 7 when design and code reviews are first introduced. Figure 9 shows data on A/FR and test defects for the 1,821 programs for which I have data. Here, A/FR values above 3 are associated with relatively few test defects while A/FRs below 2 are associated with relatively many test defects. PSP's suggested strategy is that engineers initially strive for A/FR values above 2. If they continue to find test defects, they should seek higher A/FR values. Once they consistently find few or no test defects, they should work to reduce A/FR while maintaining a high process yield.

Achieving higher product quality is the reason to increase A/FR. Once the quality objective is met, A/FR reductions will increase productivity. Since engineers generally cannot determine product quality during development, the A/FR measure is a useful guide to personal practice. By striving to increase their A/FR, engineers think more positively about review time. This helps them reduce compile and test time, and it reduces defects found in test.

The difference in time the engineers spend in compile and test shows how effective the A/FR measure can be. In one class, 75 percent of the engineers spent more than 20 percent of their time compiling and testing program 1. On program 10, only 8 percent did. Similarly, with program 1, no engineer spent less than 10 percent of the time compiling and testing, while with program 10, 67 percent did.

Understanding productivity. PSP-trained engineers learn to relate productivity and quality. They recognize that it makes no sense to compare the productivity of one programming process that found no test defects with one that had many. Defect-filled code will likely require many hours in integration and system test. Conversely, once engineers

Task-Oriented Information Sharing Among Software Developers

Frank A. Cioch

Department of Computer Science and Engineering
Oakland University
Rochester, MI 48309-4401
(810) 370-2183

cioch@oakland.edu

Abstract

Most software developers realize that when they are communicating with users they must try to take into account the perspective of their audience. This theme dominates the requirements analysis and user documentation literature. A popular way to accomplish this is to imagine the tasks that the product will be used to accomplish and present the product from this task-oriented viewpoint. What developers don't often realize is that even when they are communicating with other developers, it is not unusual for these other developers to assume *roles* similar to those of users. Thus, a software developer sometimes needs to take into account the user perspective even when presenting information to other developers.

This paper first explains the task-oriented viewpoint and how it differs from a features/topics-oriented viewpoint by reviewing how these different viewpoints are each used in the requirements analysis and user documentation domains. It then shows the variety of development situations that would benefit from a task-oriented viewpoint because the developer receiving the information is playing a user role. Finally, an example is used to present an approach on how to use the task-oriented viewpoint. The suggested approach is to present a list of task/subtasks that need to be performed to solve problems facing the developer along with the specific steps required to accomplish those tasks/subtasks. This approach forces developers to realize that during development other developers are assuming a user role and as such do not share their perspective. The resulting task-oriented document provides how-to-accomplish information which allows the information recipient to see the intent behind the software's features.

Keywords

useable and useful software, productivity improvement, risk assessment, software comprehension, software understanding, documentation, training, information sharing, technology transfer

Biographical Sketch

For over a decade at Oakland University Frank A. Cioch has taught graduate level software engineering and software quality courses to practicing professionals in the Detroit area. He received his Ph.D. from the University of Michigan, specializing in software comprehension. For the past three years he has been under contract at the U.S. Army Tank-Automotive Research, Development and Engineering Center, working on-site to facilitate software engineering practices. The task-oriented approach described in this paper has been used for information sharing among project team members.

Task-Oriented Information Sharing Among Software Developers

Frank A. Cioch
Oakland University

Introduction

There has been a great deal of interest in facilitating the sharing of technical information between software developers and users, particularly for requirements analysis and user documentation. This is required to avoid misinterpretations that result in costly rework and to improve product usefulness and usability. Recently, the focus has shifted away from taking a functional or features-oriented product perspective and toward taking a task-oriented user perspective. The basic idea is to learn what the user will be trying to do with the product and then describe in a procedural fashion how users actually work with the product to accomplish these specific tasks.

Use cases are a form of task-oriented specification that have recently become popular during requirements definition [Jacobson, 1992; Rumbaugh, 1994]. The specification includes a list of the tasks that the user of the product will want to perform (use cases) along with a description of the sequence of steps that must be followed to accomplish the task. Storyboarding [Andriole, 1989] is a task-oriented approach that is used to allow users and developers to collaboratively model the tasks the user will use the product to perform. The task-oriented method can also be found in customer-centered design [Holtzblatt and Beyer, 1993] and scenario-based design [Carroll, 1994], two approaches that incorporate how users actually work into the software design process.

The task-oriented approach is also well known in the user documentation domain. Rettig [1991] clearly states the goal of the task-oriented approach when he says that "[m]anuals should not just describe the features of a product, they should help people get things done." First in Williams and Beason's [1990] list of eight important documentation guidelines is that the writer must look at the product from the user's point of view and make the effort to translate the information about the software to information that is useful to the user. Brockmann [1990] makes the point that software developers must make the effort to move out of the functional perspective and think about *why* users need the information in order to meet their information needs. Weiss [1985] suggests that in order to understand the user perspective the software developer should view the document as a device whose function is to help users use the product without the reader having to actively figure out how to use the document/device.

All of these approaches share the following common objective: instead of describing the product in terms of its capabilities and features, the software developer is instead asked to think about

what tasks users will be trying to accomplish in the user problem domain and relate product features from the product domain to the work that needs to be done in the problem domain. Instead of being capabilities/features-oriented with a product perspective, the focus is task-oriented, shifting to the perspective of the user and describing how the product can be used to solve the users' problems.

The user documentation literature contains descriptions of both task-oriented and topics-oriented documentation and suggests when each is most useful. Both Weiss and Brockmann describe a topics-oriented manual as one that contains a complete description of everything that could be done with the product. It is organized around product features. Both authors suggest that this topics/features orientation is useful for reference manuals. These types of manuals are most useful after the user already knows how to work with the product and just needs to look things up. The reader must know enough about the structure of the product in order to determine the structure of the document for navigating purposes. A new user who needs to learn about the product will not know how to find the information they are looking for.

Both Weiss and Brockmann describe a task-oriented document as one that shows how to do specific things. It is organized around the specific tasks and procedures that are carried out by the user. The goal is to help the user get things done with the technology. The writer should provide only the information needed to help users accomplish their tasks. It has to fit the reader's method of working and require the least amount of attention and learning. This requires that the writer analyze what users do, how they use the product, and what information they need. Both authors suggest that this approach is well suited for tutorials for novice users. Weiss also suggests that experienced users would benefit from this approach in what he terms a demonstration document. This task-oriented document contains a description of the steps required to complete the tasks experienced users often want to carry out.

The user documentation literature makes the point that it is difficult to get the developer out of developer mode and to describe things from a user perspective. As both Weiss and Brockmann point out, awareness of techniques is alone not enough. They give many reasons.

First, developers have a tendency to view technology as an end in and of itself. They want to brag about the multitude of features that their product has and things that their product can do. The problem is that users want to learn how to do their tasks, solve their problems, and improve their performance. The developer is typically not interested in learning about the user domain and describing the product in user domain terms. It takes the focus away from the developers' interests and their accomplishments.

A related reason, given by Weinberg [1988], is that software developers are often poor communicators because they have a tendency to overload the listener with their output and err on the side of not observing/listening to the reaction of the information recipient.

Brockmann points out that it is difficult for developers to describe things from a user perspective because developers are accustomed to thinking in terms of how the product works. When asked to communicate their knowledge, it is easy to communicate what they know about all of the different features and how the product works. It requires too much effort to think about what users need to learn about the product, such as how it impacts their work or what tasks they want to use it to accomplish.

Another reason that it is difficult for developers to assume a user viewpoint is that they know how to use their product so well that it is unconscious and automatic. Adopting a user perspective requires that the developers write things down that are obvious and automatic to them. They simply can't put themselves in the position of the novice user. Asking a developer to document usage during, rather than after, development will alleviate, but not eliminate, the problem. In order to overcome the limitations of the developer mindset as it relates to software testing, it is not recommended that the same programmer develop and test [Davis, 1995]. A different viewpoint is required to expose errors resulting from misinterpreted requirements, unexpected usage, and omissions, oversights and unwarranted assumptions. The same may be true for documenting usage.

Task-Orientation For Developer/Developer Communication

The task-oriented approach to information sharing can be used to facilitate communication between developers rather than between developers and users. Developers are becoming increasingly aware that they must take a user perspective in order to develop useful and usable products. *But when it comes to communicating with other developers they often make the assumption that other developers share their perspective.* In practice, this assumption is erroneous. The following three scenarios illustrate this often overlooked point.

Scenario 1: Configuring Another Developer's Module

I recently examined documentation for a set of configuration files that had to be set up properly for a UNIX[1] process that was responsible for displaying the out-the-window view for a ground vehicle simulator. The types of parameters that had to be given values concerned the starting location of the vehicle in the world data base; the data base to be used; information about the number of articulated parts of the vehicle being displayed; the number of perspectives that could be used for viewing and the angular differences between them, etc.

The documentation was organized so that the configuration parameters were described in the order in which they appeared in the configuration file. All of the allowable values of the parameters were given, along with a short description of what the parameter was used to set. The parameter grouping was done in a way that reflected the use of the parameter in the program. This resulted in a product features organization, grouping together all parameters by relating them to a given feature of the UNIX process. For example, parameters related to IRIX Performer channel settings were together, parameters related to input files were together, etc.

In this situation the other developers are assuming a user role. The documentation was used when developers needed to make configuration file changes during product demos when the module developer was absent. They wanted to change the starting position of the vehicle, or use a different world data base, etc., to demo the ground vehicle simulator in different scenarios. What happened was that it was difficult to know the complete set of changes that needed to be made to accomplish these tasks, so team members were hesitant to make changes to the configuration files at all.

The module developer assumed that the developers who wanted to make configuration file changes would be taking a product perspective. A task-oriented perspective asks the module developer to first think of the types of changes that other developers would need to make during product demos, integration, or testing. For demos, the starting location of the vehicle in the world data base might be changed, a different world data base might be used, etc. The required changes to accomplish these typical tasks is documented. Upon completion, the resulting document has the following content: to start the vehicle do these things, to change the starting location do these other things, etc. The information is organized and presented in terms of tasks performed by other developers and the steps required to accomplish these tasks rather than by the module's features.

Scenario 2: Setting Up Support Software Written by Another Developer

A graduate student wrote start-up scripts that could be used to start UNIX processes distributed over multiple hosts. The scripts allow all processes to be started by using one pull down menu item on one of the hosts. The scripts had to be tailored when new hosts were added to the network or new processes were to be started. The scripts would be used any time developers needed to bring up the complete system - demos, product integration, product testing.

The graduate student wrote documentation in a features-oriented fashion, describing the script files and how they worked. It was complete, answering all of the questions that a script user might want to know and describing all of the capabilities and features of the scripts software. The graduate student assumed that the documentation reader would actually be interested in learning

about all of the capabilities and features of the scripts. What actually happened was quite different.

Developers had a great deal of difficulty making the changes to the scripts and were on the verge of abandoning them, even though they could have proved quite useful. The problem was that during the heat of development, no one wanted to take the time to read documentation about the scripts capabilities and features. They were focused on the task to be accomplished, usually product integration or testing, or the hasty preparation of the system for an unscheduled demo. To them the scripts were a means to an end, they did not want to learn about the scripts per se. The script documentation was written as though the developer were interested in learning about the scripts themselves. The mismatch resulted in the documentation being ignored and the scripts being set aside for a manual process.

Task-oriented documentation was written, organized by typical scripts modifications that were required during demos, integration and testing. For each task, the complete list of modifications that the script user had to make was given. These were followed in a "cookbook" fashion, but when the pressure of the task was over, the script user would go back and try to understand what was done. This resulted in developers both using the documentation and understanding the scripts. The stress previously felt when script modifications were required during demos, integration and testing was replaced by tedium.

Scenario 3: Assessing the Utility of a Software Component

A developer spent a good deal of time developing a software component that could be used to "visualize" information flow between UNIX processes. He was asked to give a demo to other developers to show them the component so that they could use it if they desired. In this case, use meant integrating the component into their code. This required both making slight modifications to the component code and including code in their module which properly interfaced to the component. This is a typical software component reuse situation [Mili, Mili and Mili, 1995], where component understanding (what it does, how it does it, and how to modify it so that it does something a little different) represents an important determining factor in reuse.

He spent approximately one-half hour with the potential users of the component in groups of two-to-three at a time, focusing on the features of the component. He assumed that the decisive factor in reuse was an understanding of component features. He demonstrated the ability of the component to calculate and present statistics on the frequency with which data was sent between processes, the ability to visualize the actual values of data values, the ability to see the relative times at which different data values were sent, etc.

After the demos the developers were asked whether they intended to use the component and about their reaction in general to the component. The response was negative to both questions. The problems were twofold: some developers could not imagine the ways in which the component could be useful to them. Others perceived a prospective use, but could not determine how much effort would be required in achieving the desired result with the component.

Both of these problems are the result of taking a features-oriented rather than a task-oriented approach to demonstrating the component. The developer shifted to a task-orientation in his demonstration. Specifically, the developer took two or three of the most promising uses and step-by-step showed how to set-up, configure, integrate, run and examine the output of the component to accomplish a minimal version of each of these uses.

This task-oriented approach answered many of the developers' questions and allowed them to judge more accurately both the potential usefulness of the component and how much effort was required to use the component. Most importantly, they were able to see the intent behind the component's features. By seeing the use of the features to actually solve problems of interest, the developers had a greater awareness of what was possible and a greater appreciation of the capabilities of the component.

Task-Orientation is Useful in a Variety of Development Situations

The above scenarios illustrate that when developers present information to other developers, they need to take into account the *role* that the other developer will be playing. Developers often take the role of *user* during software development, specifically during software integration and software reuse. For software module configuration, support software set-up, and reusable software component assessment, developers needing information about the software often assume a user role and could benefit from task-oriented information.

Module configuration of another developer's module, in both configuration files and in parameters in code header files, is often performed by a developer during product integration. When performing this task, the developer is in the role of a user of the other developer's module. The same is true for developers setting up support software or assessing the suitability of a software component for integration into a product. In these situations how-to-accomplish information is needed by other developers. This information should be presented as it would be to a user of the software, for that is the role that the developer is taking during that time. A task-oriented approach in these situations has the same benefits as when applied in the requirements definition and user documentation domains.

Taking a task-oriented perspective has also been shown to be beneficial during software design, when other developers can be viewed as users of your module [Cioch and Mili, 1990a; Cioch and Mili, 1990b]. This perspective is taken in Wirfs-Brock, Wilkerson and Wiener's [1990] collaborations/responsibilities model for designing object-oriented software.

Task-oriented documentation is used, if not actually produced, by developers during development. It is used widely for documenting procedures/processes that must be followed by members of the project team. Examples include checklists for inspections, test plans and procedures, integration processes, etc.

The next section contains a specific example of task-oriented documentation to show you what it looks like so that you see how to do it.

Example

This example concerns an interprocess communication module for distributed UNIX processes running in parallel. The UNIX processes constituting the system run in a parallel distributed fashion. Each team member is responsible for a single process. Architectural design consists primarily of defining the interprocess communication that must take place between the processes.

Interprocess communication is encapsulated within a single UNIX process together with a component that is linked in each of the other processes. This IPC process/component pair, called the IPC module, is the module of interest for this example. All data is sent between processes through the use of a set of logical send and receive functions provided by the IPC module. The data routing rules are encapsulated within the IPC module.

The audience for the IPC module's Operators Guide is the development team. The IPC module is a good example to illustrate the use of task-oriented documentation because all developers must understand how to use the IPC module to participate in system integration, system testing and system demos. It is also the focal point of many system performance issues.

A topics-oriented Operator's Guide for the IPC module might be structured along the following lines:

A. Module Configuration
    1. Configuration Files
    2. Code Parameters
        a. Continuous Data
        b. Event Data
        c. Debug Output
            i. Terse vs. Verbose

ii. Continuous Data Rate
iii. File vs. Screen Output
d. Performance Parameters
i. Receive/Send Ratio
B. Module Output
C. System Integration
D. Operating System Parameters
1. message queue limits
2. environment variables

The document has a product-centered structure. The Module Configuration section is completely organized around product concepts. All configuration file information is grouped together, as is all information concerning code parameters. Within the code parameters section, product features dominate, such as features that relate to continuous or event data, the complete set of debugging options, including screen vs. file output, and product performance parameters.

Product terminology creeps into the structure, including Terse, Verbose, Continuous Data Rate, and Receive/Send Ratio. These are product-oriented concepts that make sense to the developer of the product and must be learned by the user over time. This is not unusual.

The Operating System Parameters section contains a list of system parameters that affect the performance and capabilities of the IPC module. It is organized around operating system features, such as message queues and environment variables. Within each list, the operating system feature would be related to the performance capability of the IPC module.

For the reader of the document, a relation from operating system parameters to performance is the inverse of what is needed. Typically, a developer experiences a specific performance problem and needs to know which operating system parameters, if any, impact this particular problem. The required relation is from performance capabilities to operating system parameters, the inverse of that provided in the topics-oriented documentation. A task-oriented document would be structured around the specific performance problems facing the developer using the IPC module and describe the operating system parameters that might be useful in alleviating the problem.

Task-oriented documentation for the IPC module is structured around situations that are commonly encountered while running the system. Each situation is a problem that the developer has to solve or a task that the developer is responsible for completing.

Situation 1: You need to verify that data is being passed correctly between processes.

Situation 2: You are encountering error messages when sending data between processes.

Situation 3: You want to unit test your process independent of other developer's work but you need to send and receive data from other processes in order to perform your testing.

Each situation may have variations, describing in more detail the specifics of the problem facing the developer. For situation 1, there are a number of variations that are commonly encountered by team members:

Situation 1: You need to verify that data is being passed correctly between processes.

Variation 1: Data does not seem to be making it from one specific process to another specific process.

Variation 2: Continuous data that should be transferred in smooth increments (such as joystick or mouse movement) appears to be received in spurts with gaps in the data, making the data transfer appear jumpy and sporadic.

Variation 3: You are experiencing intermittent event loss. The event makes it from sender to receiver sporadically.

For each of these variations, a general plan for solving the problem has probably evolved over time and has been used successfully in the majority of cases. This is the general plan of attack that includes all of the obvious things that the experts have learned over time that can be done to systematically track down and solve the problem. These are the things that should be done before the developer goes and asks for help. If the problem is typical, such as a typical oversight, then the developer might find a solution. For a more obscure problem the developer will probably need to get help. This task-oriented style of documentation provides good places to document lessons learned information.

This problem solving plan is presented as a list of tasks/subtasks that need to be performed to solve problems. The general solution strategy embedded in the task/subtask list is also presented.

Variation 1: Data transfer problem. A specific data item does not seem to be making it from one specific process to another specific process.

In order to solve the problem, you need to determine whether the problem is on the sending side, the receiving side, or in the IPC module itself. A solution strategy is to follow the chain of data transfers that take place as the data finds its way from sender to receiver, looking for the place that the data is lost, overwritten, or ignored.

Task 1: Make sure that the expected data routing is correctly specified in the data routing table.

Task 2: Make sure that you are actually running the system configuration that you think you are.

Subtask 2.1: Make sure that the correct version of each process has been started.

Subtask 2.2: Make sure that these processes are both using the correct version of the IPC module.

Subtask 2.3: Make sure that all processes are accessing the correct IPC module configuration files.

Task 3: In the IPC module debugging header file, set the debugging parameters appropriately for this particular data transfer problem. Then re-compile the IPC module.

Task 4: Link each process so that it uses the newly compiled IPC module.

Task 5: Once the system is running, you need to read the debugging messages generated by the IPC module to help you track down the problem.

For each task, the set of specific steps to accomplish the task is then provided. This is the level at which the product, the IPC module, must actually be operated.

Task 3: In the IPC module debugging header file, set the debugging parameters appropriately for this particular data transfer problem. Then re-compile the IPC module.

If you are interested in tracing a continuous data item, set Continuous_Data_Trace to 1 (terse). This will allow you to see if the send and receive functions are actually being called and if so, if either is returning an error code. Otherwise, set it to 0.

If you are interested in tracing an event data item, set Event_Data_Trace to 1 (terse). This will allow you to see if the send and receive functions are actually being called and if so, if either is returning an error code. Otherwise, set it to 0.

If the data transfer is being done between processes located on different hosts, set Inter_Host_Trace to 1 (terse). Without this you won't be able to see exactly where along the chain the data is being ignored, overwritten, or lost. If instead all processes of interest are on the same host, set it to 0.

If you are interested in tracing a continuous data item, set File_Output to 1 (yes) because the shell window will continue to spew out debugging messages so long as the continuous data is being passed. You won't be able to read it. If you are tracing event data and you want to do your analysis at the terminal set it to 0 (no).

If you are interested in tracing a continuous data item set Continuous_Data_Rate to 1 because you are trying to figure out why the continuous data item does not appear to be getting from the sender to the receiver and you need to see debugging output for every single send and receive to track down the problem.

Task-oriented information typically contains 'why' information, as the example above does. It is a good way to begin educating the reader so that she can begin to learn to make decisions for herself. For example, after telling the reader to set the tracing parameter to 1 (terse) a sentence could have been included telling her why not to use verbose such as "verbose is used by the IPC module developer for IPC module debugging to track down possible operating system or network problems when no data is getting from senders to receivers." This type of expected usage information is often absent in topics-oriented documentation.

The topics-oriented document for the debugging code parameters might have the following flavor:

A. Module Configuration
   2. Code Parameters
      c. Debug Output
         i. Terse vs. Verbose

Terse means only inform of successful run-time sends and receives. Verbose means inform of all function calls during run-time sends and receives.

Continuous_Data_Trace is used to control debugging messages for continuous data. 0 = turn off 1 = terse 2 = verbose

Event_Data_Trace used to control debugging messages for event data. 0 = turn off 1 = terse 2 = verbose

Inter_Host_Trace used to control debugging messages for data sent between hosts. 0 = turn off 1 = terse 2 = verbose

ii. Continuous Data Rate

Continuous_Data_Rate represents the number of times a continuous data send/receive occurs before you get a line of debugging output.

iii. File vs. Screen Output

File_Output: used to determine whether debugging messages appear on screen or in a file. 0 = no 1 = yes

In the topics-oriented document these parameters will be described only once, while in the task-oriented document these parameters will be described more than once. To illustrate, the set of steps required for another variation of the data transfer problem will now be presented.

Situation 1: You want to verify that data is being passed correctly between processes.

Variation 2: Continuous data that should be transferred in smooth increments (such as joystick or mouse movement) appears to be received in spurts with gaps in the data, making the data transfer appear jumpy and sporadic.

This is a difficult problem to track down. One solution strategy that has proven successful is to look at the sends and receives of the sending process, the IPC process on host 1, the IPC process on host 2, and the receiving process and then examine the relative rates of sends and receives.

Task 3: In the IPC module debugging header file, set the debugging parameters appropriately for this particular data transfer problem. Then re-compile the IPC module.

Set Continuous_Data_Trace to 1 (terse). This will allow you to see if the send and receive are actually being called and if so, if either is returning an error code. Otherwise, set it to 0.

Set Event_Data_Trace to 0 (turn off). In this case, the output is not of interest and will be distracting.

Because the data transfer is being done between processes located on different hosts, set Inter_Host_Trace to 1 (terse). Without this you won't be able to follow the recommended solution strategy outlined above.

Set File_Output to 0 (no) because you will want to do your analysis at the terminal rather than at your desk.

Set Continuous_Data_Rate to 10,000. This will result in one line of output after 10,000 sends or receives. Under normal operating conditions this will cause a line of output every 5 seconds. For each of the four process of interest, see if this line appears at about the same time and at about the same steady rate. If one process experiences either delays or sporadic output, that might provide some clues as to the source of the problem.

Conclusions

The task-oriented perspective applied in user documentation and user-centered requirements and design can be beneficial when applied to developer/developer communication. The suggested approach is to present a list of task/subtasks that need to be performed to solve problems facing the developer along with the specific steps required to accomplish those tasks/subtasks. This approach forces developers to realize that during development other developers are assuming a user role and as such do not share their perspective. This is particularly important for software integration and software reuse. The resulting task-oriented document provides how-to-accomplish information which allows the information recipient to see the intent behind the software's features. The benefits are the same as those so often cited in the requirements

literature: Improved communication results in productivity gains due to the reduction of misinterpretations and the resulting rework.

The task-oriented approach is not without its problems. The resulting document can be long and expensive to produce in terms of scarce senior programmer time. A mitigating strategy for document length that has proved effective is to carefully pick only the few most important tasks and document those. Once developers get the main idea, they seem to be quite adept at figuring out how to do more. They seem to need a jump start with an appropriate big-picture perspective. Producing task-oriented documentation under a mentor has proved to be a good first assignment for new team members, thus reducing the cost of producing it.

Thus far, the task-oriented approach has been used only for developer-developer information sharing situations in which the developer receiving the information is assuming a user role. This happens during development in software integration and software reuse and after development in software testing. It is unclear whether the approach would be effective as a way for a developer to explain a design issue to another developer. For example, one might try the approach to see if a module creator can explain a design issue to a new developer assigned to maintain the module. This remains a future research issue.

Acknowledgments

This work was supported in part by the U.S. Army Tank-Automotive Research, Development and Engineering Center (Contract No. DAAE07-94-C-R006). I would also like to thank Mike Palazzolo for his contributions to the ideas expressed in this paper and Ian Savage for his thought-provoking review, which resulted in substantial improvements to the presentation.

References

Andriole, S.J., *Storyboard Prototyping: A New Approach to User Requirements Analysis*, QED Information Sciences, 1989.

Brockmann, R.J., *Writing Better Computer User Documentation*, Wiley, 1990.

Carroll, J.M., "Making Use a Design Representation," *Communications of the ACM*, Vol. 37, No. 12, December 1994, pp. 29-35.

Cioch, F.A. and F. Mili, "Information Sharing During Software Development and Maintenance," *Proceedings of the Eighth Annual Pacific Northwest Software Quality Conference*, 1990, pp. 335-345.

Cioch, F.A. and F. Mili, "Use-Perspective Unit Documentation", Proceedings of the 4th SEI Conference on Software Engineering Education, published in *Lecture Notes in Computer Science*, No. 423, Springer-Verlag, 1990, pp. 136-144.

Davis, A.M., *201 Principles of Software Development*, McGraw-Hill, 1995.

Holtzblatt, K. and H. Beyer, "Making Customer-Centered Design Work For Teams," *Communications of the ACM*, Vol. 36, No. 10, October 1993, pp. 93-103.

Jacobson, I., M. Christerson, P. Jonsson, G. Overgaard, *Object-Oriented Software Engineering*, Addison-Wesley, 1992.

Mili, H., F. Mili and A. Mili, "Reusing Software: Issues and Research Directions," *IEEE Transactions on Software Engineering*, Vol. 21, No. 6, June 1995, pp. 528-562.

Rettig, M., "Nobody Reads Documentation," *Communications of the ACM*, Vol. 34, No. 7, July 1991, pp. 19-24.

Rumbaugh, J., "Getting Started: Using Use Cases to Capture Requirements," *Journal of Object-Oriented Programming*, September 1994, pp. 8-23.

Weinberg, G.M., *Understanding the Professional Programmer*, Dorset House, 1988.

Weiss, E.H., *How to Write a Usable User Manual*, ISI Press, 1985.

Williams, P.A. and P.S. Beason, *Writing Effective Software Documentation*, Scott, Foresman and Company, 1990.

Wirfs-Brock, R., B. Wilkerson and L. Wiener, *Designing Object-Oriented Software*, Prentice-Hall, 1990.

---

[1] UNIX is a registered trademark of AT&T Bell Laboratories.

Establishing an SQA Intranet Web Site

Tim Farley

3616 Palatine Ave. N.
Seattle, WA 98103
(206) 545-0744
tfarley@accessone.com

Abstract

In August, 1995, the Quality Assurance Central Services Group of Wall Data Incorporated established an internal web site. The purpose of this internal web site was to distribute software standards, templates, and tools to software developers, quality assurance engineers, and project managers at the corporate headquarters and 10 satellite development offices in the US and Japan. Though implementing the web site was not difficult technically, getting the development community to use it was challenging. Obstacles included uninstalled or incorrectly configured web browser software, and general lack of interest. Overcoming these obstacles involved improving the design and content of the site, and changing existing SQA document distribution practices. Once these obstacles were overcome, use increased and the internal web site became the preferred mechanism for distributing SQA documents within the company. Other departments and development teams then began to develop their own web sites to improve the distribution of information.

Keywords

Standards, Electronic Document Publishing, Intranet, WWW, HTML.

Biography

Tim Farley has 9 years experience in SQA. He has worked as an SQA project engineer, SQA department manager, and corporate SQA staff engineer. He received a B.A. in Computer Science and a B.A. in Anthropology from Brown University.

BACKGROUND

Wall Data is a PC connectivity software company that prides itself on having strong, entrepreneurial development teams. As such, development teams are encouraged to locate away from the distractions of the corporate headquarters to focus on producing leading technology products. Wall Data now has 10 satellite development offices in Washington, California, Georgia, Massachusetts, and Japan.

The Quality Assurance Central Services group, or QACS, is a corporate resource located in the company headquarters. Part of the charter for this group is to develop standards and communicate "lessons learned" to all development teams throughout the company. Because of the limited QACS staff size, and the geographic diversity of the development teams, QACS often relied on e-mail, a monthly departmental newsletter, and a network-accessible SQA document archive to distribute information to these satellite development offices.

Though printed copies of some materials, such as standard operating procedures, were distributed directly to project managers, there was never any guarantee that all standards were communicated to all development team members. There was also no guarantee that e-mail or newsletters would be read, or that engineers would be able to successfully navigate the sometimes cryptic filenames and haphazard directory structure of the SQA document archive. Regardless of what information, paths, and file names were e-mailed out by QACS, development team members often ended up calling and asking to be e-mailed the actual document instead.

Wall Data's product line includes internet applications, including a version of Mosaic, a World Wide Web browser. As this browser is a Wall Data product, anyone in the company can access this application from the company network and install a copy on their desktop PC.

Creating an internal web site seemed like the perfect solution to the problem of distributing information to the satellite offices since it used the company's own application software and would create an easier way for development teams to access all SQA archive information. It would also provide QACS with the means to quickly inform development teams of process changes and new practices (without having to write a complete newsletter or produce mass e-mailings), and would put all information in the hands of engineers without relying on project managers to pass it along.

The QACS web site would be essentially an experiment for the company. At the time, no internal corporate web sites existed at Wall Data. Within QACS, the creation and maintenance of the web site would be the responsibility of a single engineer, and web site tasks would be secondary to regular QACS staff functions. There was no corporate mandate to create an internal web site, and no team dedicated to its creation. The QACS web site would be a grassroots solution to the problem of distributing information throughout the company.

WEB SITE DESIGNED AND IMPLEMENTED

In August, 1995, there were no internal web sites at the company and no resources for QACS to establish and maintain dedicated web and ftp servers. However, QACS was able to create an internal SQA web site using existing resources, including Wall Data's version of Mosaic and the internal company network. The QACS web site would be accessed as files directly from the company's network drives by using the "Open File" web browser command to open an HTML file, format it, and display it like any other web page. Lack of a web server would mean the web site would not be able to process on-line forms or count web site accesses, but it would be able to present hypertext links to documents to make accessing the QACS archive easier.

The web site was designed to meet the following goals:

(1) The site should not look like a corporate web site. Even though it would be used to distribute corporate standards, a casual look and feel would be appropriate for the company culture.
(2) The site should have a consistent look and feel regardless of what browser was used to access it. Only HTML commands supported by all browsers would be used to implement the site.
(3) The site content should give people a reason to visit at least once a week.

The initial design of the web site attempted to meet these goals as well as provide an easy to navigate interface to the QACS document archive. The web site was divided into the following pages:

Main - An initial page to welcome people to the site, explain the purpose, and present links to all the other topic pages of the site.

What's New - News and announcements from QACS. This page would be updated throughout the week as needed, at the very least each Monday.

Templates & Examples - Document templates and examples of completed documents.

Tools - Internally developed software testing tools.

Technical Bulletins - Internally developed guidelines, lessons learned, and "How To..." documents.

Standard Operating Procedures - Corporate standards for software development and testing.

Newsletter Archive - On-line versions of monthly SQA newsletters.

Gallery of Greatness - On-line photo gallery of development team members recognized by QACS for their quality contributions.

Wall Data - Link to the company's external Home Page.

Each topic page would have the same layout, with a unique title and icon to identify it, a brief paragraph describing the topic, the contents of the page (usually hypertext links to Microsoft Word for Windows documents), and navigation buttons to return to the main page or go to other topic pages. To access archive documents, the user would select a hypertext link from a topic page, which would start the Word application and then open the Word document. The user could then either read the document on-line or print a copy. They could also save a copy of the document or template to their local hard disk. Please see Appendices A and B for examples of the main page and a topic page.

The site was designed and implemented in 8 hours over a weekend using Microsoft Notepad, Adobe Photoshop, and ClickArt Art Parts.

The underlying directory structure of the SQA document archive was also changed. The files were rearranged to match the organization of the web site. Subdirectories for the HTML files were also created. Since all the HTML files and all the SQA archive documents resided on the same logical network drive, relative paths were used for all hypertext links. All pages and all links were tested using all browsers. Key development team members were also given early access to the site to get their feedback.

INTERNAL WEB SITE INTRODUCED

A special edition of the SQA newsletter was e-mailed to all development team members, project managers, and executives announcing the web site. The newsletter described the purpose of the web site, how to access it, how to install and configure a web browser, and what additions were being planned for the future. The web site was also used in a QACS presentation to executive management. E-mail and memos from QACS now contained the path to the web site and brief instructions how to use it.

Executive management and key development team members that saw presentations, or were given demos, seemed enthusiastic about the site. However, most development team members did not receive presentations or demos due to lack of QACS resources and higher priority projects. Most were expected to learn about the site through either the newsletter or e-mail. Though it should have come as no surprise, QACS only received about 10 comments from users over the first month, and continued to receive calls and e-mailed requests for documents or paths to templates. When told that they could get all the documents through the web site, many callers said they did not want to have to use another application just to open a file.

At this point it became clear that the QACS web site would not be flooded by users, and that there were some fundamental obstacles that would have to be overcome before the development community would start using it.

OBSTACLES UNCOVERED

Few development team members were in any position to try out the SQA web site immediately. Most did not have web browser software installed on their systems at the time the web site was introduced, and most had not accessed the World Wide Web before. Many also claimed to not have the time or the disk space to install the browser application. They also felt they rarely needed anything from the SQA document archive. To most, installing a web browser seemed like a lot of work with little benefit.

Those that did install the browser software often did not read the instructions in the newsletter on how to configure it to recognize Word documents. While some web browsers will prompt the user for an application when an unknown file type (.doc) is encountered, the early version of our Mosaic product did not. It treated the unknown file type as a text file. For users accessing Word documents without first configuring their browser, the result was a screen full of garbage characters. This often left users with the impression that the web site did not work and gave them little reason to check back at the site for news and updates.

Another problem with the early version of our web browser was that users had to manually clear their disk cache or request a page reload in order to see updated web site information. Web page information is often stored on the local disk to improve performance when users return to a page they have already visited. Pages that are revisited will be recreated from the information in cache rather than reloading the page from the network. Unfortunately, this gave users the impression that the contents of the web site never changed since they were always viewing the cached version of the page and never seeing the updated one.

Many of these problems could have been anticipated and solved before the web site was introduced. Surveys could have been conducted to determine SQA archive use, web use, browser familiarity, and site content preferences. Training could have been provided on internet applications, browser configuration, and site use. Usability testing could have been performed to confirm the appropriateness of the site content, organization, and presentation. QACS assumed some risk in quickly creating and releasing its web site, and initial use of the site suffered for it.

OBSTACLES OVERCOME

Three basic problems needed to be solved. First, people needed to be convinced that they had a reason to visit the web site. Second, they needed to have web browsers installed.

Third, they needed to have their web browser configured correctly. These problems also needed to be solved without any additional web maintenance resources.

Two changes were made to give people a reason to visit the web site. One was to improve the content of the site so more people would want to use it. The development community was solicited for ideas for additional content. The other was to force the development community to use the site by no longer e-mailing or distributing printed copies of some documents.

To improve the content of the site, the following topics were added:

QA Partner - Guidelines, "How To..." information, and training schedules for automated testing.

Training - Class outlines, slides, and materials for QACS training classes.

Library - Software engineering books and articles available from QACS.

Quote of the Week - Wit and wisdom from software engineering's most respected writers and practitioners.

Customer Satisfaction - Survey results, metrics, and action items for measuring and improving customer satisfaction. This page was intended to be used by Customer Service, Sales, Technical Support, and project managers.

Top 20 Customer List - Listing of the company's 20 most important customers.

How to Create a Home Page - Instructions, templates, and tools to help departments and project teams create their own web pages.

Setup - Instructions for how to configure a web browser for Word, Excel, and PowerPoint files, and add the SQA web site to the browser "hotlist."

The biggest change was adding the setup instructions. Before, the only instructions for configuring browsers were in a newsletter. Now, every page featured a link to the setup instructions page and the main page included a notice to new users to go to the setup page and configure their browser. Even with this, the setup instructions tended to be ignored. Eventually, the main page was replaced with a separate page featuring a large stop sign and a warning that documents would not be accessible until the browser was correctly configured (Appendix C). A link could then be followed to either the setup instructions, or to the regular main page. Users could also change the path to the QACS web site saved in their "hotlist" and go straight to the main page without always having to see the warning page.

In additional to soliciting development team members for ideas for content for the site, they were also asked to supply their own content. Though only a few developers actually provided content for the site, team members felt the site was more inclusive since they were being asked to provide content.

Web site development tools were added to help other departments and development teams create their own web sites. More web sites would mean more content and more reasons to install a web browser and use the intranet. Development team members who might not want to take the time to install a web browser just to access the QACS web site might do it to access their team's web site or the web site of another development team. And, once they had the browser installed, they might then visit the QACS site.

Finally, QACS stopped distributing printed or e-mailed copies of many documents. The only way development team members could get the documents was by going to the web site. Whenever a request came in for a document, e-mail was sent telling how to get it off the web site. There was some initial resistance to this, but over 2 or 3 months many people did decide there was a reason to install a web browser and use the intranet.

All these changes helped to give development team members a reason to use the SQA web site, install web browsers, and configure them correctly.

USAGE INCREASED AND CONTENT IMPROVED

Three months after its initial introduction, executive management increased their support for the intranet and chartered the MIS team to create and maintain a dedicated web server and help other departments create their own web sites. At this time, the QACS web site was still the only internal web site for the company. It would be another 3 months before any departments other than MIS created a web site.

Moving to a server meant that a number of changes had to be made to the web site. First, all SQA documents needed to be moved to the ftp server and all HTML files moved to the web server. All HTML files needed to be modified during the move to use fully qualified paths to link to the SQA documents on the ftp server as relative paths would no longer work. Also, write-access to the HTML files and SQA documents was now more limited since only the QACS manager and one staff member had permission to write to the ftp and web server directories. Before moving to the server, the entire QACS department had write permission. All users throughout the company continued to have read-only permission.

Moving to the server also allowed some new features to be added to the web site, including:

Newsgroups - QACS created internal newsgroups for QA Partner and Development Process Issues and Ideas. Eventually, development teams would create another 20 newsgroups on technical topics.

Access Counter - The "hit counter" showed for the first time how many visitors the site was getting. After 3 months, the site was only receiving about 40 hits a month. After 6 months, the site was up to 200, and after 9 months it was up to 400.

Surveys - Web site surveys were added to find out what people liked and didn't like, what they found most useful, and what else they would like to see. The survey comments led to a number of user interface changes, including changing the background color from gold to white, rearranging the icons on the main page to eliminate some scrolling, and adding drop shadows to the icons to make the interface less plain.

Please see Appendices D and E for examples of the updated main page and a topic page.

The release of Microsoft Windows 95 also further eliminated some installation problems. Many development teams changed to this operating system, which comes bundled with a web browser application (Internet Explorer). New employees receiving new PCs would already have the web browser installed. This browser would also prompt for an application when it encountered an unknown file type, and not display a screen of garbage characters. (Survey results showed that, 9 months after introducing the web site, our Mosaic product had been replaced almost completely by either Netscape Navigator or Microsoft Internet Explorer.) Microsoft Internet Assistants for Word, Excel, and PowerPoint (which convert existing documents into web pages) made it easier for other departments and development teams to create web sites, which increased intranet use in general, including use of the SQA web site.

About a month after the web server was established, the MIS team created a top-level "Information Navigator" page from which users could follow links to other web sites on the intranet. Eventually, this page would include links to web sites for Technical Support and a dozen development teams in addition to the QACS web site. The MIS team further encouraged the development community to use internal web sites for document distribution by instituting a maximum size limit on e-mail enclosures. Though the initial impact of this was to encourage development teams to compress their e-mail enclosures, it no doubt eventually helped convince some development teams to create their own web sites.

Nine months after its initial introduction, the SQA web site was being used as it was intended.

BENEFITS REALIZED

Over the course of 9 months, QACS changed the way documents were distributed within the company. Though it is not known how much money was saved by using electronic distribution for all SQA documents, QACS saved $800.00 per month when it stopped printing and distributing copies of 1 monthly report. (It is also possible that these savings were lost due to increased local printing of the same documents.)

All satellite offices now used the QACS web site to access company standards and templates. The content of the standards and templates also improved. Many changes were made to standards and templates as comments were received from new document users. Changes made to documents were also instantly available on the web site, and were immediately announced on the "What's New" page of the web site. The "Standard Operating Procedures" topic page was also modified to show which standards had been newly modified, or were currently having changes reviewed and approved.

On those occasions when development team members called to ask for a specific standard or template, it was usually because a standard did not exist that they expected to find on the web site. This led to the creation of additional needed standards and templates. It also prompted some development teams to create their own standard operating procedures for processes specific to their development team. Procedures developed by other departments, such as Software Publishing, that were related to development team activities were also added (or linked) to the QACS web site as needed.

The QACS web site also led to improvements in the company's external web site. More development team members now visited the company's external home page and were able to provide marketing with corrections for wrong or out-of-date product information. This also led to the creation of a content review process to assure that development team members, particularly SQA engineers, reviewed all external home page content for correctness before marketing could publish the content on the World Wide Web.

CONCLUSIONS

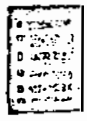
An internal SQA web site can improve your ability to distribute standards, templates, and tools within the company, improve the content of standards, and also encourage other departments and development teams to document their standard practices. Creating an internal web site can be done quickly, inexpensively, and without a lot of technical knowledge. However, you should not expect to see improvements immediately. You can reduce the amount of time necessary to establish your web site and increase the likelihood that your web site will be used successfully by planning your web site introduction and marketing as carefully as you plan your web site content.

Appendix A

Quality Assurance Central Services

Welcome to the Quality Assurance Central Services home page. From here, you can access templates, tools, procedures, and technical bulletins to help make software development and testing more efficient and effective. You can also subscribe to our monthly newsletter, *Quality Times*.

What's New



Templates and Examples

Tools

Technical Bulletins

Standard Operating Procedures

Newsletter Archive

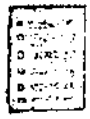
Gallery of Greatness

Going Places

Please send us your suggestions and comments.

Last updated 08/25/95.

Appendix B



Templates and Examples

The following templates and examples have been developed for company use to assure consistency and completeness for standard development and testing tasks and deliverables.

Templates

Test Plan

Test Cases

Test Summary

SQA Plan

SQA Summary

Software Development Plan

TC Release Notes

Beta Test Plan

Post Project Review Report

Project Definition

TMP Customer Service Notification

PTF Customer Service Notification

Examples

Test Plan

Trace Matrix

Test Results

Test Summary

Return to main menu.

[What's New Templates Tools Technical Bulletins Standards Newsletters Gallery Going Places]

Appendix C

You must have your browser configured to read *Word* files to access the information on the Quality Assurance Central Services Home Page. If you do not have your browser configured for this, or do not know if it is, please read these instructions.

If you have already configured your browser to access *Word* files, you can continue straight into the QACS Home Page. You will also want to update your hotlist or bookmark for the new Home Page location.

You can always set up your browser at a later time if you would like to check out the Home Page first. To find the setup instructions, just select the *Setup* link at the bottom of any topic page, or the *read this first* link from the main page.

If you have any questions, or need help configuring your browser, contact Tim Farley.

WALL DATA®

Quality Assurance Central Services

Welcome to the Quality Assurance Central Services Home Page. From here, you can access our Top 20 Customer list and monthly Customer Satisfaction Survey results. You can also find templates, tools, procedures, and technical bulletins to help make software development and testing more efficient and effective. If you are new here, read this first.

Be sure to enter the Contest of Greatness!

What's New

Quote of the Week

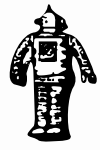
100%

Customer Satisfaction

Issues & Ideas News Group

Tools

TEMPLATES



QA PARTNER

TOP 20

Customer List

Training

Newsletter Archive

Library

QA PARTNER NEWS GROUP

Going Places

Gallery of Greatness

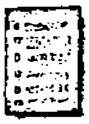
INFORMATION NAVIGATOR

Please take the QACS survey and let us know what you think about the home page.

Last updated 2/8/96. This Home Page was designed and implemented by Tim Farley for the Quality Assurance Central Services Group.

visitors since 4/16/96.

Appendix E



Templates and Examples

The following templates and examples have been developed for company use to assure consistency and completeness for standard development and testing tasks and deliverables.

Templates

Release Approval Form
: The Release Approval Form is appended to the *Using the Release Approval Form* SOP.

Localization Testing Checklist
: This was put together by Pat Hyland and is based on the checklist developed by Bob Sarver in August, 1995, and recently send out again by Sergei Kalfov.

SoftCop Testing Checklist
: This was developed in Bellingham (I don't know who did it) but it came to me through Pat Hyland.

Test Plan
: Based on the IEEE Standard for Test Documentation. Used to plan the testing effort, including what will and won't be tested, the approach to be used, the software release criteria, resource requirements, and schedule. **Test Plans are now required for release of software products.** See the Release Approval Form SOP for details.

Test Cases
: Based on the IEEE Standard for Test Documentation. Used to define the exact tests that will be performed. Describes each action to be performed, and the expected results. **Test Cases are now required for release of software products.** See the Release Approval Form SOP for details.

Test Summary
: Based on the IEEE Standard for Test Documentation. Used to describe the results of testing, and whether or not the requirements of the Test Plan were met during test execution.

SQA Plan
: Based on the IEEE Standard for Software Quality Assurance Plans. Used to plan SQA tasks and deliverables concerning reviews, metrics, release requirements, and testing.

SQA Summary
: Based on the IEEE Standard for Test Documentation. Used to describe the results of the SQA effort, and whether or not the requirements of the SQA Plan were met.

Software Development Plan
: Used by Software Developers to plan software requirements, design, and implementation activities. Includes risk assessment, resource assumptions, and schedule.

TC Release Notes
: Used by Software Developers to document for PQA Engineers the content of Test Candidate releases. Describes known problems, fixes, and other changes from the previous TC release.

Beta Test Plan
: Used to plan for beta testing, including what the test will include, who will be involved, when the

Examples

PTF Documentation Examples
: There are two short examples of PTF documentation available. The first uses none of the optional fields. The second uses some of them. Both have been shortened, and do not contain all the fixes that were included in that PTF. See the PTF Documentation SOP for more details.

SQA Plan
: Example SQA Plan for LAMBADA for the Internet. *(Under Development)*

Test Plan
: Describes the plan for testing the example LAMBADA product.

Trace Matrix
: An example of how to trace test cases to testable product requirements. Used to objectively track functional test coverage.

Test Cases
: Example Test Cases for LAMBADA for the Internet. *(Under Development)*

Test Results
: An example of how test results can be recorded.

Test Summary
: Describes the results of the LAMBADA testing effort. Shows how the summary is linked to the specific Test Plan used for the testing effort.

Return to main page.

[What's New | Quote | Customer Satisfaction | Standards | Templates | Tools | QAP | Training | Library | Newsletters | Gallery | Going Places | Setup]

EXPERIENCES WITH THE SEI RISK EVALUATION METHOD

Peter Hantos, Ph.D.

Xerox Corporation
701 South Aviation Boulevard, MS: ESAE-375
El Segundo CA 90245
Phone: (310) 333 - 9038
Internet: phantos@es.xerox.com

ABSTRACT

The SEI (Software Engineering Institute) approach is a disciplined and systematic way to confront risk in software development. The Risk Management Paradigm, built on the Software Risk Taxonomy as a foundation, consists of a continuous set of activities to identify, communicate, and resolve software technical risks. This presentation is an experience report, describing *process-related* experiences from five assessments carried out at Xerox. We will also summarize the benchmarking feedback we have acquired from other companies. While we will briefly describe the SRE (Software Risk Evaluation) method itself, presenting the associated principles and logistics, the emphasis will be on our own experiences and the customization opportunities. Two of the key, inherent attributes of the SRE method are confidentiality and non-attribution, consequently we will not reveal any facts about the actual, *project-related* results and conclusions of the assessments.

KEYWORDS

Risk Management, SEI

BIOGRAPHY

Peter Hantos is Principal Scientist of the Corporate Software Engineering Center. His tasks include the coaching and mentoring of software process improvement teams across Xerox on software technology related issues, identifying and sharing best practices, methods and tools. He is also involved in establishing process repositories/asset libraries, and the development of software training. One of his current responsibilities is the internalization and institutionalization of a systematic software risk management approach for the corporation. In his previous position at Xerox, Dr. Hantos managed a software engineering organization providing methodology, tools and infrastructure support for members of the Corporate Research and Technology Division located in El Segundo, California.

1. INTRODUCTION

Assessment and management of software risks are now recognized industry-wide as critical project management activities. Most of the Xerox products are architecturally embedded computer systems, and the software content of these products is rapidly growing. Consequently, success or failure in the software domain will directly determine the success of our products. The literature dealing with software risk addresses a range of subjects (such as patients being over-radiated, missiles being misfired or baggage shredded at the Denver Airport...). However, the primary issue for Xerox products is to avoid the cancellation of a project. This involves gaining control of the product "QCD" (Quality - Cost - Delivery) factors. Many Xerox products have a software content exceeding 10,000 Function Points. 10,000 Function Points are roughly equivalent to 1,280 KLOC (Thousand Line Of Code) of "C" source code. According to Software Productivity Research [1] the cancellation probability of projects of this size is at least 50%, and the average project delay is 100%. The recognition of these alarming trends provides ample motivation to implement systematic methods to identify and to confront risks.

2. WHAT IS RISK MANAGEMENT?

Risk management is a set of continuous activities to identify, plan, track and control software risks. It is conducted in the context of normal program management (Fig. 1).

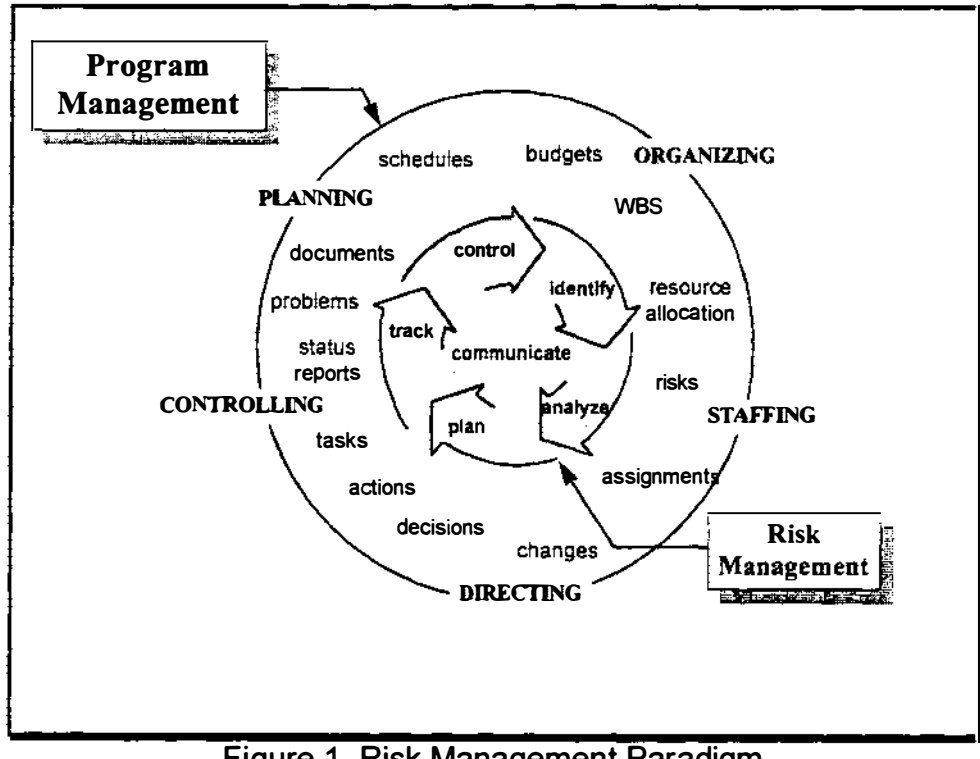


Figure 1. Risk Management Paradigm

[1] Software's Chronic Crisis, W.W. Gibbs, Scientific American, September 1994, pages 86 - 95.

Risk management activities, i.e. identifying and confronting risks should begin as soon as possible. By integrating risk assessment early in project management, one can avoid problems associated with calling for a risk assessment after a major project exposure has been identified. A practical question is to determine when the risk evaluations should actually take place. The spiral model of the software development process [2] requires the identification and resolution of risks prior to starting a new development phase. These are natural check-points, where we can decide if a formal assessment is justified or not. In most instances however, programs are monitored and managed in a waterfall fashion [3] at the senior management level. For these programs, SEI has identified the following high leverage points, shown on Fig. 2:

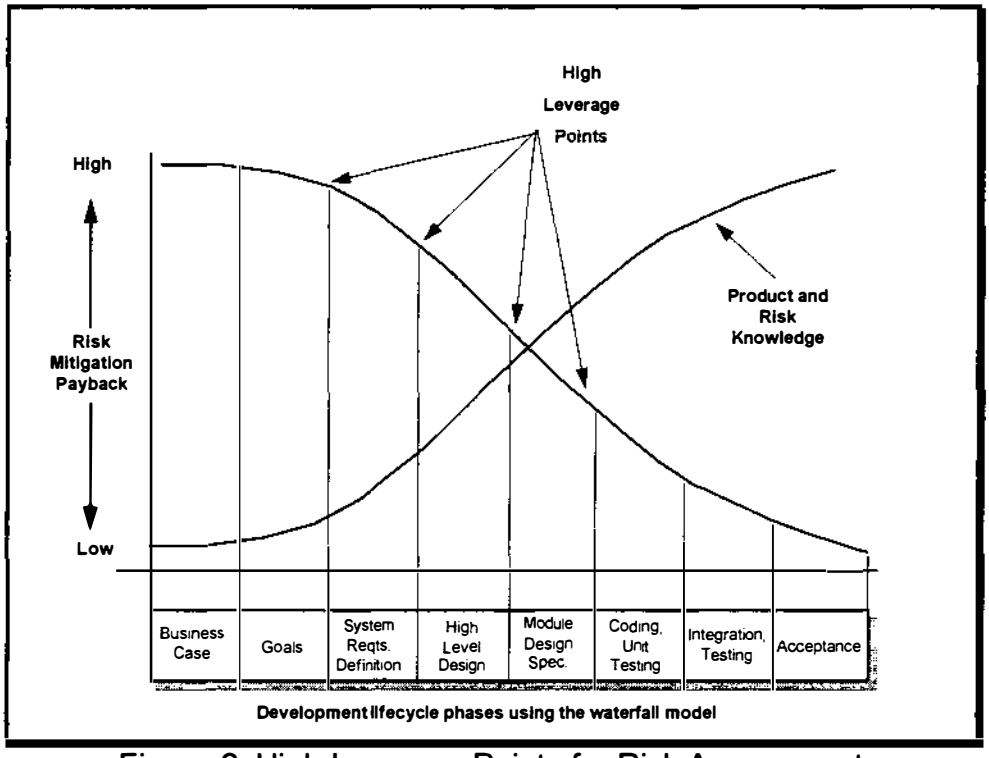


Figure 2. High Leverage Points for Risk Assessment

3. THE SRE APPROACH

Software Risk Evaluation (SRE) was developed at Carnegie Mellon University's Software Engineering Institute (SEI). The scope of SRE is identification, analysis and preliminary action planning for mitigation. The method is introduced in detail by Sisti and Joseph in [4]. The SEI software risk taxonomy, described in the Appendix, provides a consistent framework for risk management. The taxonomy is organized in three major *classes*, Product Engineering, Development Environment, and Program Constraints. On the next level of the taxonomy the *elements* of the classes are identified. Finally, at the

[2] A Spiral Model of Software Development and Enhancement, by B. W. Boehm, IEEE Computer, May 1988.
[3] Software Engineering Economics, by B. W. Boehm, Prentice Hall, Inc., 1981, pages 35 - 38.
[4] Software Risk Evaluation Method, Version 1.0, by F. J. Sisti and S. Joseph, CMU/SEI-94-TR-19, October 1994

bottom level *attributes* are assigned to all the risk elements. SEI also developed a tool called TBQ (Taxonomy-Based Questionnaire) to identify software risks in a program. A sample segment of the TBQ follows:

A. Product Engineering
2. Design & Implementation
d. Performance

Starter question:	***[22]***	***Are there any problems with performance?***
Possible Cues:		☛ ***Throughput***
		☛ ***Scheduling aschyncronous events***
		☛ ***Real-time responses***
		☛ ***Impact of hardware/software partitioning***
Starter question:	***[23]***	***Has a performance analysis/simulation been done?***
Follow-up questions:		***(YES) (23.a) What is your level of confidence in the results?***
		(YES) (23.b) Do you have a model to track performance?
		...

Structured peer-review interviews are carried out by an independent assessment team using the TBQ questionnaire. The identified risks are recorded, and at the end of each interview their impact and likelihood of occurrence are analyzed. Typically five or six interviews are conducted during one week, depending on the size of the project. The first day is for site orientation, program overview, and a training session for the assessment team. Over the next few days, interviews and analysis sessions follow alternately. Then, there are sessions on data consolidation, briefing preparation, and data confirmation. At the conclusion of the SRE, the consolidated risks are presented to project personnel and management. Later, the assessment team prepares a detailed report on the risks and suggested mitigation strategies.

It should be noted, that while the SRE is treated as an SEI product and they offer both training and complete assessments, their main goal is to empower customer organizations to customize and internalize the process, and eventually carry out assessments on a regular basis without SEI's help. The easiest way to receive information on training programs is via Internet (customer-relations@sei.cmu.edu) or the World-Wide Web (http://www.sei.cmu.edu).

4. BENCHMARK BEFORE GETTING STARTED

Let's assume that you have already attended the training and already arranged a briefing for your senior management by SEI. All the right buzzwords were said and the right buttons pushed, still you have to go through a tough justification process. Again, nobody will openly resist the idea of risk evaluation, but everybody will have concerns about cost. The overall cost will be composed of the cost of sending people for off-site training, retaining consultants for the assessment team, and the implicit cost component caused by the fact that the interviewed people and the internal members of the assessment team will be away from their regular activities for a given period.

Another valid issue is the scalability of the method. "This is great, but our project is too small, we need more lightweight processes", or "This sounds like a good idea, but our project is much bigger, and this approach is too soft for us" are the typical concerns. The most effective way to break this barrier is to inquire about other companies' experiences.

Getting benchmark information is not, however, an easy task due to the nature of the issues surrounding risks. Also, SEI as an organization and all individuals on the assessment team are obligated to sign a confidentiality agreement. Usually, SEI is free to use assessment data and conclusions for statistical or analytical purposes only if the information is used without attribution.

All inquiries for benchmarking information have to be handled on a case-by-case basis. If you are considering engaging the SEI for a risk evaluation, we suggest asking the Manager of the Risk Program at SEI (Ron P. Higuera) for current references, and he might be able to identify companies who agreed to provide information directly. At Xerox we were able to conduct phone interviews with representatives of three such companies. This information was presented to the first pilot program's senior management. All three companies, as well as Xerox customized the process and carried out the SRE differently; and we feel that even in its "coded" format this is valuable information for people becoming acquainted with the method.

4.1 Company "A":

At Company "A" the person who provided the data was a Senior Program Manager, responsible for many critical programs. SEI was retained to execute two SREs. The first one was a comprehensive risk review of a program valued at roughly $ 20M, covering all aspects of the project, not only software development. The second one was an SRE on a software project valued at $ 3M, consisting of a 300 KLOC mission-critical real-time software system, developed for the United States Air Force. SEI was retained for the SRE only, and was not asked to present a detailed risk mitigation. Company "A" management was very pleased with the results, and the Senior Program Manager was impressed with the power of the SEI/SRE interviewing method. In his opinion, the main value of the assessments was the dramatic improvement in communication among

management and program personnel. The SRE report also played a key role in determining program priorities. They used these two assessments and the formal SEI training program to internalize the process. Some of their recommendations are as follows:

- The SRE should be carried out as early as possible, and continued rigorously with follow-ups.

- Even if the program is on track, or ahead of schedule, program personnel should be asked to explain their success. The responses reflect on organizational maturity and process capability.

- The SEI process should be adhered to during the assessment. Substantial customization should take place only after the results of the first assessment were analyzed.

- Gaining commitments from key people internally for the assessment team is difficult, but necessary. A solution is to make clear to senior managers that while today they are asked to provide a person for the assessment of another program, tomorrow they might need competent people for an assessment of their own program.

4.2 Company "B":

At Company "B" a so-called Team Risk Assessment variant of the SRE was used. In this case the team was defined as the development group *and* the customer. As a result, the risk management plan was developed together with the customer. The project size was approximately 100 KLOC, a Command and Control Real-time system, developed for the United States Navy. The following conclusions were presented to us:

- The peer-group interviewing approach proved to be an efficient way to discover problems. It seemed to be more effective than the traditional approach where individual risk managers are assigned to the program.

- The interviews revealed new problems, and the team was able to identify risks which had not surfaced earlier.

- The experience improved their relationship with the customer. The risk factors become immediately visible to the customer, and the collaboration on the mitigation strategies proved effective.

- SEI's role was strictly process technology transfer, and not the formal administration of the SRE. The company developed its own risk mitigation plan, but also provided information to SEI to "debug" the process.

4.3 Company "C":

The assessment team members attended the formal SEI training and tried to customize and streamline the process for their company.
SRE was executed on three projects:

- The team started with a pilot project of approximately 30 KLOC. In this project, SEI's only role was to help in the preparation.

- The second project was a major program involving the US Government, the US Air Force and various subcontractors. The system was very complex, consisting of ≈500 KLOC software, mixed programming languages and operating systems, Command & Control and Telecommunication applications with strict real-time requirements. SEI participated in a mentoring and monitoring capacity.

- The third project was similar to the second, introducing an added level of complexity by adopting Object Oriented Technology. SEI's role was limited to monitoring the process.

Company "C" representative shared the following details:

- They were successful in streamlining the process and carry out assessments with a smaller team.

- Invariably, the interviews yielded additional risk factors not considered before.

- They decided to call in domain experts only after they determined and prioritized the risks. This provided a cost-saving opportunity.

- They implemented a new practice to assign a formal Risk Review Board to every major project to monitor the status of the identified risks.

5. XEROX EXPERIENCES

As of today five formal software risk evaluations were completed at Xerox. They ranged from a mini-assessment to a major, SEI-led Software Risk Evaluation, including the development of mitigation strategies. In the next few paragraphs we will present our observations.

5.1. Software Risk Taxonomy and the TBQ

Apparently the role of the actual questions is much less than what our original perception was, based on the SEI training materials. The taxonomy proved to be

complete, the questions were customized on the fly. The interviews have their own dynamics; the questions stimulate the process.

The other role of the taxonomy is to devise an interview strategy before the session. This is necessary because usually there is not enough time to touch on all issues, so the assessment team has to make sure that the most important questions, relevant to the particular interview group, are asked first. It is a very useful practice to leave a 10 - 15 minute block of time for soliciting issues which were missed during the structured questioning.

5.2. Structure and the Timetable of the Assessment Week

Our initial perception was that the bulk of the work will be to administer the interviews. In reality, every interview was followed with a long evaluation session, where the length and the thoroughness was fully justified. It is important to schedule the final consolidation session for an afternoon, preferably in an "open-ended" fashion. The assessment team has to have enough time to process and prioritize the results, and depending on the issues and complexity, these activities take much longer than is usually expected. Since the briefing session has to be "hard" scheduled, due to the fact that all interview participants and managers have to be invited, it is not prudent to jeopardize it by not leaving enough time for preparation. The time we allocated on the morning of the last day was barely enough to review the presentation overhead slides for the briefing, and provide immediate, preliminary feedback to program management.

5.3. Use of computers

A strength of the SEI process is that issues are recorded on a flip-chart and on a computer *at the same time*. This is different from the everyday practice where notes are taken on a white-board or flip-chart, and transcribed later. The participants can view and discuss the issues on flip-charts, and at the end of the interview session the issues are available in electronic form as well. Worksheets are generated instantly to facilitate the evaluation and prioritization. This activity requires the availability of backup computers and printers, since technical problems are unavoidable.

5.4. Team building

The official SEI material refers to the first meeting of the assessment team members as training, and in fact this meeting does have a training function even if all the team members attended earlier formal training as well. But our feeling was that it played the role of team-building, which was perhaps even more important than the informative function. We also scheduled a team dinner at the end of the first day which helped to build a good working relationship among participants. Since the assessment team members spend time arguing about the issues, a constructive and friendly environment is vital for efficient work. One minor caution about scheduling the team

dinner: Always do it on the first night. Since time management becomes very difficult after the interview sessions started due to delays and discussions during the evaluation phase, it may not be possible to schedule a nice and relaxed dinner on another evening.

5.5. Interview Dynamics and Interviewing Style

The leader who is asking questions must excerpt control over the meeting. The goal is to uncover issues, not resolve them on the spot. The immediate objective is to get a concise and clear definition of the problem, and document it on the flip-chart. Unfortunately many participants are tempted to vent and go on tangents in response to the assessment team's clarifying questions. While it is important to provide detail if the assessment team does not seem to grasp the issues, still the objective is that the participants have to feel comfortable with the actual wording on the flip-chart, which has to be as concise as possible.

Another delicate issue is that during the SRE we have dual goals, and it is not very easy to satisfy them simultaneously. On one hand we are clearly trying to discover future risks, problems affecting the achievement of the product QCD's. This objective would suggest that issues which can be classified as "water under the bridge", i.e. problems relevant to the past, but being currently worked at, or in fact fixed, should not be documented on the flip-chart. On the other hand the SRE is a learning experience for the division and, in fact, the company, so issues which have broad value should be recorded as well with the appropriate qualification, so the team does not spend time on finding mitigation strategies for resolved issues. The dual nature of the goals has to be clearly communicated, so participants will not spend too much time talking about past, already resolved, problems.

Beside flip-chart and computer, a third kind of recording is taking place as well. These notes, called context recording, augment the brief and concise problem statements on the flip-chart, so at the time of the team discussion and evaluation the context can be read back for clarification purposes. Context recording was a new experience for us, again, different from our everyday meeting practice. It proved to be very useful, because many times even the person who originally raised the issue could not remember what a particular flip-chart entry meant at the end of the session.

5.6. Team size

Management's original perception at the first SRE was that the planned assessment team size (8 people) was excessive, and the team should consist of object oriented software domain experts only. Nevertheless our own experience showed that both the size and the structure of the team was right. The administrative functions, recording, computer processing and prioritization are equally important to the domain specific questioning. It was also beneficial to be able to create four two-person subteams and pair up Xerox-SEI personnel to work parallel on impact area issues.

The size of the interview teams turned out to be not as critical as we assumed, because the interview dynamics determined if the allocated interview time was really enough or not. Our conclusion is that the key is to sense the dynamics of the group and exercise more control if needed, and not limit the number of people in the session.

5.7. Definition of Impact Areas

The granularity of the collected information corresponds to the granularity of the risk taxonomy table. For sake of more efficient processing, the identified risk factors are further classified and consolidated into user-defined, project-specific impact areas. First, we solicited impact area definitions from the "customers" of the SRE, e.g., program management. These impact areas were used for the preliminary structuring of the issues. After the completion of the interviews, we were able to discover further relationships between the issues, leading to either new impact areas, or the consolidation of existing ones. The structuring and presentation of the issues were acknowledged as the most valuable features of the assessment, because program management gained additional insight into areas where they believed risk factors were already known. The somewhat weak point of the methodology is how to capture and convey concisely the fact that a given problem belongs to more than one impact area.

5.8. Calibrating Expectations

It must be clearly understood from the beginning, that the SRE is uncovering and recording issues brought up by the project personnel. This means that if nobody brings up a particular risk, than the SRE team will not be able to discover it. Also, the SRE team has to make painstaking efforts not to distort the input from the interviewed personnel. We think that this aspect of the SRE is definitely different from other program review processes. The assessment interviews represent a morale boost to the organization, and they might have a certain "therapeutic" effect. Participants interpret it as a serious commitment from senior management to find and fix problems. To derive this benefit, the confidentiality and non-attribution goals must to be communicated and reinforced in every interview session.

6. SUMMARY

In summary, we can confidently conclude that the process knowledge transfer from SEI was successfully completed, and now we are able to carry out training on risk assessment, as well as organize and lead SREs. Feedback from the completed SREs shows that both management and project personnel were satisfied with the process and the results. After piloting the activity on three major programs we were ready for full customization and internalization. The subject of software risk management becomes part of the official Engineering Excellence Training program, and program

managers across the corporation are encouraged to use the process. Our experience shows that the process is easily scaleable by adjusting the number of interview groups and the size of the assessment team, and the taxonomy is complete enough for all the software projects we have to deal with. Also, we are working on a specially customized version of the method by defining a mandated major phase gate review in the product delivery process.

- ★ -

APPENDIX

COMPLETE SOFTWARE RISK TAXONOMY

A. PRODUCT ENGINEERING

1. Requirements
   - a. Stability
   - b. Completeness
   - c. Clarity
   - d. Validity
   - e. Feasibility
   - f. Precendented
   - g. Scale

2. Design and Implementation
   - a. Functionality
   - b. Difficulty
   - c. Interfaces
   - d. Performance
   - e. Testability
   - f. Hardware Constraints
   - g. Non-Development Software

3. Code and Unit Test
   - a. Feasibility
   - b. Testing
   - c. Coding / Implementation

4. Integration and Test
   - a. Environment
   - b. Product
   - c. System

5. Engineering Specialties
   - a. Maintainability
   - b. Reliability
   - c. Safety
   - d. Security
   - e. Human Factors

B. DEVELOPMENT ENVIRONMENT

1. Development Process
   - a. Formality
   - b. Suitability
   - c. Process Control
   - d. Familiarity
   - e. Product Control

2. Development System
   - a. Capacity
   - b. Suitability
   - c. Usability
   - d. Familiarity
   - e. Reliability
   - f. System Support
   - g. Deliverability

3. Management Process
   - a. Planning
   - b. Project Organization
   - c. Management Experience
   - d. Program Interfaces

4. Management Methods
   - a. Monitoring
   - b. Personnel Management
   - c. Quality Assurance
   - d. Configuration Management

5. Work Environment
   - a. Quality Attitude
   - b. Cooperation
   - c. Communication
   - d. Morale

C. PROGRAM CONSTRAINTS

1. Resources
   - a. Staff
   - b. Budget
   - c. Schedule
   - d. Facilities

2. Contract
   - a. Type of contract
   - b. Restrictions
   - c. Dependencies

3. Program Interfaces
   - a. Customer
   - b. Associate Contractors
   - c. Subcontractors
   - d. Prime Contractor
   - e. Corporate Management
   - f. Vendors
   - g. Politics

Utilizing the Capability Maturity Model for Software to Achieve and Retain ISO 9001 Certification

Allen Sampson
Tektronix, Inc.
Measurement Business Division
P.O. Box 500, M/S 39–732
Beaverton, Oregon 97077

(503) 627–2883
e–mail: Allen.W.Sampson@TEK.COM

Abstract

The Capability Maturity Model for Software (CMM), developed by the Software Engineering Institute, and the ISO 9001 standard, developed by the International Standards Organization, are separate methods concerned with quality and process management. In the Measurement Business Division of Tektronix, the CMM and ISO 9001 software process and quality initiatives were combined to achieve and retain ISO 9001 certification. An ISO 9001 framework is presented that utilizes the CMM to facilitate and drive software process improvement. The experience of successfully developing a software quality system and achieving notable software process and product quality improvements at Tektronix is described along with an examination of future improvement areas.

Key Words

ISO 9001, software quality system, Capability Maturity Model for Software, key process areas, software process improvement

Biographical Sketch

Allen Sampson is a software quality manager with Tektronix' Measurement Business Division. In 15 years at Tektronix, he has held positions in software test, software quality assurance, software quality management, and program management. He is the recipient of the Tektronix 1996 Software Quality Leadership Award and is certified in software process self–assessments and Tektronix quality system audits. He holds a BS and MS in Computer Science from the University of Utah. His interests include software engineering, software process improvement, and software measurement.

1 INTRODUCTION

The Capability Maturity Model for Software (CMM), developed by the Software Engineering Institute, and the ISO 9001 standard, developed by the International Standards Organization, are separate methods that share a common concern with quality and process management. A product instrumentation group within the Measurement Business Division of Tektronix combined the CMM and ISO 9001 software process and quality initiatives to achieve and retain ISO 9001 certification.

ISO 9001 and the CMM were used to develop a software quality system that specifies the organizational structure, policies, procedures, and processes used by the organization. The development of the software quality system followed a simple approach using the following steps:

- understand ISO 9001 and the CMM
- define the software quality system
- implement the software quality system
- use and improve the software quality system

The experience of developing the software quality system is described using the same intuitive approach.

2 OVERVIEW

It is helpful to look at key aspects of ISO 9001, the CMM, and the Tektronix product instrumentation group to understand the development of the software quality system.

2.1 ISO 9001

The ISO 9001 standard prescribes generic quality system requirements that are tailored to the needs of an organization. The basic content of the ISO 9001 standard is captured in the 20 clauses shown in Table 1. A full description can be found in the ISO 9001 standard [4] and guidelines such as ISO 9000–3 [1,5]. The 20 clauses were used to develop the framework for the software quality system.

1. Management responsibility	11. Control of inspection, measuring and test equipment
2. Quality system	12. Inspection and test status
3. Contract review	13. Control of nonconforming product
4. Design control	14. Corrective and preventive action
5. Document and data control	15. Handling, storage, packaging, preservation and delivery
6. Purchasing	16. Control of quality records
7. Control of customer–supplied product	17. Internal quality audits
8. Product identification and traceability	18. Training
9. Process control	19. Servicing
10. Inspection and testing	20. Statistical techniques

Table 1. Clauses of the ISO 9001 Standard

2.2 Capability Maturity Model for Software

The CMM describes a series of five software process maturity levels ranging from ad hoc and chaotic processes to mature, disciplined software processes. Each maturity level has several key process areas (KPA) that indicate the areas an organization should focus on to improve its software process. A full description of the CMM is given by Paulk [6]. Figure 1 shows the CMM maturity levels and key process areas. The key process areas were used to develop much of the content (processes and procedures) of the software quality system.

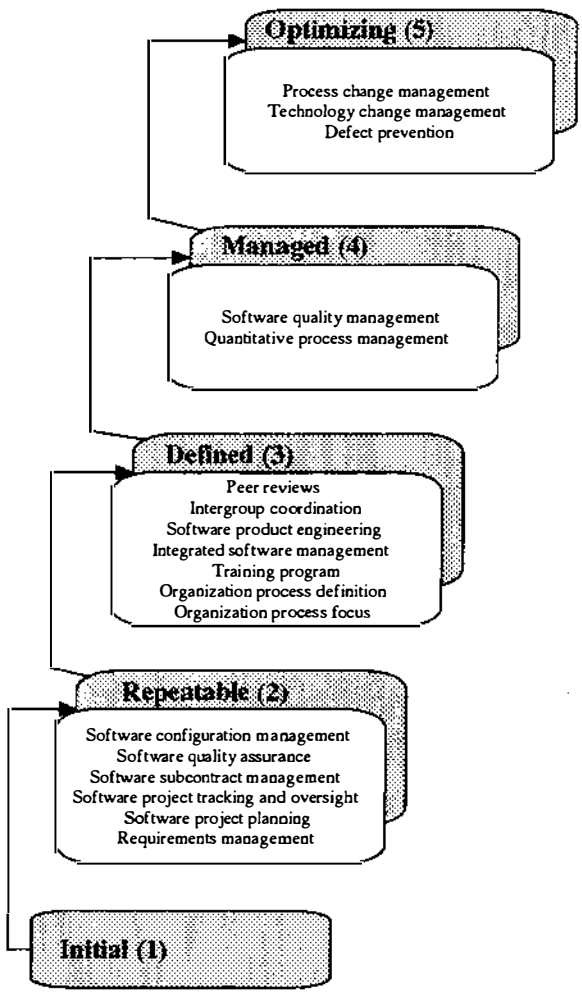


Figure 1. The CMM Key Process Areas by Maturity Level

2.3 Instrumentation Group

Some key aspects and general characteristics of the Tektronix instrumentation group include:

- **Organizational Structure**: A matrix structure organizes the group into functional areas along one axis and projects along the other axis. Functional managers are over each of the areas and program managers are over each of the projects. The functional areas include software engineering, hardware (electrical) engineering, marketing, and manufacturing among others. Figure 2 shows the matrix organizational structure. The software engineering functional area consists of software design engineers and a separate software quality group responsible for software processes and software quality assurance activities.

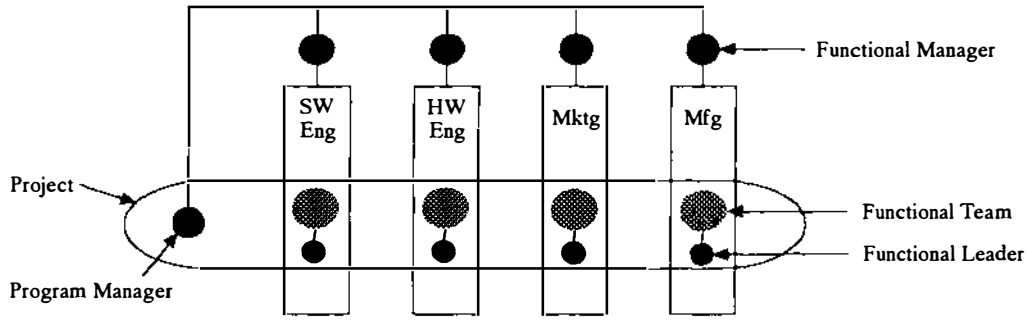


Figure 2. The Matrix Organizational Structure

- **Projects**: Projects are product development efforts that are staffed by teams from each of the functional areas. Each team is led by a functional leader. The functional leaders and program manager form a project core team that meets periodically to manage and coordinate the project.
- **Software Teams**: Software teams including software design engineers and software quality assurance engineers range from 2 persons to more than 20 persons.
- **Instrumentation:** The products developed are real–time embedded systems implemented in SmallTalk, C++, C, and assembly. Product code size is in excess of 500,000 lines of code.
- **Management Directives:** The development of the software quality system began with a corporate directive to achieve ISO 9001 certification and a strong software functional management commitment to improve the software process.

The organizational structure and management directives facilitated the development of an organization process focus and an organization process definition, two key process areas of CMM level 3.

3 DEFINING THE SOFTWARE QUALITY SYSTEM

The structure, content, and operation of the software quality system are explored next by examining the conceptual software development model, the software quality system framework, and the software quality system content including policies, procedures, and processes.

3.1 Software Development Model

The conceptual operation of the software quality system is illustrated by the software development model shown in Figure 4. The software policies, procedures, and processes are defined in the software quality system (top box in Figure 4). Each software project (the dashed box in Figure 4) creates a Software Project Plan and Software Quality Plan that consciously tailor the procedures and processes of the software quality system to the needs and constraints of the project. The Software Project Plan and Software Quality Plan are the key documents that bind the project to the software quality system.

Once created, the plans are used to run and track the project software development activities. As work progresses on a project, corrective actions are taken when deviations from the plans occur, the plans are updated, and process improvements are identified.

Process improvement activities are funnelled to the software quality group for disposition. Minor changes to the software quality system are handled directly by the software quality group. Major changes are handled by ad hoc continuous improvement projects, formed from interested members of the software group and usually chaired by a member of the software quality group. Emphasis is placed on resolving the root cause of software quality system deficiencies and problems.

The software development model is very similar to the model used by Raytheon [3].

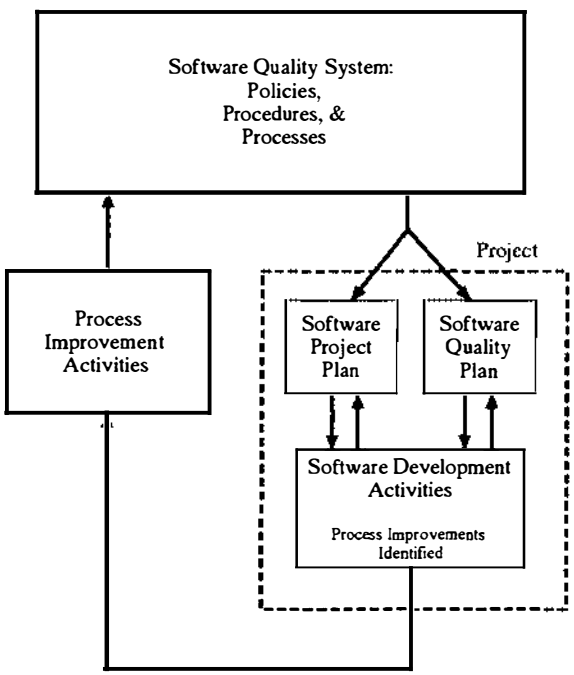


Figure 4. The Software Development Model

3.2 Software Quality System Framework

The software quality system is organized into the three tiers. The tiers are defined below and an example of the relationship between the tiers is shown in Figure 3.

1. **Corporate Quality Manual**: A quality manual translates ISO 9001 to meet both business needs and certificate requirements. Each of the 20 clauses of the ISO 9001 standard has a corresponding section in the quality manual.
2. **Corporate Policies**: An extended set of corporate policies provide further definition of the processes contained in the quality manual. The policies specify responsibilities common to all functional areas.
3. **Functional Area Policies, Procedures, and Processes**: An extended set of functional area policies, procedures, and processes describe responsibilities specific to each functional area.

The basic quality system is established by the tier one quality manual and then successively extended by the policies and procedures of tiers two and three. The common element in each of the tiers is support of the ISO 9001 clauses.

For example, the design control clause of ISO 9001 requires design and development planning. The planning requirement is expressed in the three tiers as follows:

Tier 1: All design and development activity shall be planned.

Tier 2: All functional areas shall use a common set of milestones and milestone requirements for planning.

Tier 3: The software functional area shall also use software specific milestones and milestone requirements for planning.

The ISO 9001 clauses form the software quality system framework for implementing the CMM software processes.

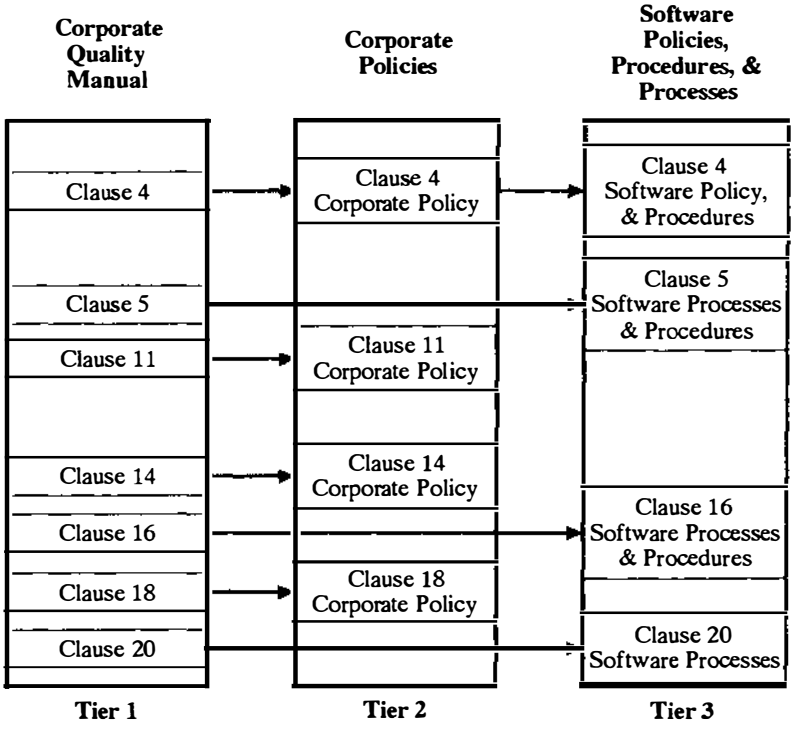


Figure 3. The Three Tiers of the Software Quality System

3.3 Software Quality System Content

The software policies, procedures, and processes are found in tier three of the software quality system. Table 2 lists the software processes along with supported ISO 9001 clauses and the related CMM key process areas. The table represents our road map for satisfying ISO 9001 requirements and improving our software processes.

The table maps the primary relationships between the software quality system and the key process areas based on our interpretation of ISO 9001. Paulk [6] gives a rigorous contrasting of ISO 9001 and the CMM for comparison.

Eighteen of the twenty two policy and procedure documents are in use today. The four remaining procedures are under development and the entire software quality system is continuously maintained and improved.

ISO 9001 Clause	Tier 3 Software Policies, Procedures, & Processes	CMM Key Process Areas
4	Software Development Policy	Intergroup Coordination (3) Software Product Engineering (3) Organization Process Definition (3) Organization Process Focus (3) Software Quality Assurance (2)
4	Procedure for Software Project Plans Procedure for Software Quality Plans Procedure for Software Project Tracking and Oversight	Software Quality Management (4) Integrated Software Management (3) Software Project Planning (2) Software Project Tracking and Oversight (2)
4	Procedure for Software Requirements	Requirements Management (2)
4	Procedure for Software Configuration Management	Software Configuration Management(2)
4	Procedure for System Requirements Procedure for System Architecture Procedure for User Interfaces Procedure for Hardware/Software Interfaces Procedure for Software Designs Software Coding Guide Procedure for Software Testing Procedure Software Archiving Procedure for Software Project Notebooks	Organization Process Definition (3) Software Product Engineering (3)
5	Procedure for Document Control	Software Configuration Management(2)
14	Procedure for Software Corrective Action Procedure for Software Change Control Procedure for Post Project Reviews	Organization Process Definition (3)
16	Procedure for Software Quality Records	Software Product Engineering (3)
18	Procedure for Software Training	Training Program (3)
20	Procedure for Software Metrics	Software Process Management (4)

Table 2. Tier 3 Software Policies, Procedures, and Processes

Notable aspects of the software quality system include software process tailoring and software procedure and process definition.

Software Process Tailoring

The Software Development Policy provides milestones, milestone checklists, and basic rules for tailoring the milestones and checklists to projects. Milestones mark the completion of software lifecycle phases such as requirements, design, coding, integration, and test. The milestone definitions are tailored for use in waterfall, incremental delivery, and spiral lifecycle models among others. Checklists specify the inputs, activities, and outputs for each phase and are also customized to the selected software lifecycle model.

Each software project uses the tailoring rules to create a Software Project Plan and Software Quality Plan suited for the project. Mandatory sections in both plans specify the tailored process:

- **Model Description**: A description of the tailored process model is given for the project, explaining why the model was chosen and how it will be used.
- **Milestones**: The milestones used to implement the tailored process model are listed.
- **Milestone Checklists**: The planned inputs, activities, and outputs are given for the milestone ending each phase.
- **Policy Exceptions**: Exceptions to milestone and checklist usage are given along with rationale explaining the deviations.

Signatures by the software functional manager, program manager, and the project functional leaders signify approval of the tailored plans.

Software Procedure and Process Definition

A common format and paradigm are used for defining the software procedures and processes. The format consists of a document main body and supporting appendices for process descriptions, guidelines, and checklists.

- **Main Body:** A terse one or two page main body links the document to the software quality system, sites relevant compliance procedures, and lists responsibility and control.
- **Process Description**: A process diagram and written process description define the software process for the engineering activity. The process description is written in general terms and usually contains tailoring provisions.
- **Guideline:** A guideline defines the recommended sections and content for documents such as a software requirements specifications and software design specifications. On–line templates containing recommended sections and informational text are copied for use on projects. The informational text provides tailoring and content guidance during document creation and is easily turned off or removed after the document is complete.
- **Checklists**: Checklists are maintained that capture group wisdom about historically important work product attributes. The checklists are used to guide reviewers toward defects.

During the development of the software quality system, emphasis was placed on achieving ISO 9001 compliance and establishing best practice software processes, while keeping the costs of using the system to a minimum.

4 IMPLEMENTING THE SOFTWARE QUALITY SYSTEM

The implementation of the software quality system was handled in several phases. First, the climate for organizational change was assessed and activities were identified to facilitate the planned changes. Next, a base software quality system was planned and established to achieve ISO 9001 certification. After certification, continuous software improvement efforts were used to extend the software quality system and achieve software process improvements.

4.1 Climate Assessment

Before implementation activities were started on the software quality system, two important steps were taken. A software quality group was established and post project reviews were held.

Software Quality Group

A software quality group was formed from an existing test group and given the charter to develop software processes and perform standard software quality assurance activities. The move initiated the software process improvement effort and set the expectation for the development of a software quality system.

Post Project Reviews

Post project reviews were held on three previous projects to determine common organizational problem areas that might be candidates for early software process improvement efforts. The reviews were also held to identify software process champions and early adopters (people willing and able to define, implement, and use software process improvements). The process improvement areas and champions were the key components to beginning the software process improvement effort. Successes from the early process improvements fueled the momentum of process improvement activities.

4.2 Base Software Quality System

The base software quality system was defined to have the minimum procedures and processes necessary to achieve ISO 9001 certification. The system included the software development policy, ISO 9001 related procedures, and a selected set of CMM related procedures.

Plan

The software policies, procedures, and processes needed for the entire software quality system were identified first. Then the minimum set was selected to achieve the base software quality system. The selection included:

- **Software Development Policy**: The full policy was defined.
- **ISO 9001 Related Procedures**: All ISO 9001 related procedures were selected including procedures for document control, quality records, corrective action, and change control.
- **CMM Related Procedures**: The CMM related selection included procedures for software project plans, software quality plans, software requirements, peer reviews, and code guides.

Software Development Policy

The software development policy was written by the software quality group with input from management and software functional leaders. Existing policies, in use more than 15 years, served as the primary source for creating the software development policy. Changes were made to accommodate the new matrix organizational structure and satisfy ISO 9001 requirements.

ISO 9001 Related Procedures

The ISO 9001 related procedures were also developed by the software quality group. Emphasis was placed on satisfying the basic requirements of ISO 9001 without placing undue burdens on software teams.

For example, project controlled document masters are maintained on line with a single controlled copy maintained in a project notebook. Uncontrolled copies may be retrieved by the project team from the on line master at any time. It is the responsibility of the holder of an uncontrolled copy to verify the version number against the master document list before use.

CMM Related Procedures

The CMM related procedures were developed by continuous improvement project (CIP) teams. The CIP teams were initially formed from the champions and early adopters identified in the post project reviews. As procedures were developed and successfully deployed, other members of the software functional area as well as other functional areas were encouraged to join the CIP teams. The use of CIP teams allowed a large percentage of the organization to make direct contributions to the software quality system.

The CMM related procedures not selected for development in the base system were entered as defects against the system (since the procedures were referenced in the software policy). Software project teams were instructed to continue the current practice in process areas with missing procedures.

Deployment

As the software development policy and each of the procedures were developed, training sessions were held and the procedures were deployed for use. The base software quality system was piloted on a single project before being widely deployed on other projects. Nine months were required to define, implement, and deploy the software quality system and receive ISO 9001 certification.

4.3 Extended Software Quality System

The activities surrounding the extension of the software quality system included attending software process self–assessment training, developing the remaining CMM related procedures, and performing continuous improvement activities.

Self–Assessment Training

After the development and deployment of the base software quality system, training sessions were held on accelerating change[2] and software engineering process improvement self –assessments[7]. The accelerating change seminar provided a practical guide for understanding and working with the human and organizational processes of change. The self–assessment training provided a useful understanding of the CMM self–assessment process and gave new insights into the organization. Both training

sessions and the subsequent self–assessment contributed to software process knowledge and increased the momentum of software process improvement activities.

Extended Software Quality System

The remaining CMM related procedures were developed and deployed using CIP teams similar to the development of the base software quality system. Software procedures were selected for development based on the expected pay back to the organization and were targeted for use on specific projects. Many times the champions and early adopters that formed the CIP teams were also the first users of the procedures on projects. Linking process improvement efforts to product development projects was a significant factor in maintaining process improvement momentum.

Continuous Improvement

A change control procedure and online defect tracking tool were established to handle problems and defects encountered by the software functional team in the daily use of the software quality system. As the quality system is used and defects are encountered, entries are made using the defect tracking tool. A change control board (consisting of software functional leaders and management) meets periodically to determine the disposition of the defects. Emphasis is placed on eliminating the root cause of the defects.

Post project reviews continue to be used for identifying software process improvement areas. The reviews are held at the end of a project by the software functional team with participation from other functional teams. The reviews initially used an open format targeted at expected problem areas, but have slowly changed to focus on the operation of the software development policy and procedures.

5 USING THE SOFTWARE QUALITY SYSTEM

5.1 Results

The software quality system was successfully implemented and deployed and is now under continuous improvement. Highlights of the development effort include:

- **ISO 9001 Certification**: The instrumentation group achieved ISO 9001 certification in December 1993 and has successfully retained that certification.
- **CMM Self–Assessment**: A software process self–assessment was performed in September 1994. Projects in the instrumentation group were assessed at CMM level 2 with significant CMM level 3 behaviors. Subsequent use of the CMM questionnaire on a yearly basis indicates continuous improvement by projects in the organization towards CMM level 3.
- **Organization Acceptance**: The loss of non–adopters (people leaving because they were unwilling to use the software quality system) was limited to less than ten percent of the organization. This is accredited to actively soliciting input on development of the quality system and instilling ownership of the quality system within the organization.
- **Projects**: Six projects have used the software quality system during the two and a half years since it was deployed. The projects have yielded products with high customer acceptance and numerous industry awards.

- **Software Quality Metrics**: Key software metrics indicative of the product and process quality include:

Defect Rate:	0.43 defects per normalized KLOC
Code Reuse:	Exceeding 80 percent
Cost Accuracy:	Within 12 percent of initial estimates on last 2 projects
Schedule Accuracy:	Within 2 percent of initial estimates on last 4 projects

5.2 Future Improvements

Areas for future improvements in the software quality system include:

- **Software Metrics Program**: The natural progression in the continuous improvement of the software quality system is the development of an organizational software metrics program. Facilitating the collection analysis, and retention of metrics can greatly reduce the overhead of manual efforts.
- **System Engineering Emphasis**: An experienced weakness of the CMM is in the area of systems engineering. More emphasis needs to be placed on system requirements definition and system architecture specification by examining technical resources outside of the CMM.
- **University Affiliations**: The establishment of university affiliations in process areas requiring improvements is one means of facilitating process improvement. The organization benefits by gaining access to world class technical expertise and the university benefits by gaining research access to industry.

REFERENCES

1 Beaumont, L.R., ISO 9001, The Standard Interpretation, Middletown, NJ: ISO Easy, 1995.

2 Fiman, B., "Accelerating Change", Implementation Management Associates, Inc., January, 1994.

3 Haley, T., Ireland B., Wojtaszek, E., Nash D., Dion, R. "Raytheon Electronic Systems Experience in Software Process Improvement", CMU/SEI–95–TR–017, November 1995.

4 "International Standard ISO 9001", ISO 9001:1994(E), July 1994.

5 "International Standard ISO 9000–3", ISO 9000–3:1991(E), May 1993.

6 Paulk, M.C. "A Comparison of ISO 9001 and the Capability Maturity Model for Software", CMU/SEI–94–TR–12, July 1994.

7 Rifkin, S., "Software Engineering Process Improvement Self–Assessment Training", Master Systems Inc., ver 2.3; CMM v1.1 Q1.1.0, 1993.

Script Channeller Methodology for Developing an API Test Application

A Software Evaluation of Effective Practices in Testing Techniques

by
Ashley Khong Eng Wee

P.O. Box 6423, Bellevue, WA 98008-0423. U.S.A.
(206) 562-7179
75541.3576@compuserve.com

Foreword

In the past 2 years of my involvement in testing the correctness of the SDK (Software Development Kit). I have devised a manner of testing software that will increase the quality of the Application Programming Interfaces (API) that are involved. This manner of testing will incorporate Script Channeling techniques into one tool that will be extendible and scaleable when required. Using this method, the time it takes to introduce new test cases and regress previous test cases will be dramatically reduced resulting in higher quality and reliability of the software component due to the completeness of the testing process.

During my past development and testing work done on the SDK, it became apparent that the parameters that are passed through the APIs are finite in nature. There are finite number of data types but infinite number of values of the data types. So by dividing the valid and invalid values for the data types that work for the APIs and storing them into a Parameter Database, the result will be an extensive set of test cases that exercises the APIs to its limits. By introducing scripts, parameters, execution order and execution multiples in which APIs are processed are determined by the content of the scripts.

Since the shipment of the SDK, there have been far fewer customers and support engineering groups reporting problems than most other products. The skeleton of the methodology has provided a framework to build on for extending other features of the methodology such as a definition language and source code generation.

Ashley Khong Eng Wee

Abstract

The Script Channeller methodology is to treat all APIs as a medium of passing and returning values no matter what the values are with respect to other APIs that depend on it. Once a call is made, the API must respond with the parameter passed or return values through the standard parameter values and return values of a function. Both the reaction and response of the API must be reasonable. In this paper, the response is defined as the feedback delivered from the API after it is called; that is, the return values and parameter values obtained from the API. The reaction is defined as the manner in which the API reacts to the call and the parameters passed to the APIs; that is API does not cause general protection faults and memory leaks during or after the call.

The API is treated as the smallest possible micro object in which values are passed through this medium and channeled through the API. It must respond and react in a reasonable manner which will not cause faults at a Macro Level where it is used. However, not all Micro Level testing can be achieved at this level since other Micro Level object's parameter may depend on outputs of other Micro Level parameters. With this in mind, scripts can be written to handle the Micro Level objects. Scripts can be just one micro object or a few micro objects to perform a task. A Script Manager will be used to handle several scripts and coordinate the functionality performed by the scripts. Several Script Managers can be written to execute the scripts in different ways. A Script Core is required to initialized Script Manager settings before use. A Parameter Database will contain sets of ideal, boundary, failure, and stress values. The sets of values will be alpha, numeric, ranges, bitmaps, objects, pointers, handles, and NULL which will range in length, size, and limits of the content. The parameters may be stored in any form of database such as source code structures, text file, SQL, or any database engine.

Scope of this paper

The following shows the component layering and structure of the Script Channeller. Figure 1 on page 7 shows you the layout of the methodology in a hierarchy format. The diagram shows how the layers are related so that one level may manage the other level during execution. Each level may manage the other level based on the reaction and response of the lower level.

- Micro objects
  - APIs
- Macro objects
  - Script
  - Script manager
  - Script core
- Parameter database
- Share code
  - Error handling
  - Initialization file
- Reporting
  - Automated pass/fail reporting
  - Source code generator
  - Problem location identifier
  - Process logging
  - Display window
- Cycles
  - Micro cycles
  - Script cycles
  - Script manager cycles

Audience

Although this paper describes and explains the Script Channeling methodology as a fully developed application, the methodology may be applied in parts. That is, depending on the development phase and development schedules, parts of the product or parts of this methodology can be used to fit within the product datelines. This paper will be appropriate for engineering managers, software development engineers, software test engineers who are concerned about the scaleability of the current test case of their APIs.

Engineering managers

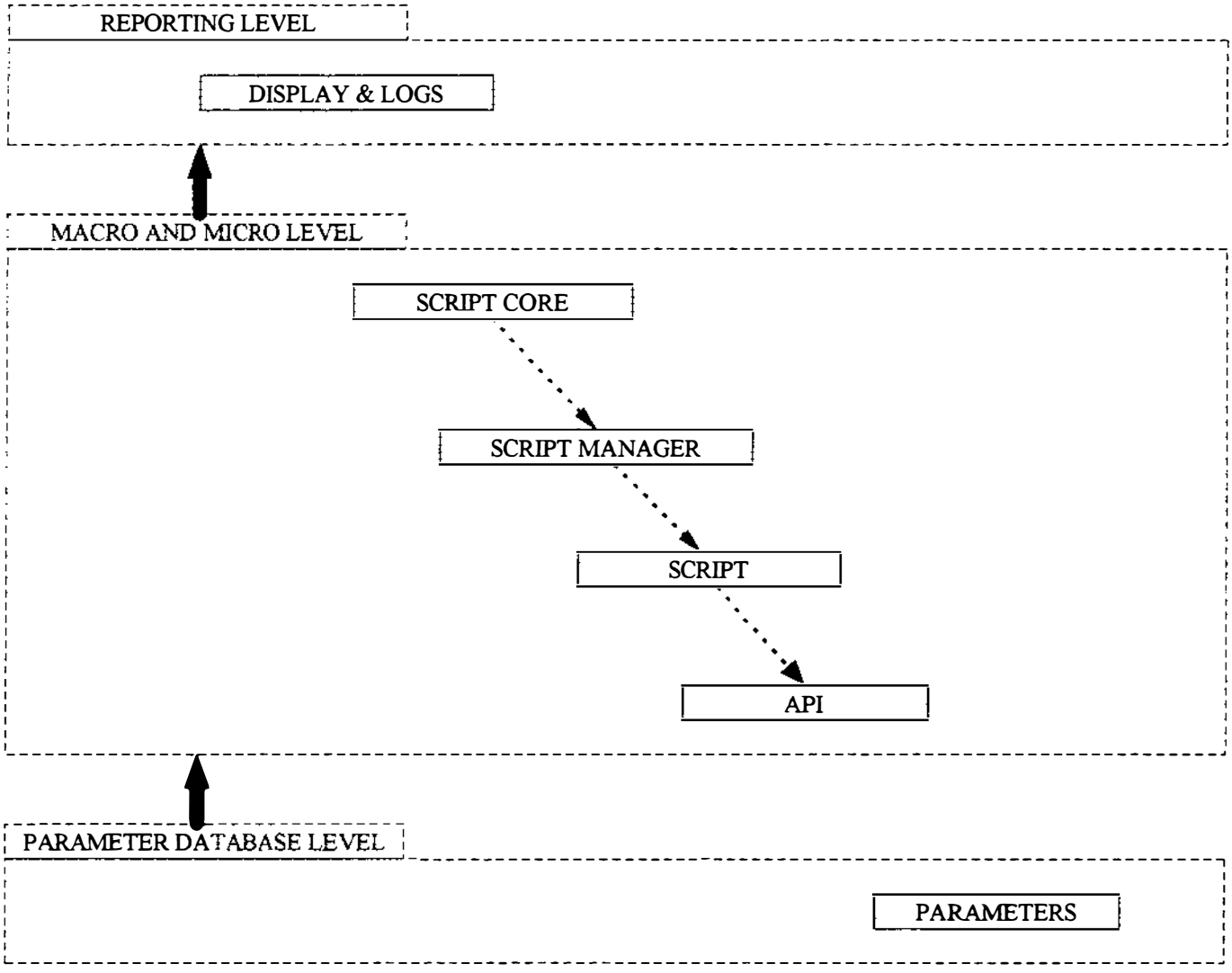
Engineering managers should read the entire document and pay attention to the process of Script Channeling. Engineering managers will know what areas of their product are critical sections and may plan to use the Script Channeller to test those components. You may determine at what phase in the software cycle to deploy the application. You may also provide specifications on the execution and process logs of the Script Channeller application.

Software development engineers

You should read the entire paper to understand the methodology. Once the Script Channeling methodology is clear, an application can be designed to suite your API set. The application may be designed to exercise critical sections of your software components. Keeping in mind that the usability and extendibility of the application during the design phase. The application should be easily executed and the selection and modification of the parameters in the Parameter Database should be maintainable.

Software test engineers

You should pay attention to the idea of the methodology and concentrate on the Script Manager level, Parameter Database, process logging, and reporting sections. You will ultimately be the ones using the application and will provide important feedback on the implementation of the application and the problems with the APIs.



Legend:

Direction of data flow

Direction of process control

Script Channeller Hierarchy FIGURE 1

I. Introduction

The Script Channeller methodology is a means of testing a set of APIs by using only one instance of each API in the life of the executing script. If several calls to the APIs are required during the script execution, the same API will be used but parameter values will change. This reduces the programming errors that may occur during normal software development. Since scripts are written and proven to work for a particular task, the same script may be reused in future to regress on the current problem. This means that the script may be stored and reused at a later date by other script calls or for regression testing.

Because of the method of Script Channeling, only one micro object needs to be made available while a vast amount of parameters may be passed to it. This will change the standard methods of testing of producing numerous sample programs which are similar but perform different tasks. Script Channeling will perform many different task with Scripts and Script Managers by combining with the database of parameters to perform full functionality testing in one application. This methodology of software testing will reduce the total time period required for producing test applications, reduce maintenance of test applications, increase the broadness of testing with the Parameter Database, increases the effectiveness of producing immediate results, produces effective automation with scripts and Script Managers, and simplify the regression process.

This document describes a technique that has already been demonstrated and proven. The Script Channeller will provide engineers with the knowledge to achieve practical increments in improvements in the software quality of their products.

II. Planning

The Script Channeller application that is to be developed using this methodology will not be a conventional application were the user will expect predictable results from the execution of the scripts. Since the data that is supplied to the scripts will be of a valid and invalid nature, the outcome of the execution can produce erroneous results as well as valid results. Because of the unpredictable values that are supplied to the APIs, the application may crash due to erroneous values. These are the problems that the methodology intends to find which will result in a more robust set of APIs. The whole purpose is to find the problems before the ISV finds them.

One feature that needs to be determined before the development of the Script Channeller is how the scripts are going to be executed. There are a number of ways that the user may execute the scripts depending on their technical background. The execution of the script may be through commands that are written in an editable format. That is, once the user has written the commands with the necessary parameter values, he may execute the scripts.

Another method of executing the script is by having a predefined set of scripts that read in values from the Parameter Database. Once executed, the scripts will read in the values from the Parameter Database and pass it over to the APIs.

The Parameter Database may be stored in different forms. It can be stored in a file, database, structure, or user entry. The effectiveness of the testing will depend on the richness of the Parameter Database and how extendible the Parameter Database is. For example, if I choose to have the parameters stored in a file, it will be difficult to maintain the file on it own. It will be advantageous to have an editor that modifies and organizes the file.

III. Components

A. *Micro objects*

1. APIs

The APIs are the Application Programming Interfaces that will be the lowest level used in the Script Channeling methodology. The APIs to be used in this layer are determined by the creator of the application that uses this methodology. For example, if the developer of the Script Channeller application decides to write an application that retrieves data from a database, he will have connection APIs and retrieval APIs. All the APIs used in the connection and retrieval will be in the micro object level. The APIs will exist in a script that will contain cycles and logging. Some specialized scripts may contain other code such as displaying a dialog box to validate a window handle. In the Script Channeling methodology, the APIs resides in the Micro Level because this level is the slave of a higher level, Macro Level, that drives the Micro Level.

The advantage of using the micro objects is that since the objective of the assurance is to make sure that the APIs work within a reasonable constraint, only one statement of the API is written in the application and any problem residing from the API can be immediately isolated to the file that contains all the Micro Level APIs source code. Later on, I will discuss about the process logging that will show the steps that have been taken to reach a particular API.

B. *Macro objects*

1. Script

The script is a layer that encompasses the API. The script layer will handle the calls from the Script Manager layer and directly call the APIs.

The parameters that are set by the Script Manager level will be passed to the Script Level and executed by the APIs. These parameters can be of many forms. A parameter could be a value, variable, file, text, definition, and so on. When the parameters from the Script Manager are passed to the Script Level, the Script Level will extract the information and execute the APIs according to the extracted parameter information. For example, if a table is passed from the Script Manager to the script, the script will need to obtain each row of the table and pass each line of the table to the script until the end of the table has been reached.

The script layer also provides the process logging of the Micro Level. The API that is called will be logged in a log file. Together with the logged API, all the parameter values that were used to execute the API will also be logged. After the execution of the API, all the return values and return parameters will be logged for determining the outcome of the executed API. The user of the Script Channeller application can determine the outcome of the execution by observing the process logging problems.

The script layer also calls a display mechanism for writing the inputs and outputs of the API to a screen dump or log file. This logging of API calls will provide the user with the process of the API calls that are made when the Script Manager executes the scripts.

2. Script manager

The Script Manager is a layer that holds the sequence of script calls. The quantity, duration, parameters of the scripts are determined at the Script Manager level.

The Script Manager level contains a sequence of scripts that may be executed by the user of the Script Channeller application. The user may select any one of the available Script Managers and execute it.

3. Script core

The Script Core is the foundation of the application in which the Script Channeller is built. It contains the entry point into the program. It creates and initializes the objects after the program has been launched. The Script Core also handles the windows messages, windows control, creation of threads or processes, initializing variables, and handling exiting.

C. Parameter database

1. Parameter database

The Parameter Database contains all the parameters of the all APIs at the Micro objects level. The parameters could be of string, integer, real, floating point, structure, and so on stored in a file, database, structure, or simply hard-coded. The best form of parameter storage is to have the parameters directly accessible through an editing tool such as a text editing program or a database browser.

The Parameter Database holds all the parameter values of the APIs in a database. For each API, it requires parameters of certain types and ranges. The database will hold valid, boundary, fail, and stress values for the application to call.

D. Share code

1. Error handling

The error handling will provide the user with a description of the problems that have been encountered with the Script Channeller. Exceptions may be raised from the Micro Object level to the Macro Object level so that the upper level can handle the returns from the APIs called at the Micro Object level.

2. Initialization file

The initialization file will contain all the initialization values for starting the Script Channeller application.

E. Reporting

1. Process logging

The process logging is a display of all the APIs called at the Micro Object level. For each execution of the Script Manager, the APIs that are called by the script which is in turn called by the Script Manager are written sequentially into a log file. Each new call to an API and its input and output will be written to the log file. The information of the APIs are appended to the end of the log file so that the called APIs can be traced.

Other instrumentation can take place such as recursion count, cycle count, time stamps, error code, and error strings.

2. Display window

The display window is the standard output of the Script Channeller application. It will contain information similar to process logs.

3. Automated pass/fail reporting

For each API, a list of expected return parameters and range of values can be listed in the Parameter Database. This list may be compared at run time with the process logs to determine if the execution was successful or failed. A summary of the process logs with pass/fail indicators can be added to the report.

4. Source code generator

The program may be designed to generate source code of the APIs that were called during the execution of the scripts. The code that is generated should be in the API format of a statement. This source code generation will make it simple for the engineer to add other statements and controls to the generated code as source samples for development and debugging.

The source code generator statements are sequentially the same as the processing logs except that the logs contains the inputs and outputs of the APIs called.

5. Problem location identifier

During the course of executing the scripts, some APIs are bound to return undesired values. This mechanism may be used to alert the user of a particular API called during the execution of the scripts.

6. Pause

A pause is temporary termination of execution of the scripts when a problem occurs. The user can identify the process logged so far and determine the problem before the rest of the scripts are executed as part of the Script Manager.

F. Cycles

1. Micro cycles

The Micro Cycles is a term used to describe the process of determining the reliability of an API when it is called numerous times. The cycles can be used to determine if the API responded appropriately after a few calls. It can be used to determine if the handle count is changing, memory is allocated, adverse responses such a general protection fault, or validating returned parameters and values.

The input parameter values at this level will be constant and cannot be changed. The input parameter values are passed from the Macro Level to the Micro Level.

2. Script cycles

Script Cycles is a term which describes the cycles that are used to loop the Micro Level with various input parameter values. The input parameter values may be the same or the values may be different.

At the Micro Level, the same call to the API at that level may not be valid more than once. The Macro Level should handle Micro level APIs that do permit and do not permit more than one instance of the call and make sure that reverse effect of the call is executed in the same quantity. That is if the same parameters were used to create a thread, a new thread should be created each time the API is called. However by doing this, the user must be aware that creating numerous threads will cause processor resources to be used up and slow system performance unless the threads are destroyed. If the purpose of calling numerous APIs is desired, it can be monitored by other performance monitoring tools available to determine system resources, CPU speed, memory allocation, and so on. At times, some APIs may not permit a second instance of an object once the first instance is created. That is, the first instance must be released before the same API can be called again. In this case, all calls to the API after the first call will be invalid and the API must respond within reasonable expectations.

3. Script manager cycles

Script Manager Cycles are used to perform a specific task with a combination of scripts. The task may be to allocate and de-allocate the memory. The purpose is to fully perform the requirements of the task and clean up after the execution so that after the termination of the task, the state of the program or machine will be as if the task had not even run.

The purpose of Script Manager cycles is to determine if there will be any change in the state of the program or machine when a task is executed numerous times.

IV. Deployment

A. *Users and User Interface*

Depending on the Script Channeller user, the user interface of the may cater for high end user or a low end user. If the user is unfamiliar with the software details and is interested only with the pass or fail of a test case, the user interface may than be driven by check boxes and push buttons. The Parameter Database may also contains expected inputs and outputs of the APIs. It may also contain ranges of values in which the parameters may operate. By adding expected values into the database, the Script Manager may determine whether to automatically display pass or fail when the scripts are run. If the user is more familiar with the software details, software definition language may be used to mimic the Script Managers so that it may call on the scripts with a selected Parameter Database.

B. *Identifying problems*

The problems of the executed scripts may be determined from the process log. This is done by observation. An ideal process log will also contain the pass or fail execution of the script which will immediately flag to the user an immanent problem with the APIs.

V. Factual Summary

A. *Speed*

One draw back of this methodology is the speed compromise due to the large amount of data passing between the Micro Level and the Macro Level. Since all the APIs parameters and return values are logged into a process log and display window, the speed of execution also depends on the speed of disk media and video on which the Script Channeller is running.

B. *Parameters*

The disadvantage of the Parameter Database is the need to distinguish between the valid and invalid parameters. You will need to work out the parameters that work together with other scripts to perform a task and the expected return or output from the APIs. This part will be time consuming but the advantage is that once they have been worked out, the same scripts can be run over and over through future revisions to determine any breakage in the underlying API code. The main advantage is that since the expected return or output has already been determined, any changes to the design or architecture of the API can be validated with the scripts by rerunning them.

C. *Logs*

The logs may be used to determine what script was run and what Parameter Database was used. At the same time, the point of the problem may be isolated by observation of the logs. By observing the logs, the input and output of the problem API may be determined and reported as a problem. The process of how

the problem arrived may also be determined from the process log. This will make it simple for problems to be isolated and fixed and there is always an immediate reproduction which may be determined from the process logs.

I. Conclusions

After this methodology has been implemented and deployed in an assigned product or component, there will be a dramatic improvement in the speed of regression of test cases. The Software Test Engineer will find that any code changes can be verified simply by rerunning the scripts to test for breakage or bugs; the process to trace the problem is made simple by reviewing the process log file. The Software Development Engineer will find a reduce number of applications that are required to be developed and maintained; only one application will be written using the methodology while several scripts will perform testing of the APIs. The Engineering Managers will find better coherence between test and development as problems found with the product will immediately be traceable with process logs. Overall communication should improve between team members as problem claims can be substantiated with scripts, logs and problem reports.

With the introduction of scripts, parameter databases, process logs and so on to your application, the Script Channeling methodology will benefit software engineering teams in the process of developing and testing products and components to provide substantial increase in productivity, clarity of trouble shooting, better reporting, diversity of test cases, reusable code and scripts, and shorter turn around time in testing and development.

Software Test and Evaluation Measurement: An Industry Update

Bill Hetzel, Software Quality Engineering

Abstract

This talk provides a progress report on measurements and metrics in use for evaluating and assessing software test and evaluation.

About the speaker

Bill Hetzel is a principal in Software Quality Engineering based in Jacksonville, Florida. Software Quality Engineering is a consulting firm with a special emphasis on better software engineering and quality assurance.

Mr. Hetzel is well-known as a software testing consultant and regularly presents industry seminars on testing and test management. He has helped many companies develop and implement better testing standards and procedures. He has authored several books including The Complete Guide to Software Testing *(John Wiley & Sons, Inc., 1988) and* Making Software Measurement Work: Building an Effective Measurement Program *(John Wiley & Sons, Inc., 1993). A graduate of Brown University, Mr. Hetzel earned an M.S. degree from Rensselear Polytechnic Institute and a Ph.D. in Computer Science from the University of North Carolina.*

Test & Evaluation Measurement
An Industry Update

PNWSQC
October 29, 1996 Portland, Oregon

Bill Hetzel
Software Quality Engineering

SQE

1. Understanding T&E Measurement

SQE

What is T&E?

The purpose of T&E is to validate (is this what what we want) and verify (is this correct) each of the software project deliverables; identifying any defects in those deliverables in a timely manner.

Activities whose primary purpose or aim is evaluation and determination of fitness for use.

SQE

What is T&E Measurement?

'Quantified observations' about any aspect of T&E (product, process or project)

Use of these observations to understand and manage T&E effectively

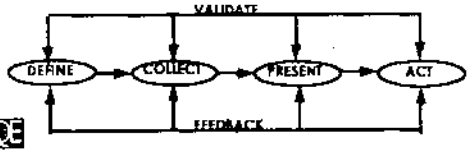
"Easy to get 'numbers', what is hard is to be sure they are right and understand their meaning"

SQE

T&E Measurement Practices

Our 'behaviors' in measuring T&E

Simplified Measurement Process Model



SQE

Measuring T&E is Hard!

T&E is itself software measurement. Measuring T&E is meta-measurement and that's doubly hard

Significant process maturity and supporting infrastructure is required

Most companies lack the pre-requisites
Testability Maturity is WEAK

SQE

2. Distinguishing Good, Better and Best Practices

What is Better/Best Practice?

Practice that usually produces better or best results

Implies criteria
to define usage
to rank results and define "better"

A Short Test (1)

True or False?

Brushing your teeth every day is a good practice?

A Short Test (2)

True or False?

Brushing your teeth after every meal is a better practice?

A Short Test (3)

True or False?

Brushing your teeth after every bite is an even better practice?

Question:
What Happened??

Answer: The criteria changed!!

MORE (even of a good thing) is not always BETTER

Good/better/best practice has no meaning until you can agree upon the ranking (measurement) criteria to use

3. What is 'Typical' Industry Practice?

SQE

Learning about Practices

- **Tons of information available**
  Case studies, books, conferences, seminars, experience sharing, etc
- **Practice benchmarks/surveys**
  'Measurements' of the practice

SQE

SPRC Practices Study

Ongoing Empirical Research
- **Measure what industry is doing**
- **Examine trends and impacts**

Software Measurement since 1991

Software Test & Evaluation since 1988

SQE

'Usage' & 'Value' Scales

Usage

0 NOTUSED
1 INFREQUENT—some of the time
2 COMMON—most of the time
3 STANDARD—all the time

Value

0 UNIMPORTANT—a waste of time
1 LIMITED—would be nice
2 SIGNIFICANT—recommended
3 CRITICAL—should be standard

SQE

Participants

- **Survey respondents include**
  Conference attendees
  Seminar attendees and selected SQE clients
  Others as opportunities or interest arises
- **Not a typical or random sample**
  Experienced practitioners and managers
  Larger software organizations
  Attending test and measurement events

Better than average practices

SQE

Most Measures are Little Used

Almost all the measurements surveyed are INFREQUENTLY USED or NOT USED at all by the majority of the participants

T&E measurements, especially those measuring benefits and effectiveness show particularly low usage

SQE

Almost All are Highly Valued

Many by over 90% of the participants

Nearly all by at least half of the participants

Usage is not low due to low value!

Usage of "Core" Metrics

Schedule	62%
Defects (in testing)	58%
Defects (after release)	56%
Effort by Activity	41%
Size (LOC or FPs)	34%

Many are not yet collecting SEI's core metrics

Fewer still really use and act on them effectively

Test Mis-Management?

Effort spent on testing and reviews measured?

Not at all	36%	} 71%
Infrequently	35%	
On most projects	17%	
All projects	11%	

Test efficiency and effectiveness measured?

Not at all	49%	} 84%
Infrequently	35%	
On most projects	10%	
All projects	6%	

Analyzing Effectiveness

Percentage using most or all of the time

	87	88	89	90	91	92	93	94	95	96
Faults and defects analyzed			30	27	26	32	24	28	21	23
Cost of testing measured	24	21	25	28	31	29	24	25	28	29
Cost of debugging separated	21	18	22	20	23	24	26	26	22	26
Effectiveness measured	15	18	14	16	15	23	19	25	16	21

Only a few do it well

Understanding Coverage

Percentage using most or all of the time

	88	89	90	91	92	93	94	95	96
Test objectives inventoried								21	35
Requirements coverage analyzed	24	28	32	45	42	32	24	34	36
Code coverage analyzed	13	14	18	18	27	28	23	18	19

Most Don't!

Documenting Tests

Percentage using most or all of the time

	88	89	90	91	92	93	94	95	96
Tests identified and named	52	60	58	65	61	55	55	65	72
Test procedures documented		50	51	53	59	50	62	64	74
Test summary reports	46	50	52	67	65	64	59	60	67

Many Don't!

Unit Testing Practice?

Percentage using most or all of the time

	Unit Level	System Level
Test procedures documented	24%	67%
Test summary reports	14%	60%
Defects tracked	21%	71%

"In most organizations there is no unit testing process... unit testing is invisible to management or anyone else on the team."

Individual Variation

- **BIG variations in usage**
- **Bottom quartile - almost no measures**
- **Top quartile - many of the measures**
- **Top decile - nearly all of the measures**

State of the Practice (1)

T&E measurement practice is very immature

Most don't track effort or time by activity or level
Most don't measure or track their testware
Very few analyze the defects and failures missed
Very few track test coverage and what isn't covered
Many don't understand the patterns of defects found
Very few understand or analyze T&E effectiveness

Most are unable to manage the key issues

State of the Practice (2)

"...test and evaluation practice remains largely unsystematic, unmeasured and unmanaged..."

"... big differences between better, best and the rest..."

4. What is "Best" (or at least Better) Practice?

Make your Testware VISIBLE

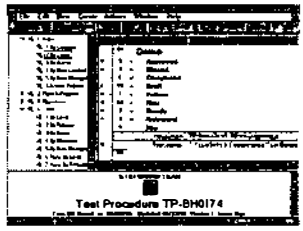
Track your test assets, status, when used or run and results

Measure static attributes and coverage

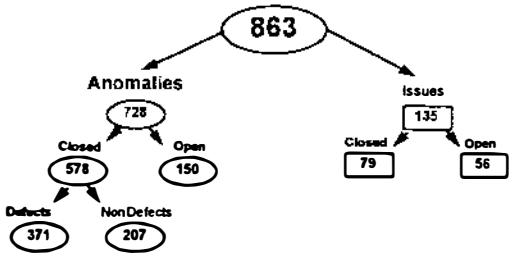
Work to understand the defects you find and miss (and why)

T&E Management Systems

- Maintain your tests in a visible repository
- Track status and execution
  - Objectives
  - Specs
  - Tests
  - Changes
  - Runs
  - Run Results
  - ...

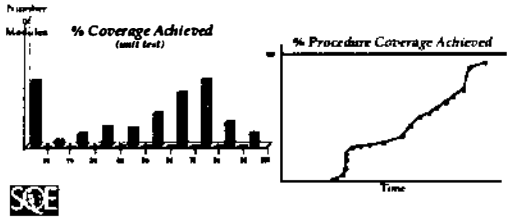


Problem Analysis



Coverage Analysis

- Make coverage measurement routine
- Examine and understand what you don't cover
- Consider requirements, design and code coverage



Stress the end RESULTS

- Measure benefits and effectiveness
- What gets missed - not just what is found
- What doesn't work - not just what does

Perceptions can and should be quantified

Defect Detection Effectiveness

$$DDE = \frac{\text{Defects Found}}{\text{Defects Present}} \times 100\%$$

DMP = Defects Missed Pct = 100% - DDE

- Measure each T&E stage or level
- May weight by severity or impact
- Drops over time as defects are found
- Always an over-estimate for the true DDE

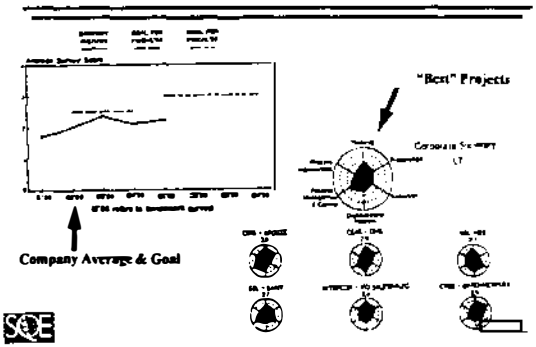
Defect Age

$$DDAge = \sum_{Phases} (\text{Phase Found} - \text{Phase Introduced})^2 / \#\ \text{Defects}$$

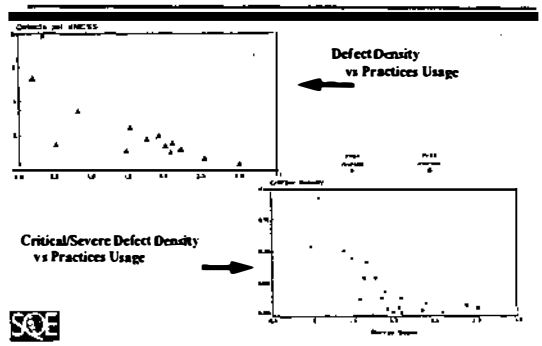
$$DFAge = \sum_{Phases} (\text{Phase Fixed} - \text{Phase Detected})^2 / \#\ \text{Defects}$$

- Measures whether you are finding/fixing defects sooner (a good goal for T&E)
- Age of 0 if found/fixed in the same "phase"

Use of Test Practices



"Results" of Test Practices



Don't Forget the Customer

How does the customer really feel?

Do your T&E measurements correlate with measured customer satisfaction?

Think DOLLARS and SENSE

Understand effort and cost

Relate benefits and results to effort

Try to get a handle on rework and waste

Make measures make business sense

Rework

1165 Total Problems

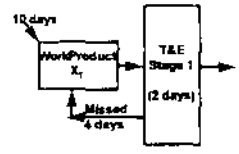
1 day	71%	< 1 week	5%
< 2 day	15%	< 2 weeks	1%
< 3 day	7%	< 1 mos	.08%
		< 3 mos	.02%

This 7% required more fix time than the other 93% combined

"Measure" the rework on "big" defects
Use a plug factor for the little ones

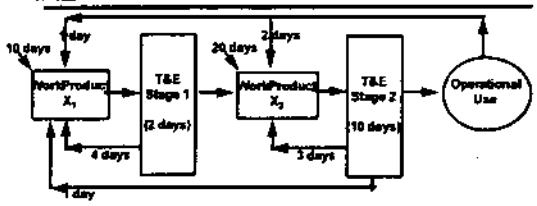
Rework Missed

OOPS
A measure of work effort 'missed'



$OOPS_{X1}$ = 4/12 = 33%

OOPS Example Continued



Project OOPS
$OOPS_{X_1}$ = 6/12 = 50%
$OOPS_{X_2}$ = 5/30 = 17%
$OOPS_{SYS}$ =11/42 = 26%

Test OOPS
$OOPS_{Stage\ 1}$ = 2/2 = 100%
$OOPS_{Stage\ 2}$ = 2/10 = 20%
$OOPS_{T\&E}$ = 4/12 = 33%

SQE

Focus on REALITY

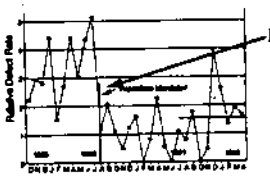
Measures aren't goals or status reports

Stay grounded in application to practice and improved understanding

Most important goal is TRUST and confidence of the project team

SQE

Defective Fix Rate Example



Inspection of all fixes mandated (Three-fold improvement in % of defective fixes)

Question: What caused the big improvement?
Hint: It was NOT because defects were found during the inspections!!

SQE

Measure Your Measures

Ongoing meta-measurement is crucial
- **Who is (is not) using the measures**
- **How accurate (inaccurate) are they**
- **What decisions/actions are being taken?**
- **Are they valid?**

TRUTH as the highest purpose
- **Detect and root out dysfunction**
- **Emphasize reality and understanding**

SQE

Summary

- **T&E measurement is a key practice area**
- **Invest in infrastructure and tools - build measures in as part of the work flow**
- **Make T&E activities and results visible**
- **Focus on what is being missed and the rework effort or waste**
- **Motivate truth and understanding**

SQE

Vision

"All of us (practitioners and managers) really <u>USING</u> good measurements as a part of all our practices and key decisions"

- ***Building good measurements into our processes, tools and technology***
- ***Requiring good measurements before taking action***
- ***Expecting good measurements as we act***
- ***Insisting on good measurements of our measurements***

Measurement as a way of life!

SQE

A Major Opportunity

- T&E is a major part of all software efforts
- Effective measurement is critical
- Many different aspects to consider
- Many approaches and strategies for success

SQE

Discussion

SQE

Software Testing as Acceptance Sampling

Jarrett Rosenberg

Sun Microsystems
2550 Garcia Avenue, MPK17-307
Mountain View, CA 94043
Rosenberg@Eng.Sun.COM

The purpose of software testing is to accurately determine the reliability of the software under test. As such, it is fundamentally a statistical problem, yet this aspect of it is often ignored. This paper describes how an application of a statistical quality control method, acceptance sampling, provides a way of answering the question "How much testing is enough?"

Jarrett Rosenberg is a statistician and quality engineer at Sun Microsystems. He received his doctorate from the University of California at Berkeley, and worked at Xerox and Hewlett-Packard before joining Sun in 1990. He is an ASQC Certified Quality Engineer and Reliability Engineer, and a Senior Member of the Institute of Industrial Engineers.

Pacific Northwest Software Quality Conference, Portland OR, Sept. 1996

Software Testing as Acceptance Sampling

Jarrett Rosenberg

Sun Microsystems
2550 Garcia Avenue, MPK17-307
Mountain View, CA 94043
Rosenberg@Eng.Sun.COM

The purpose of software testing is to accurately determine the reliability of the software under test. As such, it is fundamentally a statistical problem, yet this aspect of it is often ignored. This paper describes how an application of a statistical quality control method, acceptance sampling, provides a way of answering the question "How much testing is enough?"

1.0 Introduction

As software becomes ubiquitous, consumers are increasingly concerned about its quality and reliability. Eventually there will be inescapable demands for an objective, rigorous certification of a given software's reliability. Yet, the central problem of software testing is this: there are far too many possible inputs and execution paths for all of them (or even an appreciable proportion of them) to be tested. How then can one be assured that a given program will execute correctly? Since the problem is a statistical one, the answer must be statistical as well.

The use of statistical methods in ascertaining quality and reliability goes back many decades, and a variety of techniques have been developed to guide the testing process. These techniques can be used either for *estimation* ("How many remaining defects are there?") or for *prediction* ("What is the probability of a failure occurring by n hours?"). Furthermore, these estimates and predictions can be phrased in either temporal or atemporal terms. The former asks about defect or failure rates over some period of time, while the latter simply asks about "total" defects or failures; both perspectives are useful.

While temporal approaches to software reliability have always emphasized the statistical nature of software testing atemporal approaches have typically ignored this aspect, focusing instead on metrics or questions of code coverage. While these questions are important, they cannot provide quantitative answers to basic questions of software testing such as:

- How many defects are present?
- What is the likelihood of a failure occurring?
- How much testing is enough to reach a given level of confidence in a particular reliability estimate or prediction?

These are statistical questions, and there must be statistical methods for answering them. This paper describes the statistical aspect of software testing and how it provides answers to these questions. The unifying framework is that of stochastic testing, which has both temporal and atemporal reliability methods; this paper will focus on the primary atemporal one, acceptance sampling. Before we start, however, some terms must be defined.

2.0 Software Testing

Following IEEE [1983], we say that programming *errors* introduce software *defects* or *faults*, which are code that is syntactically, semantically, or pragmatically incorrect, and which may under some conditions cause *failures*: performance of the software that deviates from its intended behavior.

A *test* is an execution of the software, producing a result that is compared to the correct result (generated by some official source, the *oracle*). If the obtained result agrees with that of the oracle, the test is successful or passed. otherwise a failure has occurred. Generating an error condition can be a successful result, of course.

Tests can be categorized along various dimensions:

1. The use of knowledge about the code being testing.
2. The size and scope of the software being tested.
3. The kind of data flow coverage involved in the test.
4. Whether the testing is exhaustive or not.
5. The context in which the testing takes place.
6. Whether the code is modified or not.

While these dimensions are logically orthogonal, in practice we find that unit testing is more often structural and deterministic, with more detailed coverage, and without use of an operational profile, while system testing is more often the opposite in all these respects. There are, moreover, classes of defects which can only be uncovered in system testing, such as timing and load problems.

Because it is impossible to test all inputs. most forms of testing turn out to be some variation of either haphazard or stochastic testing. The former can give us little assurance about the reliability of the software, while the latter provides a variety of methods for doing so.

3.0 Stochastic Testing

In stochastic testing, sequences of tests are executed by one or more identically configured machines starting from some reproducible initial state. Testing is terminated after some pre-determined period of time or number of failures has occurred, and statistical methods are used to provide estimates of defect and failure rates. A critical requirement for the use of statistical methods is satisfying their assumption of valid

sampling: the sample of tests must be an accurate representation of the entire population; in practice this means that the sample must be drawn randomly. The first step in meeting this requirement therefore is the specification of the sample frame, that is, the set of all possible tests. This is basically the problem of specifying the input language.

3.1 Specifying the Test Inputs

The set of all possible test inputs can be described by a grammar, with the actual input used in a test considered as a sample of the infinite set of sentences generated by that grammar (see Maurer [1990] for a practical application of this basic principle). The alphabet of the grammar is the set of basic test inputs, and the grammar's productions describe how the basic inputs can be combined into appropriate sequences. In addition, the grammar can be probabilistic, with frequency weights attached to the individual alphabetic elements and the productions as desired, thus changing the distribution of sentences in the language. This allows the sampling frame to be either "all logically possible inputs" or "all inputs conforming to a given operational profile". The determination of the sampling frame is then an explicit choice on the part of the tester in specifying the grammar and determines the domain of generalizability of the test results.

Stochastic testing thus involves generating a random sample of sentences from the input language by means of the grammar and using them as test inputs. If the sentences are randomly sampled, then statistical theory can produce the results we want. A critical question is thus "How do we know the input sentences are a representative sample?" The answer is that (a) we know the alphabet (the set of basic inputs) and we know (or can stipulate) its frequency distribution; (b) we know the set of production rules and we know (or can stipulate) its frequency distribution; (c) we can use a randomization device to select sentences at random from the language; (d) we can (in fact, must) use large samples, which will minimize any remaining unknown biases in the sample.

This fact, that we can be assured that the input test sample is a random sample from the universe of tests and thus satisfies the basic assumption of statistical methods, is what distinguishes stochastic testing from merely haphazard testing, which can never assure us that the subset of tested inputs validly represents *all* inputs.[1]

3.2 Evaluating the Test Results

The problem of evaluating the result of a test is the greatest technical challenge in software test engineering, especially for client/server and graphical user interface software. The ideal is for the test generator to also be the oracle and/or evaluator, but the ideal is far from being a reality.

From a theoretical perspective, however, the important issue is whether or not the oracle can be assumed to be always accurate ("infallible"). If not, then we must model the testing situation by including a probability that the oracle misses a failure, or even that the oracle incorrectly reports a failure when there is none. Fortunately, in stochastic testing "fallible oracle" models can be easily made by modifying the standard models to incorporate this factor, as described below. To simplify the exposition, then, we can assume that an oracle exists and that it is infallible.

1. A common misconception is that non-random distribution of defects in the code being tested invalidates the randomness assumption of stochastic testing. In fact, it is the randomness of the *sample* that is critical. To take a concrete example, even if a basket of fruit has the bad pieces at the bottom and the good ones on top, a random sample (which will include pieces from throughout the basket) will produce an accurate assessment.

3.3 Estimating Defect and Failure Metrics

Once we have generated a random sample of tests, executed them, and evaluated the results, we require a quantitative method for estimating and/or predicting the number of defects and/or failures. A number of techniques have been developed over the past half-century for just this purpose. the principal temporal method is *reliability measurement,* while the principal atemporal method is *acceptance sampling*. Surprisingly, even though acceptance sampling was developed earlier, and is the method most directly applicable to classical software testing methods it is much less familiar to software test engineers than reliability measurement methods.

4.0 Acceptance Sampling

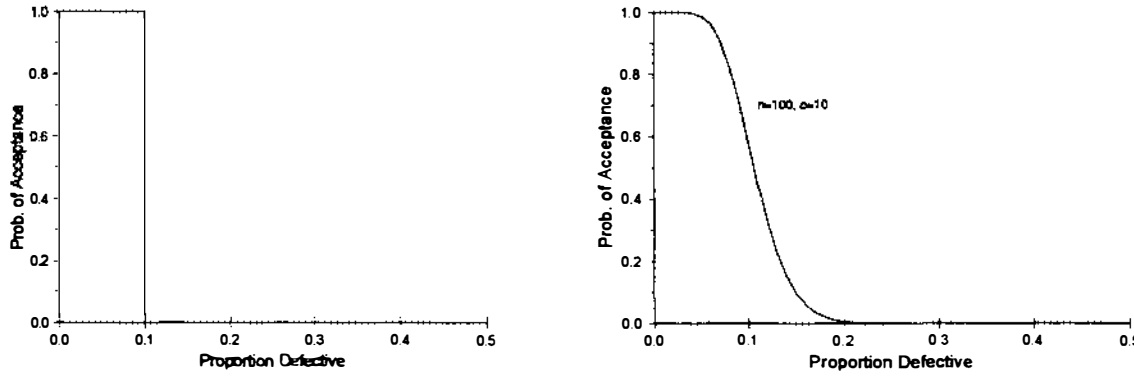
Acceptance sampling was developed in the 1930's and '40's as a quality assurance method for the supplying of product from producers to consumers.[1] Its goal is to provide a valid way of assessing the quality level of a product lot by measuring the quality of a sample from that lot. In such an assessment, there are two possible errors that can arise:

- a lot with an acceptable quality level can nevertheless be rejected (a Type I error, the probability of which is called the *producer's risk,* symbolized by α).
- a lot with an unacceptable quality level can nevertheless be accepted (a Type II error, the probability of which is called the *consumer's risk,* symbolized by β).

The challenge of acceptance sampling is to determine the appropriate sample size to examine, given predetermined levels of acceptable quality acceptable consumer's risk, and acceptable producer's risk.

An acceptance sampling plan can be most concisely characterized by its *operating characteristic* (OC) curve; an ideal OC curve is shown in Figure 1a. The ideal is that a lot with a quality level (defect rate) less than the predetermined Acceptable Quality Level (AQL) will always be rejected, while one at or above the AQL will always be accepted; this reduces both the producer's and consumer's risk of an error to zero.

FIGURE 1. Operating Characteristic Curves: (a) The Ideal, (b) An Actual Plan ("n = 100, c = 0").



1. The standard reference is Schilling [1982]; see also Duncan [1974].

In the real world, there is always some degree of uncertainty due to the error inherent in sampling, and so a typical OC curve will look more like Figure 1b. Here there is a small probability both of rejecting a good lot and accepting a bad one, the typical values of 0.05 and 0.1 being used for these, respectively. The degree to which an OC curve approaches the ideal shape for a given level of quality and producer's and consumer's risk is dependent on two factors: the size of the sample examined, n, and the *acceptance number* or number of defective items allowed in it, c. In the sampling plan depicted in Figure 1b (called an "$n = 100$, $c = 10$" plan),

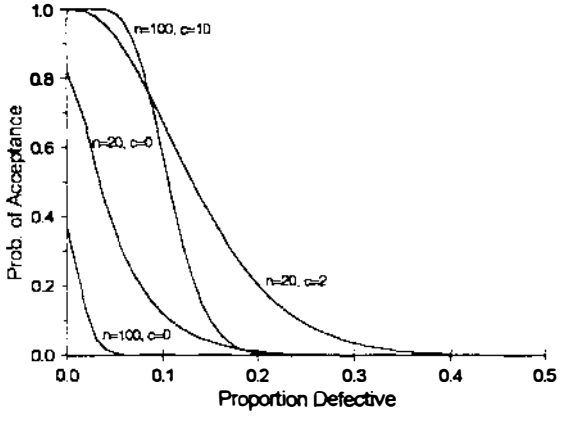
- the AQL is 10% defects in the lot,
- the consumer's risk (probability of accepting a lot with a higher percentage of defects) is 0.1,
- the producer's risk (probability of rejecting a lot with a lower percentage of defects) is 0.05,
- the sample size is 100,
- the number of defective items allowed in the sample is 2 (i.e., if no more than 2 of the examined units are defective, the lot is accepted).

In operation, if a sample of 100 units is examined and one is found to be defective, we can conclude with 90% confidence that the lot has no more than 1% defective units in it, and so it is accepted. If three units in the sample had been defective, we would have been 95% confident in believing that the lot had more than 1% defective units in it, and would have rejected it.

With this acceptance sampling plan there is a 95% chance of accepting a lot with 10% defective units in it, but only a 20% chance of accepting a lot with 15% defective units in it. If we were to keep the sample size the same and lower the acceptance number to, say, zero instead of two (an "$n = 100$, $c = 0$" plan), then there would only be a 35% chance of accepting a lot with 1% defective units in it (not a good idea if we were actually willing to accept that many defects, but very useful if the desired quality level were 0.1%).

Figure 2 shows the OC curves for several different plans. Note that the sample size, n, has the greatest effect: both the "$n = 20$, $c = 2$" and the "$n = 100$, $c = 10$" plans aim for the same level of defects ($c/n = 10\%$), but the plan with the larger sample size has a curve much closer to the ideal. Note also how increasing the sample size moves the
"$c = 0$" plans closer to the ordinate.

FIGURE 2. Operating Characteristic Curves for Several Acceptance Sampling Plans.



Over the past half-century a wide variety of acceptance sampling techniques have been developed, including ones which incorporate errors in evaluating whether a unit is defective (this can be easily handled by appropriate changes to the producer's and consumer's risk levels). Of these many variations, the most relevant to software testing is *conformance sampling*, where the number of defects allowed in the sample is set to zero, and the emphasis is on reducing the consumer's risk.

5.0 Software Testing as Acceptance Sampling

The relationship between stochastic software testing and acceptance sampling is more than just analogy: the stochastic testing process in its atemporal aspect *is* an acceptance sampling procedure, where a random sample of possible computations are examined for failures.

In the case of minor failures, where some non-zero level is acceptable, classical acceptance sampling is directly applicable. For example if we assume typical producer's and consumer's risks of 0.05 and 0.1, respectively, then a test of 1000 randomly chosen inputs that produces three failures leaves us 90% confident that at most 0.8% of inputs will cause a failure. Testing 5000 cases with three failures leaves us 90% confident that 0.2% of inputs will cause a failure.

Of more importance is the case of critical failures, where the Acceptable Quality Level is 0%. A variation of acceptance sampling called conformance sampling (see Schilling [1982]) has been developed to handle just this case. In conformance sampling, the sample acceptance number is set to zero, and the emphasis is on reducing the consumer's risk (because we are not worried about accidentally rejecting a perfect lot).

The case of zero defects in the sample is a special one because it is like trying to prove a negative; we can only specify a high level of confidence that the actual defect level is very low. The emphasis then is on specifying an upper bound for it.

The simplest conformance sampling plan treats the tests as a series of Bernoulli (binary outcome) trials, and thus typically uses the binomial distribution as a model. The sample size needed is determined by the formula (Hahn [1974])

$$N_C(P) = \log\left(1 - \frac{C}{100}\right) / \log\left(1 - \frac{P}{100}\right) \quad (1)$$

where C is the percent confidence desired, and P is the acceptable percentage of defects. For example, to be 99.9% confident that the actual defect rate (percent of inputs that cause a failure) is no greater than 0.01%, some 69,074 failure-free tests are needed. Conversely, after 69,074 failure-free tests, we can be 99.9% confident that the upper bound on the defect rate is 0.01%.

As can be seen, high confidence in high levels of quality requires large numbers of tests. Can the number be reduced? No, but the testing time can by partitioning the test set and running it on multiple machines in parallel. It is not even necessary that the test set be determined in advance and then allocated among machines; as long as each machine independently samples randomly from the same sample space (using a different seed, of course), the result is the same. The only drawback to this technique is that it relies even more heavily than usual on the "statelessness" assumption of software testing: that the correctness of a computation is independent of the (legal) state of the machine. If a fault only triggers a failure after a certain very long sequence of computations has occurred, then it will only be found on a machine that runs

the full length of that sequence. However, given that such "phase of the moon" failures are always hard to find, this is not a serious drawback for such a time-saving technique. Once again it can be seen that the existence of a random sampling mechanism is the key to stochastic testing.

We can see then, that acceptance sampling is a simple yet very powerful method for estimating atemporal quality levels in software. As long as we have a method for generating a random sample of the input space, we can use this method to produce quantitative estimates at whatever level of accuracy we desire.

6.0 The Use of Acceptance Sampling

The theoretical effectiveness of stochastic testing is not surprising given that it has been used for decades in other domains. What remains to be seen is whether it can be practically applied to the domain of software, to answer the question of "How much testing is enough?"

6.1 Some Objections to Acceptance Sampling

An obvious objection to stochastic testing in general is that it relies critically on the correct random sampling of the space of possible inputs. In theory this is easy to specify, but in practice it is much harder: the set of inputs may be hard to define, and the frequency distribution unknown. The answer to this is two-fold.

First, testers are already specifying inputs, often by automation; stochastic testing simply ensures the validity of testing by being even more systematic and automated. The use of a grammar-driven generator is neither a new nor difficult-to-implement idea, and is very powerful.

Second, testers are already specifying input distributions implicitly, and stochastic testing simply encourages them to be even more explicit and flexible about it. A weighted grammar-driven generator easily allows testing several different input distributions, and allows testers to switch between logical/ uniform and operational profiles very easily. In short, while stochastic testing, since it is based on sampling, produces estimates that are only valid under the given sample model, exactly the same situation occurs in any sort of reliability testing, where the reliability estimate is valid only under certain assumptions, such as operating temperature. The use of statistical methods allows both a precise statement of the generalizability of the results and a quantitative estimate of their precision. Neither of these can ever be provided by haphazard testing.

Another obvious objection to stochastic testing is that it requires the automated generation of large numbers of test cases but simply assumes the presence of an oracle for evaluating the results, whereas in practice such evaluation is often not trivial. However, this is not a criticism of stochastic testing *per se*, since it is true of *all* forms of software testing. Moreover, only stochastic testing provides a quantitative method for incorporating the effect of a fallible oracle, by adjusting the parameters of the OC curve (see Schilling [1982] for details).

A final objection to acceptance sampling in particular is that it appears to ignore the relationship between faults and failures, by assuming that there is a 1-1 relationship between the two. Friedman and Voas [1995] have discussed the notion of "fault size", the number of failures that are due to a single fault, and have created methods to estimate average and minimum fault size. For temporal estimates of software reliability, especially reliability growth models, this may be useful because a large average fault size implies a more

rapid growth in reliability than that of a 1-1 model. However, for atemporal estimates of software reliability, fault size is not relevant, since acceptance sampling is not a reliability growth method, but a one-time conformance assessment: it is only whether or not a failure occurs that is important.

6.2 The Practical Application of Acceptance Sampling

Even if we admit the theoretical advantages of acceptance sampling, there is still the question of how it should be carried out in the process of developing a complex system. Before we consider that, it needs to be emphasized that acceptance sampling by itself is not, and cannot be, sufficient for ensuring software reliability. There are three reasons for this.

First, purely functional testing (the usual application of stochastic testing) by definition ignores critical paths and failure analysis, two of the cornerstones of reliability engineering. Fault-tree analysis and Failure Modes, Effects, and Criticality Analysis (FMECA)[1] must be done to ensure that critical components are correct, and that the consequences of failure conditions are understood. These are better done through exhaustive structural testing, not to mention reviews and analysis of the design and its source code implementation.

Second, experience has shown that the way to maximize system reliability is not just to minimize failure rates but to provide some form of "fault-tolerance" (e.g., redundancy) for when failures do occur. Indeed, good fault-tolerant design can even allow (relatively) higher failure rates than would be possible without fault-tolerance. Thus it is usually more cost-effective, both for the customer and the software developer, to expend resources on making the system fault-tolerant rather than spending thousands of hours testing for the last defect.

Finally, and perhaps most important of all, quality cannot be simply tested into a product, no matter how good the testing procedure.

Thus the best approach to developing reliable software entails the use of

- fault-tolerant design
- failure analysis
- structural testing (typically in conjunction with failure analysis)
- atemporal stochastic testing (acceptance sampling)
- temporal stochastic testing (reliability measurement)

Given that temporal stochastic testing (reliability measurement) provides more information than atemporal stochastic testing, one might ask why acceptance sampling should be used at all. There are two reasons:

First, acceptance sampling is both theoretically simpler, and easier in practice to carry out, than reliability measurement. No assumptions need be made about failure rates, and only pass/fail information need be collected. Differing execution rates are not a concern in parallel testing.

Second, because of its conceptual and practical simplicity, acceptance sampling is the most straightforward way of demonstrating reliability conformance: the customer need only stipulate the operational profile and the number of tests to be passed.

1. See Chapter 13 of Ireson & Coombs [1988] for an introduction.

The strength of acceptance sampling is thus its use as a *final conformance assessment*, rather than as a developmental tool, since it does not allow for measurement of reliability growth.

We come thus to the basic question "How much testing is enough?"

For acceptance sampling, the answer is: when no more than c failures are observed in a randomly selected sample of n tests, given producer's and consumer's risks of β and α, respectively, and a predetermined Acceptable Quality Level. The tester is free to set the quality parameters to whatever levels are desired, and then will know precisely how well the software meets them.

7.0 References

Duncan, A. (1974), *Quality Control and Industrial Statistics*, Fourth Edition, Irwin, Homewood, IL.

Friedman, J., and J. Voas (1995), *Software Assessment: Reliability, Safety, Testability*, Wiley, New York, NY.

Hahn, G. (1974), "Minimum Size Sampling Plans," *Journal of Quality Technology*, *6*, 3, 121-127.

IEEE (1983), *IEEE Standard Glossary of Software Engineering Terminology*, ANSI/IEEE Std 729-1983.

Ireson, W., and C. Coombs, eds. (1988), *Handbook of Reliability Engineering and Management*, McGraw-Hill, New York, NY.

Maurer, P. (1990), "Generating Test Data with Enhanced Context-Free Grammars," *IEEE Software*, *8*, 4, 50-55.

Schilling, E. (1982), *Acceptance Sampling in Quality Control*, Dekker, New York, NY.

A New Approach to Managing Project Requirements and System Testing

Author:

Leslie Allen Little

c/o Aztek Engineering

2477 55th St. Suite 202

Boulder CO 80301

Keywords/Phrases: System Test, Requirements, Traceability, RDBMS, Hardware, Software

Biographical Sketch:

Leslie Allen Little is the senior Quality Assurance Engineer at Aztek Engineering. His primary areas of expertise are Software Quality Assurance, Telecommunications and RDBMS. Mr. Little holds an AAS in Computer Science, a double major (BSBA) in Philosophy and Computer Information Systems from the University of Southern Colorado and a MS in Telecommunications from the University of Colorado at Boulder.

Abstract

System verification of a product with respect to requirements was the primary motivating force for creating and implementing ***RTC*** (Requirements & Test Case Database) at Aztek Engineering. RTC primarily interrelates requirements with test cases. All hardware and software requirements for the project are stored in the database and each requirement is linked to one or more test cases.

RCP (Requirements Change Proposal Process) which facilitates the need to manage and increase awareness of requirements over time was implemented as a complement to RTC. Together, RCP and RTC are the primary processes comprising RMP (Requirements Management Process).

Although the reasons for creating RMP included the need for a more automated tracking system, increased system test productivity, verification of test coverage for all requirements, and consistent management of requirements changes, RMP has resulted in a far broader set of changes. Prime examples are the creation of a greater awareness of actual project requirements throughout the development cycles and the heightened awareness by both the management and marketing communities to explicitly specify requirements as part of the design process.

Like any process, RMP continues to be refined. These tools have progressed from a simple idea of seeking a better way to manage and improve the efficiency of the system testing process into an integral part of the overall product life cycle. RMP has greatly contributed to the overall success of the product.

Introduction

In the fall of 1993, as part of a new product development, this author was faced with the monumental task of organizing and actualizing a test department with limited resources for the purposes of testing a telecommunications switch (essentially a Private Branch Exchange). The project was expected to involve up to 40 personnel, introduce a unique packaging system with newly-designed electronics, and create 150,000 to 200,000 non-commented source code lines.

After some initial research and given the limited staffing considerations for the system test department[1], it was quite apparent that a different approach to testing the system would be necessary to meet project demands. The old school of creating system test documents, wading through documents, maintaining these documents, etc., would not be sufficient. Given the author's background in applications and database development, creating a database of requirements and then linking those requirements to test cases appeared to be the ideal solution. This approach was proposed to management and a preliminary nod of approval was given.

The first task was understanding, as best as possible, what the requirements of the system would be. At this early stage of RMP development, RTC was the only system being developed. A general set of requirements, primarily from the system test perspective, were constructed for RTC. Once developed, the database schema and structure were created. Lastly, the laborious task of transcribing the essential elements of seven existing and rather lengthy requirements documents began. This process was made difficult by the need to reduce some 200+ pages of existing requirements into a consistent, yet complete and non-redundant, set of requirements in a format acceptable to database entry and manipulation.

In retrospect, a number of mistakes were made during this process, with the primary mistake being that too many requirements, many of which were non-verifiable and ambiguous, were not sufficiently reviewed and updated. In retrospect, these requirements should have been either corrected or removed. This deficiency has led to a continual need to *clean up* old requirements. Another significant oversight concerning the construction and use of the RTC was the lack of recognition for the need of a strong, structured process to control requirement changes, i.e., the addition of RCP. Without such a process, requirements *creep*[2] begins to occur providing yet another source of requirement corruption. As a result, RCP was created and is currently serving the needs of the project.

Requirements Management Process (RMP)

Managing requirements is a difficult task. Without sufficient tools to manage requirements, the majority of all projects are doomed to failures and delays. The implementation of RMP at Aztek Engineering has improved the quality of the project. The incidence of field failures is comparable to or lower than other similar projects. The personal resources devoted to system test, relative to the size of the project, is low comparative to this author's knowledge of similar operations. In summary, Aztek is able to produce more for less with higher quality.

A figurative view of RMP is presented in *Figure 1 Requirements Management Process*. RMP consists of RCP, RTC, and the tools and reports that support these two processes. As shown, interactions occur between processes (depicted by ovals) and resources (depicted by open-ended rectangles). RTC provides outputs and receives inputs from the majority of the human resource organizations within Aztek as does RCP. RCP and RTC interact directly with each other.

[1] The staffing would initially be 1 person full time; gradually growing to the current equivalent of 2 and 1/2 persons full time.

[2] Requirements *creep* refers to the gradual changing of requirements that takes place after their initial review. This takes the form of an ever expanding set of requirements as the development community adds their interpretation to the requirement, then system test asks for *corrections* to the product based on their interpretation of the requirements, etc.

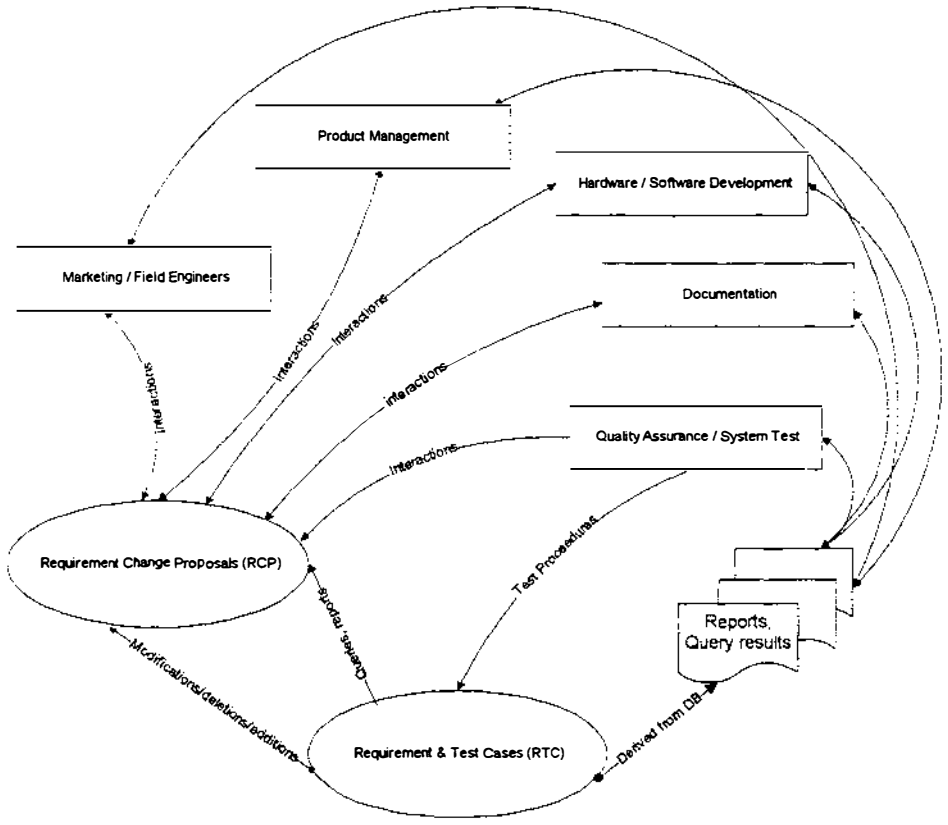


Figure 1 Requirements Management Process

Requirements Change Proposal Process (RCP)

RCP essentially serves as a requirements change management system. It is a separate process from RTC, embodied with its own database and tool set. RCP is vaguely modeled on the inquiry cycle[1] where changes are enacted via a requirements discussion process. RCPs contributions to RMP are:

1. Provides a mechanism for discussing and reviewing the request by users to add, delete, and modify requirements.
2. Provides a centralized and accessible repository for the discussions and decisions associated with the requests to add, delete, and modify requirements.
3. Provides a gateway that requirements additions, deletions, and modifications must pass through before becoming official. This gateway is crucial for a number of reasons of which consistency and applicability are crucial.

Discussing and Reviewing Requirements

When you consider the needs of any given project with respect to requirements, there are the obvious ones concerning creation, modification, and deletion of requirements both initially and over time. Without expounding on the minute details of this process, requirement modifications, deletions, and additions are created primarily in one of two ways--singularly or in related groups.

Singular changes are usually due to events such as a test or development engineer reviewing requirements and noticing that the wording of a requirement is not quite right, or that it implies something it shouldn't, etc. These change requests are simple, can be dealt with quickly and usually involve a small audience for review purposes.

Related groups of requirement change proposals are usually associated with events such as new feature development. These changes are usually much more involved and generally involve a larger group of review members.

RCP provides a means for both types of requirement proposals to be entered into a database electronically. Appropriate review personnel are sent proposal notifications (via email) and an expected completion date. Discussions concerning the validity, applicability, clarity, etc. are generated electronically amongst interested reviewers through the database. Eventually, each proposal is either discarded or passed on to a formal review committee which essentially asks the question of whether the proposed change affects the form, fit, or function of the existing product. If so, the extent of the impact upon the existing system is examined and used to justify the need to reject, request further modifications, or to accept the proposed requirement change(s). At this meeting, the requirements controller reviews and possibly adds appropriate information that may be missing such as filter tags, target markets, etc[3]. Additionally the controller makes sure that the requirement is a *good*[4] requirement. Once completed, and if accepted, the proposal is approved and RTC is updated.

Repository for Requirements Proposal Requests

Another problem that RCP addresses is to dilute the effects of a project slowly loosing the movers and shakers who understand the reasoning processes behind the project (sometimes referred to as *brain drain*). In many projects, the original thought processes that make up the framework for decisions which in turn define the requirements of the product are usually swept away with the engineers who originally considered those thoughts. Over time, the engineers move on to other work and a new set of engineers replace them. Not having the benefit of having toiled over the tortuous details of each requirement and design decision, unless that information is stored somewhere, it essentially has been lost. Even for those original designers that do stay, remembering or recalling the details that went into a decision are sometime forgotten. As can be imagined, either of these cases can lead to subsequent decisions that invalidate and directly conflict with prior requirement and design decisions.

RCP limits the extent of this problem by providing an on-line library of discussions and decisions regarding requirements origination, modification and possible deletion. This contextual information is invaluable when, two years after the requirements introduction, a new engineer attempts to understand how and why a requirement originated.

RCP Implementation Details

The actualization of RCP involves the use of a RAZOR® database as the discussion medium. This database provides a forum where all internal and external engineers, marketing personnel, management, etc., have the means to propose, examine, discuss, and prepare requirement additions, deletions, and modifications. A requirements controller monitors all proposals, facilitates the proposals movement through the process and, in general, inspects all finalized proposal requests for completeness, conformance with respect to requirements characterizations, duplication avoidance, etc.

---

[3] As part of the form that the change proposal request author completes, there are a number of fields that directly map to RTC. If any of these fields have not been completed, they are completed so as to allow easy entry into RTC once approved.

[4] Making sure a requirement is a *good* requirements is covered in *Defining Good Requirements*.

The use of RAZOR as the vehicle for discussion centers on its inherent capabilities to track state changes, its commercial availability, ease of use, and low startup and maintenance costs. Possible future revisions to the tools used to facilitate the RCP process would need to combine state change capabilities, archiving, versioning, and threading of discussion paths (similar to the capabilities of a news reader). Similar attempts to merge these types of capabilities have been discussed previously [2][3] where the use of a graphical tool for facilitating the process, gIBIS, was quite interesting.

Defining *Good* Requirements

Defining *good* requirements is a difficult task. It presupposes that the person defining the requirement understands what comprises a *good* requirement. Definitions of requirements abound in the literature [4][5][6]. The attributes commonly associated with a *good* requirement include the following:

- Verifiable
- Complete
- Consistent
- Traceable
- Explicit
- Concise

The requirements controller attempts to certify that each requirement change proposal meets these common attributes. Having RTC at the controller's disposal facilitates this task.

Verifiability is directly enhanced since all requirements necessarily are linked with test cases. When a requirement is entered into RTC, a test case place holder must also be entered. Reports are generated that detail all existing requirements that do not have test cases written.

Completeness and consistency are helped in that RTC supplies an easy search mechanism via an editor search, an SQL search, the use of the reqs_filter relationships, etc. Additionally, easily viewing a number of *good* requirements in RTC while constructing new ones is beneficial to the process.

Traceability is the act of being able to trace higher-level requirements to lower-level ones and is directly covered through RTC.

Explicitness is not directly supported although being forced to link test cases directly to requirements gives one second thoughts about the usefulness of a requirement if not explicit.

Conciseness, in general, is not supported through RTC.

All in all, RTC does not guarantee that *good* requirements will be used to populate the database. It simply provides some tools to be used in increasing the likelihood that *good* requirements are added. The processes that an organization uses and their insistence upon new requirements meeting the criteria that has been established are the real reasons *good* or *bad* requirements make their way into a requirements database - not the database itself.

Requirement and Test Case Database (RTC)

RTC has, as its primary users, the system test, documentation, and development organizations. Management, marketing, and field engineers have primarily been receivers of information derived from the database, not active users. Reasons for this are simple. RTC was primarily established with system verification in mind, thus the documentation and system test organizations find it quite useful. Although originally tailored to assist the system test department with tracking and verifying requirements, a UNIX® shell script front end was developed to provide access to predefined reports and queries in an attempt to assist other departmental users needs. In practice, however, users needs are broad and the needs of the various departments are not met using these predefined reports. As a result, system test

personnel ended up performing numerous queries for other organizations and have created upwards of a hundred SQL queries that are occasionally executed for differing user needs. Management, marketing and field engineers generally are the recipients of particular reports that are used for determining status such as testing progress and percentages, requirements completion percentages, etc. Other report types include release notes for general releases, feature requirements, acceptance tests to be used by field engineers and marketing teams, etc.

Although the RTC process has not been perfected, keeping all requirement and testing information related to a project up-to-date and accurate can provide an enormous resource to all users of the system, and as indicated, the user base can be quite diversified. The problem is that keeping the requirement and testing information up to date is itself an enormous job. The remainder of the paper concentrates on these and other aspects of RTC.

RTC Internals

What follows is the hardware/software platform for RTC and a discussion of the tables that comprise the database.

Hardware/Software Platform for RTC

RTC was implemented using an Informix® RDMS engine (version 4.12.UE1) running on a Sparc® 10/40. RTC consists of

- Informix relational database
- UNIX shell and TCL scripts
- Informix SQL query scripts
- Informix forms for data entry
- Informix ACE reports

Additionally, text files of the requirements and test case descriptions are retrieved from the database nightly and placed on a shareable drive providing convenient access to end users via editors.

RTC Table Relationships

RTC currently consist of

- five requirements related tables
- six test case related tables
- one join table to map the requirements stored in the database to the test cases stored in the same database
- numerous validation tables

In the past, there were two additional requirement related tables, the first was used to receive data inputs via links to an external Microsoft® Project scheduling table[5] while the second was a historical dumping ground of key requirements fields that served as a crude versioning tool[6]. Additionally, links to the project problem tracking tool exist in one of the test case related tables. The project problem tracking tool is also contained in an Informix database providing advantageous benefits when relating

[5] The links to scheduling table turned out to be more work than they were worth given the startup stage of the project during the time of it's usage. During the more mature stages of a project, such links could provide useful updates to schedule dates that affect component completion dates currently embedded in the database.

[6] This table is no longer a part of RTC since RCP serves as a more sophisticated collection of historical events that lead to changes in RTC.

the two databases. *Figure 2 High-Level Relationship Among RTC Tables* gives a graphical view of these relationships. Note that all requirement tables have a primary key which is common--the requirement number while similarly, all test case tables have a primary key which is common--the test case number. The jump table, reqs_tests, links the two sets of related tables allowing easy traversal between any requirement table and any test case table.

Though difficult to perform, keeping tables simple, constructing meaningful relationships among tables and following the rules of normalization lead to an efficient database design. These principals have been used in constructing the RTC database. The following sections examine each of the tables that comprise the database and their usage.

Description of reqs_tests Table

The reqs_tests table binds the requirements side of the RTC database to the test case side of the same database.

Description of req_filtersTable

The req_filters table is a general purpose filtering table that contains filters based on any general grouping that is useful to the end user. Typical characteristicjal grouping are by

- Feature
- Verification responsibility
- Hardware/software separation[7]

The drawback to such a table is reliability over time. As more requirements are added and modified, close inspection of each requirement with respect to all possible groupings becomes important. Thus the more groupings, the greater the potential for inconsistent data sets. The requirements proposal process as explained in the section, *Requirements Change Proposal Process,* attempts to avoid this pitfall by having a requirements controller[8] who is involved in all requirement modifications.

Description of req_description Table

The req_description table is primarily concerned with the definition of the requirement description. Other related fields contained in this table include an internal/external flag, a free-form specification text field, and a free-form notes field.

The internal/external flag is used to distinguish between requirements that are externally viewable by Aztek customers[9] versus those requirements that are not externally publicized. For the most part, the external requirements are usually top-level requirements while most of the internal requirements are derived.

The specifications field is used to relate standards documents, internal documents, etc. to a given requirement description while the notes field is used to record important considerations associated with the requirement description such as origin, contextual information not contained in the requirement, etc. To a large part, RCP has negated the need for this field.

---

[7] It is sometimes useful to examine all hardware or software requirements, not both. Upon entry of a requirement, each new requirement is tagged as either software, hardware, or both.

[8] In reality, there are two controllers. The primary controller performs the day to day activities while the backup controller is there if called upon. The backup controller keeps in touch with the activities of the primary but not nearly to the level of detail that the primary controller participates.

[9] Externally viewable implies that the requirements are part of external documentation which the customer receives. Primarily this external documentation is the high level external requirements specification for the product. In rare cases, external requirements might also be defined sales literature, etc.

Figure 2 High-Level Relationship Among RTC Tables

Description of req_derivations Table

Requirements, as they are defined in RTC, are associated with two possible derivable states; either they are top-level system requirements that are not derived from other requirements or they are derived requirements. This table captures that information and is used to record requirement linkages,

especially when identifying *suspect*[10] requirements due to modification of existing requirements. Test cases are linked to requirements and thus the test engineer can use the *suspect* requirements list to determine *suspect* tests that require regression testing.

Description of req_components Table

All requirements are fulfilled by one or more components of the system. The component may be documentation components, software components, hardware components such as circuit boards, physical housing units, environmental packages, etc., or possibly some other component that is unique to the project being examined.

Associated with the component are completion dates. It can be, and sometimes is, the case that two or more components have differing beginning and completion dates for the same requirement. This is often the case in software where capabilities are implemented in one component prior to complementary capabilities in some other component. In these cases, the latest completion date of a component dictates the completion of a requirement.

In practice, the completion dates for individual components with respect to their associated requirements, has not been used. Given the startup nature of the project, the consistent churn with respect to the values contained in these fields made their benefit less than their maintenance costs. As a result, the completion date for a requirement has been maintained on a requirement basis in the req_applicability table.

Description of req_applicability Table

All requirements have applicability traits--they are expected to be met in some finite time period and they are applicable to some target market. A requirement can be applicable to only one target market or to many target markets. The req_applicability table captures these two important characteristics concerning requirements.

Description of test_components Table

The test_components table is the test case equivalent to the req_components table for requirements. Just as one or more components are responsible for the implementation of requirements, one or more components are tested with each and every associated test in the test case portion of the RTC database. Imagine a requirement that has three components slated for its fulfillment, each of the three fulfilling a portion of the requirement that needs to be fulfilled as a whole. Now image four test cases slated to test the requirement. The first three test component A, while the fourth test components A, B, and C. This database structure permits the modeling of such an arrangement.

Description of test_equipment Table

Practically all tests require some amount of equipment, even though a large portion of that equipment may be the system under test. The test_equipment table enables you to specify all equipment needed for a given test. It proves it's usefulness by later providing a simple ordering for system test personnel to use when trying to minimize set-up time for testing.

Description of test_descriptions Table

The test_descriptions records, unlike the req_descriptions records contains only two fields; the test case number and the description of the test case. Tests are always written in action/verify[11] pairs. The

[10] A *suspect* requirement is defined as a requirement whose previously tested validity has been called into question.

[11] An action/verify pair implies that the test case contains the actions that are needed in order to verify some requirement or partial requirement. They typically are worded in the form "Do x, y, z and verify q, r, s."

creation of multiple test cases is always preferable to the grouping of a large number of action/verify pairs into one test case. The reasons for this are obvious. If one action/verify pair fails while the other three action/verify tests pass, then how are the results recorded? They should be recorded as three successes and one failure. Unless the they are split into separate test cases, this becomes a burden to perform.

Description of test_markets Table

Test cases are sometimes applicable to specific target markets, particularly when the test case type is *acceptance*. In such cases, a test case may be applicable to only one target market. An example might be that *target market A* wants to test *requirement R* in a particular manner while *target market B* wants to test the same *requirement R* in a differing way. Both tests are valid and both test the same requirement, yet they are different. The test_markets table provides the ability to designate test cases on a per target basis and is very similar in principal to the market field in the req_applicability table.

Description of test_types Table

Test types refers to the somewhat artificial but useful practice of designating tests as being of differing types. Common test types are *unit*, *system*, *integration*, *regression* and so forth. This is useful for various user communities accessing and utilizing RTC. For example, development engineers, when correcting problems for a component, can retrieve and review regression tests for that component that are of the type *unit* in order to re-test the appropriate unit level tests. System test utilizes this capability frequently when determining tests to execute again once change has occurred. A typical scenario might be to query RTC for all system tests related to the administration of a particular feature. Such a query would probably utilize the test_types, test_cases, reqs_tests, req_filters, and req_components tables.

Description of test_executions Table

All verification testing is recorded uniformly in the test_executions table. This table provides fields for such things as the test case number tested, status of test, name of the tester, what platform it was executed on, related problem ticket number, and a general free-form text area to record notes. All reports concerning testing status are derived from this table. SQL queries can examine individual test case status, the number of currently blocked[12] test cases, etc.

Test Case Creation and Relations

When requirements are entered into RTC, a reminder test record is automatically created. This reminder record simply states as its description that a test case needs to be constructed. This allows queries to identify requirements that are deficient with respect to test case coverage while satisfying the needs of referential integrity constraints. Once test cases are constructed, target markets are identified, the test type is determined, etc. The actual execution of test cases is recorded in the test_executions table which is a historical grouping of all test case executions. Reports can be created identifying problem areas by component, outstanding failures, blockages, etc. A link to the problem tracking table allows test cases to be linked to outstanding problem tickets and thus when tickets are resolved, the test cases can be revisited.

RTC Advantages

There are a number of advantages for using RTC versus the traditional paper-based requirements systems. At a minimum, RTC

[12] Blocked test cases are defined as test cases that cannot be tested due to some external blockage, for example, an outstanding problem ticket that prohibits a test case from being executed.

- Increases a system test engineer's productivity
- Provides a means to measure test coverage
- Indicates *suspect* requirements and test cases given changes to the product
- Easily assigns responsibility for requirements fulfillment to components, and thus their rightful owners
- All requirements and test cases pertaining to a functional component are easily identifiable
- Heightened awareness of requirements by all participants concerned with requirements fulfillment
- Provides a convenient manner of viewing related yet distinct requirements
- Supports multiple target markets for requirements
- Provides on-line access to system test personnel, development engineers, product management, and marketing personnel throughout the product life cycle
- Enhances the ability to determine regression test strategies and to employ them

Each of the above, are significant in their own right. More importantly, RTC has proved to be a significant aid in performing, reporting and verifying test coverage, and thus providing management with a measured degree of confidence that the product is ready to be released.

For developers, RTC allows reasonably easy access to related requirements. For marketing professionals, RTC has the capability to provide all existing and upcoming requirements, when they were fulfilled and when they are expected to be. Marketing professionals won't be caught making wild promises if they keep abreast of the product's development.

For product managers, RTC provides quick reporting of current status of the project. Requirements can be examined for fulfillment and test reports are used to build confidence in the product-or to point out trouble spots before it's too late.

Future Enhancements for RTC

The next internal release of RTC is expected for the second quarter of 1997. That release will incorporate a number of the items listed in the following paragraphs. A lot of work is necessary to make this prototype into a product that applies to a broader range of applications and simplifies its use.

Intertwining Requirements Documents with RTC

The seamless merging of paper requirements documents written in Microsoft Word with actual requirements descriptions in RTC are being explored. Customers generally prefer paper documents, written in their native language that provide explanations and examples for each requirement. The seamless integration of requirement documents with the actual underlying database of requirements will resolve the continuing problems of keeping the documents and database synchronized, providing contextual information associated with each requirement, allowing easy navigation between documents and database, etc.

Microsoft's Object Linking and Embedding offers the opportunity to physically link separate applications with a minimum of effort. In this way, an Informix database (via ODBC drivers) could be linked with a Microsoft Word document. Application interfaces can be written to support traversal from one application to the other and provide a reasonably seamless integration of the two.

User Interface

User interfaces can enhance or detract from end user participation. RTC does not currently have a user-friendly interface, it utilizes a character-based interface. This is one of the primary drawbacks of the

system. As is the case in many organizations, sometimes the difference between what one would like to do and what one can afford to do is quite apparent. This has been the case with RTC. The initial task to create a means of entering and retrieving data was realized with a minimum amount of effort and expense. The result is a functional system for data input and retrieval, limited by character-based interfaces and the user's ability to understand SQL query and reporting syntax. As can be imagined, this is not the average user's forte.

Plans are in progress to move from the current character-based Informix forms to a GUI. A GUI based interface is deemed critical to the product's future. Informix's NewEra and Microsoft's Visual Basic are currently under evaluation to determine which interface supplies the best application language given the proposed requirements for RTC. A general purpose report writer will be incorporated to allow easy access to queries and reports for all users of the system.

Binary Large Object Field Support

The initial version of RTC does not support BLOB (Binary Large Object) fields. The reasoning for this is that the use of Informix forms and ACE reports precluded their usage. Moving to the future, BLOB fields, or their equivalent, should be supported. Irrespective of the more efficient use of space as a result of using BLOBs, the ability to truly intertwine requirements in a database with documents describing those requirements could reap significant benefits. Imagine a table or picture that itself was a requirement. The simplest way to represent this is by inserting the item into the database, not rewording the requirement.

ODBC Connectivity

For a tool such as RTC to be of use to numerous organizations, it must support connectivity to a number of databases. The next version of RTC is slated to support connectivity to multiple databases through ODBC connectivity.

Versioning

A critical component of requirements tracking is to be able to determine the state of the system at any given time. The versioning currently built into RCP doesn't provide this capability. It does provide the ability to trace all discussions, state changes, etc., to requirement change proposals, but it does not easily provide the ability to restore RCP to a given point in time. Restoring RTC to a particular point in time is also not trivial.

Providing a Tighter Coupling Between RTC and RCP

Though not investigated rigorously at this time, there could be arguments made both for and against the combining of the RTC and RCP processes into one set of integrated tools. There are both similarities and differences in their capabilities, areas of responsibility, and function. This will need to be evaluated. Future considerations with respect to how to achieve the tighter coupling are under review.

User-Defined Attribute Fields

RTC consists of a number of tables, each with a number of fields. The fields are pre-defined as are their relationships. Not every organization that utilizes RTC's capabilities will find the fields, as currently named, useful. Odds are, not all the relationships as they currently exist will probably be useful to all organizations. What is needed, is a simple way of allowing the administrator of the system to define the relationships, field names, etc. upon installation. Future releases of the product will incorporate such goals.

User Definable Permission Controls

Permissions controls are another feature that would add great value to the system. Given the potentially wide audience that can have access to RTC, keeping certain users out of sensitive areas is sometimes necessary.

Related Work

Early attempts to carry out an RTC type system was that of Arts[7]. Arts was developed as a bookkeeping program that operated on a database consisting of system requirements and their attributes, much like RTC. It provided upward and downward traceability in a hierarchical structure as well as database management operations on all of the contained attributes. Other similar examples are Requirements Tracer and RqT[8].

A more current project, TOOR[9], is a rather impressive system links requirements to design documents, specifications, code and other artifacts through user-definable relations. It however, doesn't address the test linkage aspects of RTC.

As for the RCP process and what is known in the literature as the *early* requirements phase, a considerable number of attempts at structuring and automating requirements espousal and collection have taken place. A set of document based memorandum macros referred to as RADIX [6] is one such attempt while the Requirements Language Processor [5] describes another. RETH is an interesting attempt to provide what is described as *semiformal hypertext* representations of requirements[10]. It is an interesting idea that attempts to blend the formal versus informal approaches that currently exist.

None of the current literature, however, appears to adequately address the issues that RTC attempts to resolve, namely the interrelation of requirements and test cases to facilitate the design, development and system test processes in a production environment.

Conclusions

Although much progress has been made in the area of requirements management with the introduction of RMP, more advances are needed. Many times the germination of an idea is the simple part while actualization is hard. RMP and its component parts, RCP and RTC, follow this model nicely. In its present form, RMP has proved most useful to the system test, documentation, and development communities. It has the potential to become far more pervasive in these important areas and to extend its usefulness from occasional use to active and regular usage in the areas of development, project management, and marketing disciplines.

Acknowledgments

I would like to extend my heartfelt thanks and appreciation to my editor, Bernice Volinsky, whose enduring help and ability to always smile has made my scribbling perfectly readable. Thanks is also given to Aztek Engineering for providing the opportunity to perform this needed research and development. Lastly, but certainly not least, special thanks is given to my loving wife, Nadia, whose tireless search for perfection and happiness has affected my life so greatly.

References

[1] Colin Potts, Kenji Takahashi and Annie I. Anton, "Inquiry-Based Requirement Analysis," IEEE Software, March 1988.

[2] Colin Potts and Glenn Bruns, "Recording the Reasons for Design Decisions," IEEE Journal, 1988.

[3] Jeff Conklin and Michael L. Begeman, "gIBIS: A Hypertext Tool for Exploratory Policy Discussion," ACM Transactions on Office Information Systems, Vol. 6, No. 4, October 1988.

[4] Alan M. Davis, "Automating The Requirements Phase: Benefits to Later Phases of the Software Life-Cycle," IEEE Computer Society's Fourth International Computer Software & Applications Conference, Chicago Illinois, pp 42-48 (October 27-31, 1980).

[5] Herb Krasner, "Requirements Dynamics in Large Software Projects, a Perspective on New Directions in the Software Engineering Process," Proceeding of the IFIP 11th World Computer Congress - Information Processing, San Francisco, pp 211-216 (1989).

[6] Weider D. Yu, "Verifying Software Requirements: A Requirement Tracing Methodology and Its Software Tool ---RADIX," IEEE Journal on Selected Areas in Communications, Vol. 12, No. 2, February 1994.

[7] Merlin Dorfman and Richard Flynn, "Arts -- An Automated Requirements Traceabilty System," The Journal of Systems and Software, Volume 4, Number 1, April 1984.

[8] Jag Sodhi, Software Requirements Analysis and Specifications, McGraw-Hill, Inc., 1992.

[9] Francisco A.C. Pinheiro and Joesph A. Goguen, "An Object-Oriented Tool for Tracing Requirements," IEEE Software, March 1996.

[10] Hermann Kaindl, "The Missing Link in Requirements Engineering," ACM Software Engineering Notes, Volume 18, Number 2, April 1993.

Automated Quality Analysis Of Natural Language Requirement Specifications

William M. Wilson
Software Assurance Technology Center/GSFC
Bld 6 Code 300.1
Greenbelt, MD 20771 USA
+1 301 286 0102
William.M.Wilson@gsfc.nasa.gov

Linda H. Rosenberg, Ph.D
Unisys Federal Systems/GSFC
Bld 6 Code 300.1
Greenbelt, MD 20771 USA
+1 301 286 0087
Linda.Rosenberg@gsfc.nasa.gov

Lawrence E. Hyatt
NASA Goddard Space Flight Center
Bld 6 Code 302
Greenbelt, MD 20771 USA
+1 301 286 7475
Larry.Hyatt@gsfc.nasa.gov

Abstract

The Goddard Space Flight Center's (GSFC) Software Assurance Technology Center (SATC) has developed an early life cycle tool for assessing requirements that are specified in natural language. This paper describes the development and experimental use of the Automated Requirements Measurement (ARM) tool. The ARM tool searches the requirements document for terms the SATC has identified as quality indicators. Reports produced by the tool are used to identify specification statements and structural areas of the requirements document that need to be improved.

1. Introduction

The Software Assurance Technology Center (SATC) is part of the Office of Mission Assurance of the Goddard Space Flight Center (GSFC). The SATC's mission is to assist National Aeronautics and Space Administration (NASA) projects to improve the quality of software that they acquire or develop. The SATC's efforts are currently focused on the development and use of metric methodologies and tools that identify and assess risks associated with software performance and scheduled delivery. It is generally accepted that the earlier in the life cycle that potential risks are identified the easier it is to eliminate or manage the risk inducing conditions [1].

Despite the significant advantages attributed to the use of formal specification languages, their use has not become common practice. Because requirements that the acquirer expects the developer to contractually satisfy must be understood by both parties, specifications are most often written in natural language. The use of natural language to prescribe complex, dynamic systems has at least three severe problems: ambiguity, inaccuracy and inconsistency [11]. Many words and phrases have dual meanings which can be altered by the context in which they are used. For example, Webster's New World Dictionary identifies three variations in meaning for the word "align", seventeen for "measure", and four for the word "model". Weak sentence structure can also produce ambiguous statements. "Twenty seconds prior to engine shutdown anomalies shall be ignored." could result in at least three different implementations. Using words such as "large", "rapid", and "many" produces inaccurate requirement specifications. Even though the words "error", "fault", and "failure" have been precisely defined by the Institute of Electrical and Electronics Engineers (IEEE) [5] they are frequently used incorrectly. Defining a large, multi-dimensional capability within the limitations imposed by the two dimensional structure of a document can obscure the relationships between individual groups of requirements.

The importance of correctly documenting requirements has caused the software industry to produce a significant number of aids [3] to the creation and management of the requirements specification documents and individual specifications statements. Very few of these aids assist in evaluating the quality of the requirements document or the individual specification statements. This situation has motivated the SATC to develop a tool to provide metrics that NASA project managers can use to assess the quality of their requirements specification documents and to identify risks that poorly specified requirements will introduce into their project. It must be emphasized that the tool does not attempt to assess the correctness of the requirements specified. It assesses the structure of the requirements document and individual specification statements and the vocabulary used to state the requirements.

2. Background

The SATC study was initiated by compiling a list of quality *attributes* that requirements specifications are expected to exhibit [6] [11]. The next step was to list of those aspects of a requirements specification that can be objectively and quantitatively measured. The two lists were analyzed to identify relationships between what can be measured and the desired quality attributes. This analysis resulted in the identification of categories and individual items that are primitive *indicators* of the specification's quality and can be detected and counted by using the document's text file.

Concurrent with development and analysis of the attribute and indicator lists, forty-six requirement specifications were acquired from a broad cross section of NASA projects. These documents were converted into ASCII text files. These files were used to develop a database containing the words and phrases used in the set of document and the number of times that they occurred. This database of basic words was used to refine the list of primitive indicators.

Using the refined list of primitive indicators, an initial version of the Automated Requirements Measurement (ARM) software was developed for scanning the specification files. ARM was used to subject each specification file to a full text scan for occurrences of each of the quality primitives. The occurrence of primitives within each file were totaled and reported individually and by category. Correlation between all totals were examined for significant relationships. Files that exhibited anomalous data and off-norm counts were examined to determine the source of these aberrations. A tentative assessments of each requirements document's quality was made based on this analysis and examinations. The source documents are currently being independently reviewed to provide a basis of comparison with the conclusions arrived at using the ARM's reports.

While the source documents are being independently reviewed, the prototype ARM software is being used to aid selected NASA projects to strengthen their requirement specifications. The results of the engineering assessments and feedback from the selected NASA projects will be used to improve ARM's assessment processes and its user interface.

3. Specification Quality Attributes

Desirable characteristics for requirements specifications are identified [2] [6] as:

- Complete
- Correct
- Ranked
- Unambiguous
- Verifiable
- Consistent
- Modifiable
- Traceable
- Understandable

As a practical matter, it is generally accepted that requirements specifications should also be **Validatable** and **Testable**. These eleven characteristics are not independent. For example, McCall's quality model [7] identifies tractability, completeness, and consistency as being factors which contribute to correctness. Also, the ISO 9126 software quality model [7] gives stability, modifiability (changeability), and testability as factors contributing to maintainability. A specification, obviously, cannot be correct if it is incomplete or inconsistent. It would also be difficult to validate a requirement specification that could not be understood.

Most, if not all, of these quality attributes are subjective. A conclusive assessment of a requirements specification's appropriateness requires review and analysis by technical and operational experts in the domain addressed by the requirements. Several quality attributes, however, can be linked to primitive indicators that provide some evidence that the desired attributes are present or absent. These primitives are alluded to in the attribute definitions below.

3.1. Complete

A complete requirements specification must precisely define all the real world situations that will be encountered and the capability's responses to them [11]. It must not include situations that will not be encountered or unnecessary capability features. Since it is difficult to anticipate all real world situations, it is much easier to detect incompleteness than determine completeness. Use of the place holder "TBD" (to be determined) is undeniable evidence that the requirements specification is incomplete. The phrases "as a minimum" and "not limited to", depending on their context, may be more subtle indicators of incompleteness

3.2. Consistent

A consistent specification is one where there is no conflict between individual requirement statements that define the behavior of essential capabilities; and specified behavioral properties and constraints do not have an adverse impact on that behavior [11]. Stated another way, capability functions and performance level must be compatible and the required quality features (reliability, safety, security, etc.) must not negate the capability's utility. For example, the only aircraft that is totally safe is one that cannot be started, contains no fuel or other liquids, and is securely tied down.

3.3. Correct

For a requirements specification to be correct it must accurately and precisely identify the individual conditions and limitations of all situations that the desired capability will encounter and it must also define the capability's proper response to those situations [11]. In other words, the specification must define the desired capability's real world operational environment, its interface to that environment and its interaction with that environment. It is the real world aspect of requirements that is the major source of difficulty in achieving specification correctness. The real world environment is not well known for new applications and for mature applications the real world keeps changing. The COBOL problem with the transition from the year 1999 to the year 2000 is an example of the real world moving beyond an application's specified requirements.

3.4. Modifiable

In order for requirements specifications be modifiable, related concerns must be grouped together and unrelated concerns must be separated [6] [10]. This characteristic is exhibited by a logical structuring of the requirements document. As an example:

5. *The XYZ system shall access the ABC, DEF, and GHI databases.*
   - 5.1. *The XYZ system shall permit and restrict access based on application type.*
     - 5.1.1. *Engineering applications shall have read access to all databases and write access to the ABC and DEF databases.*
     - 5.1.2. *Administrative applications shall have read access to the DEF and GHI databases and write access to the GHI database.*

3.5. Ranked

Ranking specification statements according to stability and/or importance is established in the requirements document's organization and structure [6]. The larger and more complex the problem addressed by the requirements specification, the more difficult the task is to design a document that aids rather than inhibits understanding. Ranking specifications according to stability and/or importance can conflict with structuring the document to be modifiable. This conflict often arises when there are safety and security requirements to be specified. Is it best to address them in separate documents, sections, or as subsections under the related functional requirements?

3.6 Testable

In order for a specification to be testable it must be stated in such as manner that pass/fail or quantitative assessment criteria can be derived from the specification itself and/or referenced information [10]. *"The system shall be user friendly."*, can be subjectively interpreted and its implementation will be difficult to test objectively. *"The system's functions shall be activated and terminated by menu selections."*, is a specification that the implementation does or does not satisfy.

3.7 Traceable

Each statement of requirement must be uniquely identified to achieve traceability [10]. Uniqueness is facilitated by the use of a consistent and logical scheme for assigning identification to each specification statement within the requirements document. The example specification statements in paragraph 3.4, above, demonstrate this type of identification. A computer program can easily recognize this type of identification by detecting sentences that begin with strings of numbers separated and terminated with periods. The structure of a requirements document can be assess by relatively simple algorithms if each specification is uniquely identified and expressed as a simple, uncomplicated statement. For the example referred to above there are specifications at three levels. There is one requirement specified at level one, one at level two and level at level three.

3.8. Unambiguous

A statement that specifies a requirement is unambiguous if it can only be interpreted one way [10]. This perhaps, is the most difficult attribute to achieve using natural language. The use of weak phrases such as "*as required*" or poor sentence structure, as demonstrated by the following sentence, will open the specification statement to misunderstandings. *"Users attempting to access the ABC database shall be reminded by a system message that must be acknowledged and page headings on all reports that the data is sensitive and access is limited by their system privileges."*

3.9. Understandable

A requirements specification is understandable if the meaning of each of its statements is easily grasped by all of its readers [11]. This is the primary reason that most specifications are expressed in natural language.

3.10. Validatable

In order to validate a requirements specification each of the individuals and organizations having a vested interest in the system solution must be substantiate that the requirements are true as stated. able To validate a requirements specification all the project participants, managers, engineers and customer representatives, must be able to understand, analyze and accept or approve it [11].

3.11. Verifiable

In order to be verifiable requirement specifications at one level of abstraction must be consistent with those at another level of abstraction [11].

4. Specification Quality Indicators

Although most of the quality attributes of documented requirements are subjective, there are aspects of the documentation which can be measured and are indicators of quality attributes. Size, which is a primitive used in many quality metrics, can be directly measured. The size of an individual specification statement can be measured by the number of words it contains. The size of a requirements document can be easily measured by the number of pages, the number of paragraphs, lines of text, or the number of individual specification statements it contains. The number of unique subjects addressed by specification statements within the requirements document can also be counted with relative ease. This count is an indication of the scope of the requirements encompassed by the document. The breadth and hierarchical depth encompassed by the document's specification statements can be measured using the document's internal identification scheme. These measures provide clues to the document's organization and depth of detail. The number of specification statements at each level of the document's structure can also be counted. These counts provide an indication as to how the specification statements are organized and the level of detail to which requirements are specified. It is also possible to count the occurrence of specific words and phrases that signal that specification statements are weak or strong.

4.1. Categories

Nine categories of quality indicators for requirement documents and specification statements were established based on a representative set of NASA requirements documents selected from the SATC's library. Individual indicators were identified by finding frequently used words, phrases, and structures of the selected documents that were related to quality attributes and could be easily identified and counted by a computer program. These individual indicators were grouped according to their indicative characteristics. The resulting categories fall into two classes. Those related to the examination of individual specification statements and those related to the total requirements document. The categories related to individual specification statements are:

- Imperatives
- Directives
- Weak Phrases
- Continuances
- Options

The categories of indicators related to the entire requirements document are:

- Size
- Readability
- Specification Depth
- Text Structure

4.1.1. Imperatives

Imperatives are those words and phrases that command that something must be provided. The ARM report lists the total number of times each imperatives was detected in the sequence that they are discussed below. This list presents imperatives in descending order of their strength as a forceful statement of a requirement. The NASA requirements documents that were judged to be the most explicit had the majority of their imperative counts associated with the upper list items.

- ***Shall*** is usually used to dictate the provision of a functional capability.
- ***Must*** or ***must not*** is most often used to establish performance requirements or constraints.
- ***Is required to*** is often used as an imperative in specifications statements written in the passive voice.
- ***Are applicable*** is normally used to include, by reference, standards or other documentation as an addition to the requirements being specified.
- ***Responsible for*** is frequently used as an imperative in requirements documents that are written for systems whose architectures are predefined. As an example, *"The XYS function of the ABC subsystem is responsible for responding to PDQ inputs."*
- ***Will*** is generally used to cite things that the operational or development environment are to provide to the capability being specified. For example, *"The building's electrical system will power the XYZ system"*. In a few instances "shall" and "will" were used interchangeably within a document that contained both requirements specifications and descriptions of the operational environment. In those documents, the boundaries of the system being specified were not always sharply defined.
- ***Should*** is not frequently used as an imperative in requirement specification statements. However, when is used, the specifications statement is always found to be very weak. For example, "Within reason, data files should have the same time span to facilitate ease of use and data comparison."

4.1.2. Continuances

Continuances are phrases such as those listed below that follow an imperative and introduce the specification of requirements at a lower level. The extent that continuances were used was found to be an indication that requirements were organized and structured. These characteristics contribute to the tractability and maintenance of the specified requirements. However, in some instances, extensive use of continuances was found to indicate the presence of very complex and detailed requirements

specification statements. The continuances that the ARM tool looks for are listed below in the order most frequently found in NASA requirements documents.

- below:
- as follows:
- following:
- listed:
- in particular:
- support:

4.1.3. Directives

Directives is the category of words and phrases that point to illustrative information within the requirements document. The data and information pointed to by directives strengthens the document's specification statements and makes them more understandable. A high ratio of the total count for the Directives category to the documents total lines of text appears to be an indicator of how precisely requirements are specified. The directives that the ARM tool counts are listed below in the order that they are most often encountered in NASA requirements specifications.

- figure
- table
- for example
- note:

4.1.4. Options

Options is the category of words that give the developer latitude in satisfying the specification statements that contain them. This category of words loosen the specification, reduces the acquirer's control over the final product, and establishes a basis for possible cost and schedule risks. The words that the ARM tool identifies as options are listed in the order that they are most frequently used in NASA requirements documents.

- can
- may
- optionally

4.1.5. Weak Phrases

Weak Phrases is the category of clauses that are apt to cause uncertainty and leave room for multiple interpretations. Use of phrases such as "adequate" and "as appropriate" indicate that what is required is either defined elsewhere or, worse, that the requirement is open to subjective interpretation. Phrases such as "but not limited to" and "as a minimum" provide a basis for expanding a requirement or adding future requirements. The total number of weak phrases found in a document is an indication of the extent that the specification is ambiguous and incomplete. The weak phrases reported by the ARM tool are:

- adequate
- as applicable
- as appropriate
- be capable
- capability of
- effective
- normal
- timely
- as a minimum
- easy
- be able to
- but not limited to
- capability to
- if practical
- provide for

4.1.6. Size

Size is the category used by the ARM tool to report three indicators of the size of the requirements specification document. They are the total number of:

- lines of text
- imperatives
- subjects of specification statements
- paragraphs

The number of lines of text in a specification document is accumulated as each string of text is read and processed by the ARM program. The number of subjects used in the specification document is a count of unique combinations and permutations of words immediately preceding imperatives in the source file. This count appears to be an indication of the scope of the document. The ratio of imperatives to subjects provides an indication of the level of detail being specified. The ratio of lines of text to imperatives provides an indication of how concise the document is in specifying the requirements.

4.1.7. Text Structure

Text Structure is a category is used by the ARM tool to report the number of statement identifiers found at each hierarchical level of the requirements document. These counts provide an indication of the document's organization, consistency, and level of detail. The most detailed NASA documents were found to have statements with a hierarchical structure extending to nine levels of depth. High level requirements documents rarely had numbered statements below a structural depth of four. The text structure of documents judged to be well organized and having a consistent level of detail were found to have a pyramidal shape (few numbered statements at level 1 and each lower level having more numbered statements than the level above it). Documents that exhibited an hour-glass shaped text structure (many numbered statements at high levels, few at mid levels and many at lower levels) were usually those that contain a large amount of introductory and administrative information. Diamond shaped documents (a pyramid followed by decreasing statement counts at levels below the pyramid) indicated that subjects introduced at the higher levels were addressed at different levels of detail.

4.1.8. Specification Depth

Specification Depth is a category used by the ARM tool to report the number of imperatives found at each of the documents levels of text structure. These numbers also include the count of lower level list items that are introduced at a higher level by an imperative and followed by a continuance. This data is significant because it reflects the structure of the requirements statements as opposed to that of the document's text. Differences between the Text Structure counts and the Specification Depth were found to be an indication of the amount and location of text describing the environment that was included in the requirements document. The ratio of the total for specification depth category to document's total lines of text appears to be an indication of how concise the document is in specifying requirements.

4.1.9. Readability Statistics

Readability Statistics are a category of indicators that measure how easily an adult can read and understand the requirements document. Four readability statistics produced by Microsoft Word are currently calculated and compared:

- Flesch Reading Ease index is based on the average number of syllables per word and the average number of words per sentence. Scores range from 0 to 100 with standard writing averaging 60 - 70. The higher the score, the greater the number of people who can readily understand the document.
- Flesch-Kincaid Grade Level index is also based on the average number of syllables per word and the average number of words per sentence. The score in this case indicates a grade-school level. A score of 8.0 for example, means that an eighth grader would understand the document. Standard writing averages seventh to eighth grade.
- Coleman-Liau Grade Level index uses word length in characters and sentence length in words to determine grade level.
- Bormuth Grade Level index also uses word length in characters and sentence length in words to determine a grade level.

Since most documents at NASA contain scientific terms which tend to contain words of considerable length, the readability scores are skewed. As shown in Table 1. below, using the data from forty-six requirements documents, the grade levels indicated by the Flesch-Kincaid, Coleman-Liau and Bormuth have a high variance, and the mean for Coleman-Liau grade level is 27.6 with a maximum of 55.80. Alternative readability packages that allow for adjustment of word length are being investigated.

	Flesch Rdng Ez	Flesch-Kincaid Grd Lvl	Coleman-Liau Grd Lvl	Bormuth Grd Lvl
mean	47.15	10.76	27.60	11.46
min	28.00	7.80	17.10	11.10
max	61.50	13.80	55.80	11.60
stdev	8.54	1.59	9.29	0.17

Table 1.

4.2. Quality Indicator/Attribute Relationships

Relationships between requirements specifications' quality attributes and categories of indicators measured by the ARM tool are shown below in Figure 1.

INDICATORS OF QUALITY ATTRIBUTES											
	Quality Attributes										
Categories of Quality Indicators	1. Complete	2. Consistent	3. Correct	4. Modifiable	5. Ranked	6. Testable	7. Traceable	8. Unambiguous	9. Understandable	10. Validatable	11. Verifiable
1. Imperatives	X			X			X	X	X	X	X
2. Continuances	X			X	X	X	X	X	X	X	X
3. Directives	X		X			X		X	X	X	X
4. Options	X					X		X	X	X	
5. Weak Phrases	X		X			X		X	X	X	X
6. Size	X					X		X	X	X	X
7. Text Structure	X	X		X	X		X		X		X
8. Spec. Depth	X	X		X			X		X		X
9. Readability				X		X	X	X	X	X	X

Figure 1.

5. RISK

Webster's New World Dictionary defines risk as the possibility of loss or injury, and the degree of probability of such a loss. As the first tangible representation of the needed capability, the requirements specification establishes the basis for all of the project's engineering, management and assurance functions. If its quality is poor, it can give rise to risks associated with the project's products and its resources.

5.1. Product Risks

Inadequacies in the requirements specification document introduces the possibility that the product's design will contain deficiencies that will be propagated as implementation faults. The possibility of these faults increases the probability that one or more of the products characteristics will be unsatisfactory.

5.1.1. Acceptance Risk

An *Acceptance Risk* is the probability that the product will not satisfy its acceptance criteria. A deficiency in any of

the requirements specification's quality attributes may introduce a risk that the product will not be acceptable.

5.1.2. Availability Risk

An *Availability Risk* is the probability that the product will not be present or functional at a time when it is needed. A deficiency in any of the requirements specification's quality attributes may introduce a risk that the product will not be delivered on time or its functionality can not be maintained in the operational environment.

5.1.3. Performance Risk

A *Performance Risk* is the probability that the product will not be capable of performing properly in the operational environment. A performance risk will arise if the requirements specification is difficult to understand or inadequately specifies the functional capabilities that are to be provided.

5.1.4. Reliability Risk

A *Reliability Risk* is the probability that the product will fail in the operational environment. A deficiency in any of the requirements specification's quality attributes can introduce a risk that the product will not achieve the level of reliability need to successfully perform its mission.

5.1.5. Reproducibility Risk

A *Reproducibility Risk* is the probability that the product can not be reproduced for replacement of the original product or for distribution to additional sites. If the requirements specification is poorly written and functions are inadequately prescribed there will be a risk that the product or components of the product cannot be replicated when replacement is necessary.

5.1.6. Supportability Risk

A *Supportability Risk* is the probability that the product can not be adequately maintained or logistically provided for in the operational environment. A deficiency in any of the requirements specification's quality attributes can introduce a risk that the resources needed to operate and maintain the product will not be provided and/or the product itself does facilitate maintenance.

5.1.7. Utility Risk

A Utility Risk is the probability that the product will less useful than demanded by the operational environment. A risk that the delivered capability will not provide the user with the full range of necessary functionality will arise if the requirements specification is difficult to understand or the functions that are to be provided are inadequately specified.

5.2. Resource Risks

The estimate of resources needed to provide a capability, including the time allocated for its acquisition, are based of the specification of the requirements that the capability is to satisfy. If the requirements are improperly specified the resource estimates will be incorrect. Product risks introduced by deficient requirements can result in delays and rework that introduce cost and schedule risks.

5.2.1. Cost Risk

A *Cost Risk* is the probability that the production or acquisition of the product will exceed allocated resources. available for that purpose. Any deficiency in the requirements specification, such as incompleteness or ambiguity, that causes the development effort to be under estimated or necessitates rework will introduce a risk that costs will exceed available funding.

5.2.2. Schedule Risk

A *Schedule Risk* is the probability that the product will not be delivered as scheduled. Any deficiency in the requirements specification's quality attributes can introduce a risk that the established schedule is inadequate or that product deficiencies will require unanticipated additional time to correct.

5.3. Quality-Risk Relationships

Figure 2 shows the areas of project risk that are directly impacted by the quality attributes of the requirements specification.

RISK AREAS IMPACTED BY SPECIFICATION QUALITY									
	1. Product Risk							2. Resource Risk	
Specification Quality Attributes	1.1. Acceptance	1.2. Availability	1.3. Performance	1.4. Reliability	1.5. Reproducibility	1.6. Supportability	1.7. Utility	2.1. Cost	2.2. Schedule
1. Complete	X	X	X	X	X	X	X	X	X
2. Consistent	X		X	X	X	X			
3. Correct	X	X	X	X	X	X	X	X	X
4. Modifiable	X	X	X	X	X	X	X	X	X
5. Ranked	X	X	X	X	X	X	X	X	X
6. Testable	X	X		X	X	X		X	X
7. Traceable	X			X	X	X			X
8. Unambiguous	X		X	X	X	X	X	X	X
9. Understandable	X		X	X	X	X	X	X	X
10. Validatable	X			X	X	X			X
11. Verifiable	X			X	X	X	X		X

Figure 2.

6. Specification Standards

Although many government and several professional organizations have published documentation standards that include standards for specifying requirements, none are universally accepted or extensively enforced. NASA has very explicit software documentation standards and style guides, but it allows wide latitude in establishing project

standards. In many instances, to minimize effort and reduce costs, projects choose to accept the documentation standards and procedures of the contractor selected to provide the needed capability. Smaller projects, of which there are many in-house in addition to contractual acquisitions, are also frequently combine requirement and design documents to conserve project resources. The standards that are imposed seldom go beyond providing an outline and a general description of the information to be provided. In many cases no style requirements are established. As a consequence of these circumstances, requirements documents from various sources bear little resemblance to one another.

The content outline of one of the IEEE's eight specification templates [6] and NASA's standard data item description (DID) for requirement specifications documents [8] are shown below and provide an example of the variance in scope that exists between standards.

IEEE 830 93

Software Requirement Specification

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, acronyms, and abbreviations
   - 1.4 References
   - 1.5 Overview
2. Overall description
   - 2.1 Product perspective
   - 2.2 Product functions
   - 2.3 User characteristics
   - 2.4 Constraints
   - 2.5 Assumptions and dependencies
3. Specific requirements
   - 3.1 External interfaces
   - 3.2 Functions
   - 3.3 Performance requirements
   - 3.4 Logical database requirements
   - 3.5 Design constraints
   - 3.6 Software system attributes
   - 3.7 Organizing the specific requirements
   - 3.8 Additional comments

Appendixes

Index

NASA-DID-P200

Requirements

1.0 Introduction
2.0 Related documentation
3.0 Requirements approach and tradeoffs
4.0 External interface requirements
5.0 Requirements specification
   - 5.1 Process and data requirements
   - 5.2 Performance and quality engineering requirements
   - 5.3 Safety requirements
   - 5.4 Security and privacy requirements
   - 5.5 Implementation constraints
   - 5.6 Site adaptation
   - 5.7 Design goals

6.0 Traceability to parent's design
7.0 Partitioning for phased delivery
8.0 Abbreviations and acronyms
9.0 Glossary
10.0 Notes
11.0 Appendices

Several problems related to the structure and organization of the source documents were frequently encountered when attempting to automate the scanning of specification files. The most troubling problem was the inconsistency in paragraph and specification identification across documents and within documents. The variations in identification schemes most frequently encountered are shown below in the order of their prevalence.

- Hierarchical Numbers - 1.2.3., 1.2.3.4.., ...1.2.3.4.5.6.7.8., etc.
- Lettered Hierarchical Numbers P1.2.3., Q1.2.3.4.., ...S1.2.3.4.5.6.7.8, etc.
- Integer Numbers - 1., 2.,3.,...10.; 1, 2, 3, 20, 21; 30; [1], [2], [3];[20]
- Letters A., B., C.; a., b., c.; a), b), c); (a), (b), (c)

Because of these numbering inconsistencies the current version of the Automated Requirement Measurement (ARM) software is implemented using the following scheme as the basis for recognizing paragraph and specification identifications.

Each requirement specification statement is assumed to be individually distinguished by one of the following markings:

a. A simple number (i.e. a number without decimal. For example: 1, 23, 104, etc.)
b. A hierarchical number (i.e. a number with decimals to indicate levels of structure. For example: 1.1.1, 23.4.5.6, etc.)

c. A lettered hierarchical number (i.e. a hierarchical number immediately preceded by a letter. followed by a period. For example: L1.1.1, B23.4.5.6, etc.)
d. An integer number (i.e. a simple number. For example: 1., 23., 104., etc.)
e. A letter designation (i.e. a single capital/lower-case letter followed by a period. For example: A., P., b., x. etc.)

Another problem encountered when scanning requirement specification files is distinguishing statements that *prescribe* required system capabilities from those that *describe* the operational environment. This problem has both structural and terminology aspects. In general three structural separations of descriptive and prescriptive information are usually used. The NASA documentation standards call for information to be presented in two distinct documents. The targeted operational environment is to be described within the NASA Concept Document, NASA-DID-P100 [8] while the standard shown above, NASA-DID-P200, is to contain prescriptive specifications. Other standards [2] [6] allocate descriptive and prescriptive information to separate parts of the same document. In several NASA contractor provided requirements documents the two categories of information were interlaced within the same sections of the document.

In some instances the ability to distinguish between statements prescribing requirements and those describing environmental features was further complicated by authors using the imperatives "shall", will", and "should" interchangeably.

7. Results

The graphs presented in this section are based on data collected by the ARM processor. The assessments accompanying the graphical presentations are based on an examination of the source documents in light of the data used to create the graphs.

7.1. Document Size And Imperatives

The relationship between document size, measured by total number of imperatives and total lines of text found in each specification document is depicted by Figure 3., below.

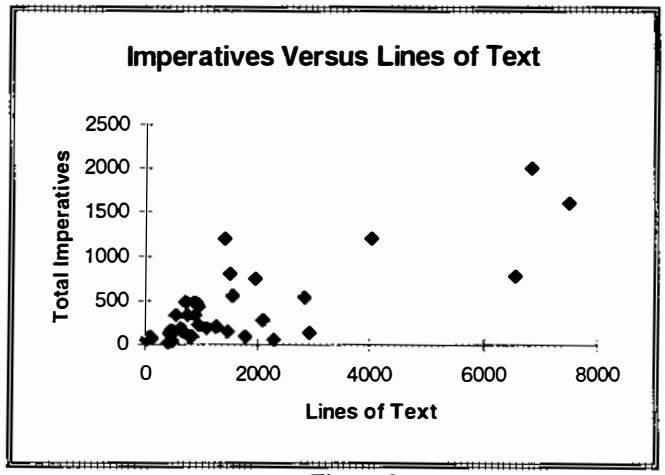


Figure 3.

The ratio of lines of text to imperatives for each specification document is shown in Figure 4. Most documents have ten lines of text, or less, for each imperative. When inspected, the documents with ratios that exceed one-to-one, appeared to have been created using a requirements analysis methodology or tool. The low ratios resulted of the fact that the documents exclusively addressed software requirements and many single statement contains multiple imperatives connected by "and" or "or". Those specification documents with ratios above ten lines of text per imperative appear to have been developed based on an ad-hoc documentation standards with scopes that includes the description of the operational environment as well a the prescription of system requirements

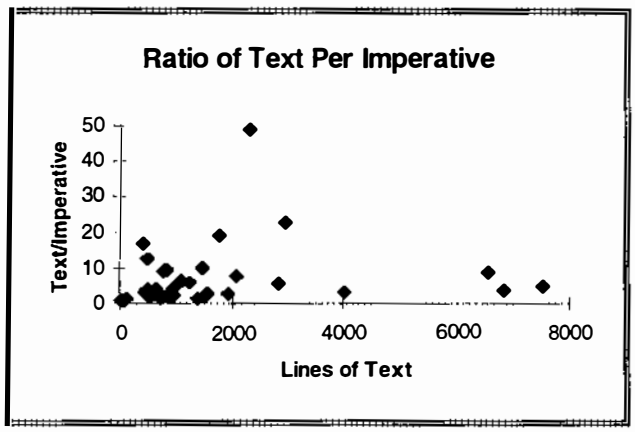


Figure 4.

The document with the highest ratio of text to imperatives also included descriptions of the project and requirements for the development environment. This extreme data point was removed in Figure 4a. to improve the visibility of the remaining data points.

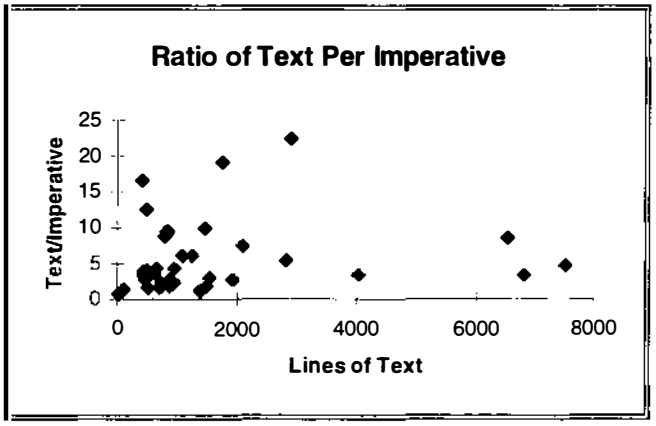


Figure 4a.

7.2. Document Size And Structure

The relationship between document size and the document's depth of text structure shown by Figure 5. This data is based on detection and automatic analysis of paragraph numbering and statement identifiers.

In general, the smaller documents have a greater percent of their statements numbered. Inspection of these documents indicates that they were probably developed using Computer Aided Software Engineering (CASE) tools. They are very hierarchical in structure. High level statements have been repetitively decomposed into subordinate statements. Every level of text is hierarchically identified and each statement containing an imperative has been given an additional unique sequential number.

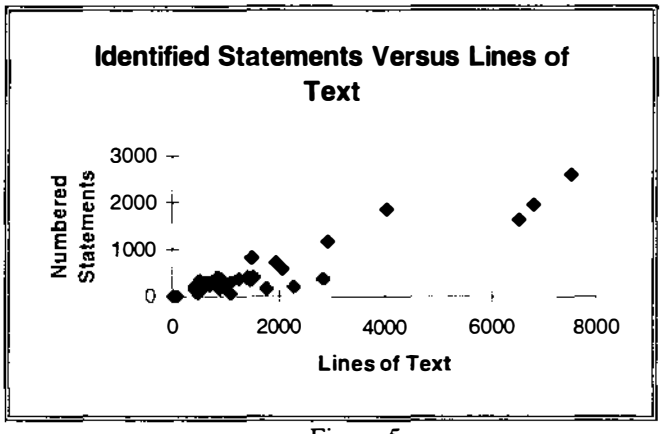


Figure 5.

Figure 6. shows the ratio of numbered statements to the number of imperatives within each specification document. Most of the documents containing less than 100 imperatives have a relatively high ratio. Inspection of these documents revealed that they were developed using a documentation format that was too detailed for the scope of the capability being specified. A significant number of the sections in these documents addressed administrative and general information rather than requirements specifications. In general, documents with more than 100 imperatives seem to have a ratio of numbered statements to imperatives close to one. This implies that these document's specification statements have a high degree of traceability.

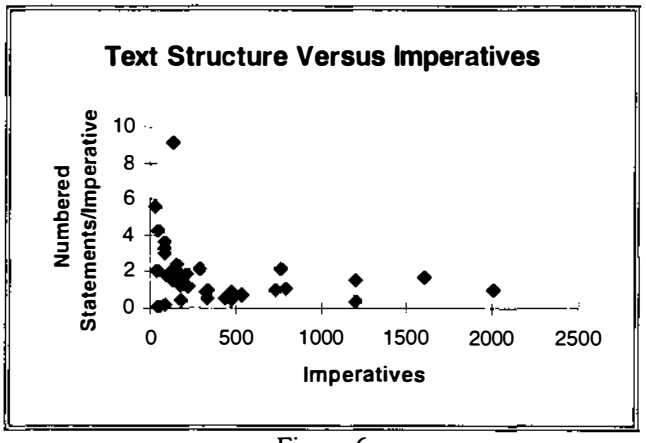


Figure 6.

7.3. Imperatives And Subjects

The relationship between the number of imperatives and the unique number of subjects of imperative statements is shown by Figure 7. Closer examination of these documents found that the real ratio between subjects and imperatives is actually lower than shown in Figure 8. In many instances the same subject was stated in different terms, apparently to introduce variety and hold the readers interest. The ARM software is not capable of recognizing different phrases as the same subject and counts each distinct combination of words immediately preceding the imperative as a unique subject. The larger documents were found to be much greater in scope and to addressed subjects at a higher level of capability. In these higher level specifications, multiple statements were used to define and bound each major functional capability

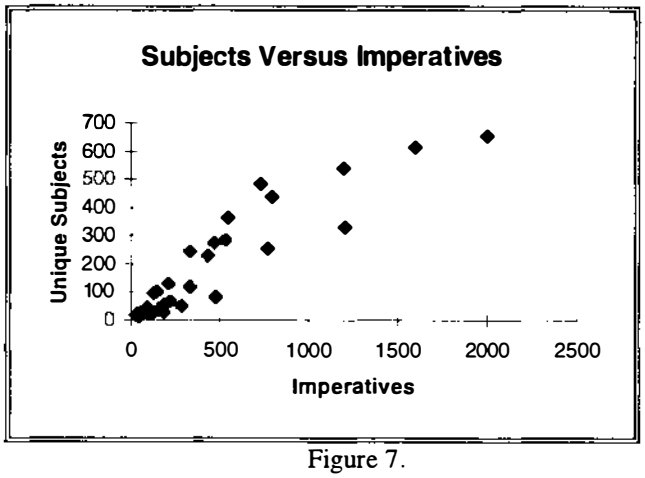


Figure 7.

Figure 8. shows the ratio of imperatives to subjects versus lines of text. Within a limited amount of variation, it appears that even in those documents containing many

imperatives per subject, each imperative is supported with a nominal number of text lines

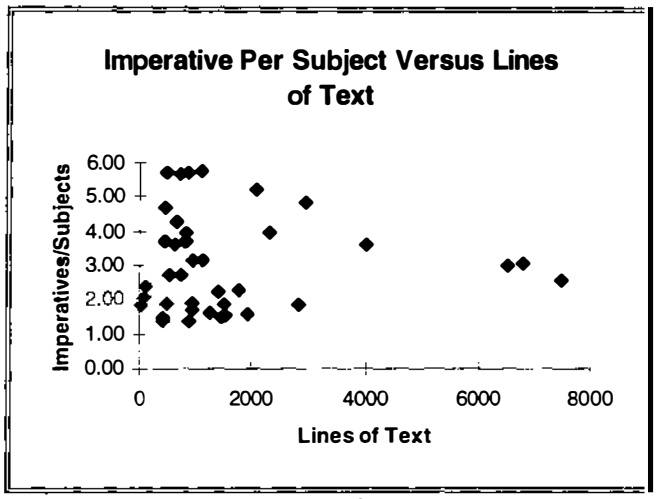


Figure 8.

7.4. Weak Phrases

Figure 9. shows the number of weak phrases versus the number of imperatives found in NASA requirements documents. This representation implies a somewhat linear relationship between these to indicators for large documents.

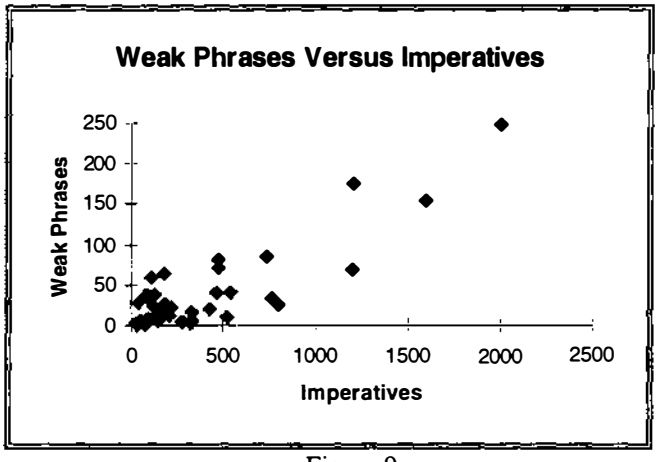


Figure 9.

Figure 10. Presents the ratio of weak phrases to imperatives versus the number of imperatives contained in the document. In most of these documents it appears that one to ten percent of the individual specification statements contain weak phrases. Weak phrases were found in thirty to sixty percent of the individual specification statements in many of the smaller documents. These are same documents that were well structured as a result of using CASE tools.

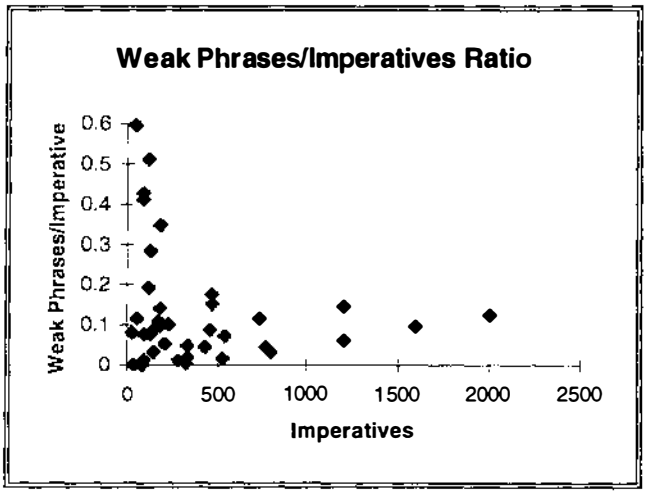


Figure 10.

8. Conclusions

Three initial general conclusions have arisen from the subject study, First, it is possible to gain insights into the quality of a requirements specification document through the use of data gathered by automated processing of the specification file. Second, the effectiveness of expressing requirements specifications with natural language can be greatly improved through relatively simple and readily available methods. Lastly, specifications developed using a proven methodology with the aid of an appropriate requirements definition tool are better structured, more consistently numbered, and crisper than those developed solely based on a documentation standard. Use of a CASE tool, however, is not a substitute for sound engineering analysis.

8.1. Management Recommendations

Based on the current results of the SATC study it is recommended that project managers ensure that requirements specification and style standards are established at the outset of the project and that all project participants are trained in the use of those standards. It is also recommended that the technical nature and intended use of the requirements document be emphasized to encourage straightforward writing styles and simple sentence structures to specify each requirement. Technical documents are not required to be interesting, however, it is necessary for them to effectively communicate with the reader..

8.2. Technical Recommendations

Requirements analysis an development of specifications should occur prior to writing the requirements document, not as a by-product of the documentation activity. Requirement specifications should be developed using a methodology that is appropriate to the nature of the project and its products. CASE tools should be used if they support the project's development methodology and the

technology being addressed by the project. Specification writers should be taught how to write simple direct statements. If conditional clauses are needed, they should be placed at the front of the sentence. Subjects, imperative/verb combinations, and objects of the verb should occur in that order. This will prevent interesting, but confusing statements, such as: "The Romans, the Greeks shall defeat if better prepared they be."

9. Future Work

Three types of enhancements are planned for the ARM software. The enhancement of immediate priority is to improve the ARM tool's user interface and to provide flexibility to tailor its reports to the user's areas of interest. This will facilitate enlisting additional projects to participate in the SATC study, add their requirements documents to the SATC database and provide user feedback.

The second ARM tool enhancement activity will take place in parallel with the improvement of its user interface and report generator. The results of the engineering review of the requirements specifications in the SATC database will be used to refine ARM's search and counting schemes. This will improve the validity of data produced and heighten user confidence in the conclusions that can be inferred from its reports

Lastly, once the ARM's search and counting schemes have been enhanced, words and phrases will be identified that can be used as indicators that the following types of requirements have been addressed within the requirements document.

- Acceptance Testing
- Data Handling
- Design Standards
- Integrity
- Maintainability
- Reliability
- Reusability
- Timing and Sizing

10. References

[1] Brooks, Frederick P. Jr., No Silver Bullet: Essence and accidents of software engineering, *IEEE Computer, vol. 15, no. 1,* April 1987, pp. 10-18.

[2] DOD MIL-STD-490A, Specification Practices, June 4, 1985.

[3] Ganska, Ralph, Grotzky, John, Rubinstein, Jack, Van Buren, Jim, Requirements Engineering and Design Technology Report, Software Technology Support Center, Hill Air Fource Base, October, 1995.

[4] Gause, Donald C., and Weinberg, Gerald M., Exploring Requirements Quality Before Design, Dorset House Publishing, NY, NY, 1989

[5] IEEE Std 729-1983, Standard Glossary of Software Engineering Terminology, Feburary 18, 1983.

[6] IEEE Std 830-1993, Recommended Practice for Software Requirements Specifications, December 2, 1993.

[7] Kitchenham, Barbara, Pfleeger, Shari Lawrence, Software Quality: The Elusive Target, *IEEE Software, Vol. 13, No. 1*, January 1996, pp. 12-21.`

[8] NASA-STD-2100-91, NASA Software Documentation Standard, NASA Headquarters Software Engineering Program, July 29, 1991.

[9] Porter, Adam A., Votta, Lawrence G., Jr., and Basili, Victor R., Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment, *IEEE Transactions on Software Engineering, Vol. 21, No. 6*, June 1995, pp. 563-574.

[10] Sommerville, Ian, Software Engineering, Fourth Edition, Addison-Wesley Publishing Company, Wokingham, England, 1992.

[11] Stokes, David Alan, Requirements Analysis, *Computer Weekly Software Engineer's Reference Book*, 1991, pp. 16/3-16/21.

A Comprehensive SQA Strategy for a Re-Engineered Handheld Data Collection Application

Bernard E. Gracy, Jr.
United Parcel Service
2311 York Rd
Timonium, MD 21093
(410) 560-4191
bxg@ismd.ups.com

Abstract

The software quality assurance (SQA) process for resource constrained handheld data collector applications must go beyond correctness, coverage, reliability, and availability -- battery consumption, memory/CPU utilization, and performance must be evaluated as well. The SQA challenge is further exacerbated if these handheld applications are targeted for both domestic *and* international usage across multiple hardware platforms.

However, the SQA process meets its ultimate challenge on a re-engineered software product. While much has been written on the software re-engineering process, little has been written on the certification of the resultant software products. This paper discusses a comprehensive SQA strategy employed to address the challenges of re-engineering a handheld data collection application that has an installed base of 80,000 users in 18 countries in 5 different languages across 3 different handheld platforms.

About the Author

Bernard E. Gracy, Jr. is UPS' DIAD Product Manager. He received his BSE in Computer Science and Engineering from the University of Connecticut in 1985 and his MSCS from Rensselaer Polytechnic Institute in 1993. He is a member of the ACM - SIGOPS and SIGSOFT.

A Comprehensive SQA Strategy for a Re-Engineered Handheld Data Collection Application

Bernard E. Gracy, Jr.
United Parcel Service
2311 York Rd
Timonium, MD 21093
(410) 560-4191
bxg@ismd.ups.com

Abstract

The software quality assurance (SQA) process for resource constrained handheld data collector applications must go beyond correctness, coverage, reliability, and availability -- battery consumption, memory/CPU utilization, and performance must be evaluated as well. The SQA challenge is further exacerbated if these handheld applications are targeted for both domestic *and* international usage across multiple hardware platforms.

However, the SQA process meets its ultimate challenge on a re-engineered software product. While much has been written on the software re-engineering process, little has been written on the certification of the resultant software products. This paper discusses a comprehensive SQA strategy employed to address the challenges of re-engineering a handheld data collection application that has an installed base of 80,000 users in 18 countries in 5 different languages across 3 different handheld platforms.

Keywords

Software quality assurance, software re-engineering, automated testing, concurrent engineering

1 Introduction

In 1989, UPS introduced its first electronic handheld data collector -- the Delivery Information Acquisition Device (DIAD) -- into the domestic US delivery operation to replace the acquisition of package information by a paper delivery record. Since that time, two additional distinct hardware platforms have been introduced, and all three platforms are in service today. Furthermore, the role and mission of the DIAD has rapidly evolved from a simple data collector to an integral component in a global real-time tracking and dispatching network, enabling UPS to offer a broad range of innovative information services. As a result, the legacy DIAD application software has had several major releases in 18 countries in 5 different languages. However, the competitive and time-to-market pressures in conjunction with the radical evolution in functionality had put great strain on the original software architecture. The DIAD application had to be re-

engineered to allow further global service expansion, reduce time-to-market, and increase product quality.

Software re-engineering is a process which improves one's understanding of software and prepares and improves the software itself for increased maintainability, reusability, and evolution [1]. However, while much has been written about the software re-engineering process, little has been written about the SQA process of re-engineered products. It has been our experience that the success of this re-engineering initiative has been tied to not only re-engineering the software product, but the associated development and SQA process as well.

2 The Legacy Development Process

The DIAD software development lifecycle strictly adhered to the classic waterfall model. One of the known dangers of this model is that it is often difficult in the beginning for the customer to state all requirements explicitly [2]. However, compared to Digital Equipment Corp's re-engineering effort [3], few problems exist in the product requirements definition process. Disparate business functions propose DIAD functionality and features to a central cross-functional steering committee which reviews the relative cost/benefit and approves funding. The sponsoring function then works closely with the DIAD software development staff in the actualization of the business requirements and user specifications. Because all UPS employees have an innate knowledge of UPS operations and the business case for any new functionality is clearly communicated, there is minimal misinterpretation of requirements or specifications by the sponsoring business function or the development engineers. Requirements are clearly defined and unambiguous. The only hindrance was that the domestic product requirements definition process was independent of the international product requirements definition process and vice-versa. As such, the product team was separated into domestic and international project teams to ensure that our internal customer's needs were being met.

The bottleneck in the lifecycle was therefore not "requirements churn" [3] but the high degree of coupling of the legacy DIAD application. All the requirements from dissimilar business functions and the associated specifications for an entire product release had to be codified before development could begin because of the high degree of interrelatedness of product components and the subsequent likelihood that defects would be introduced and/or rework required if requirements staggered in through the development cycle. Over time, business functions were required to define requirements earlier and earlier for products that would appear later and later. This too is a known detriment of the waterfall model but in this case the customer's need for patience was more product inherent than process inherent.

Furthermore, the split of the product team into domestic and international project teams exacerbated the strain on the legacy DIAD application. Because of the strict and independent time-to-market requirements in a rapidly expanding market, the single DIAD application became several distinct applications from several distinct source bases that were now separately and independently maintained. As such, a defect discovered and fixed in one source base had to potentially be replicated in several other source bases. Moreover, that defect would then have to

be certified as fixed by the product test group several times -- one for each source base across three distinct hardware platforms.

Ironically, while the DIAD was introduced to replace a manual recording process, the back end legacy product test process was strictly manual itself. Initially, test plans were developed and maintained using standard word processing applications; and manually executed by test technicians. However the same competitive and time-to-market pressures in conjunction with the radical evolution in functionality that had put great strain on the original software architecture put an even greater strain on the manual testing process. The number and scope of test plans exploded. People from many countries were brought in to assist the testing of the product in their native language. As such, product testing was taking more and more of the development lifecycle. Because of these pressures, full regression tests could no longer be done. However, this revised strategy has proven effective since the DIAD application while strained is a field proven software product.

3 Defining A Comprehensive SQA Strategy

Re-engineering is the fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in critical measures of performance, such as cost, quality, service, and speed. It requires inductive thinking to first recognize powerful solutions and then seek the problems it might solve [4]. In rethinking and redesigning our business process to support rapid global expansion and recognizing powerful solutions for the future, it became clear that we needed to:

1. Migrate from a provincial to a global requirements definition process.

2. Combine and reconcile the existing domestic and international functionality.

3. Migrate from maintaining several source bases to a single source base.

4. Migrate from the strict serialization of the waterfall model to the parallelism of a concurrent engineering model to increase engineering productivity by enabling small cross-functional development teams to work *concurrently* on the DIAD product.

5. Enable software test to be performed *concurrently* with product software development to find problems early in the development cycle instead of waiting for product/acceptance certification.

6. Enable UPS to introduce new handheld products without radically changing software.

The implications of meeting these objectives were intimidating. Originally envisioned as a simple application rewrite, the handheld re-engineering effort had also become:

- A monumental consolidation and reconciliation of requirements, specifications, and test plans of radically different application functionality, operating languages, and data collection

requirements. What were once clear and unambiguous requirements were now not only ambiguous but conflicting.

- A configuration management challenge in that we now had to account for platform, language, common functionality, region-specific functionality, country-specific functionality, and configuration differences.

- An organizational challenge to merge mature independent project teams into a single cohesive product group with a common vision and strategy

The primary objective of the handheld re-engineering effort was to be able to rapidly respond to changing market conditions. To that end, the re-engineering effort had to encompass software architecture, configuration management, test automation, documentation, organization, and the development process -- a comprehensive and sustainable SQA strategy. Digital's re-engineering effort [3] was designed to ensure customer satisfaction and business readiness by migrating from a technology-centered view of product development to a customer-centric view because of mature development process but an immature requirements definition process. Conversely, we had to migrate from a customer-driven release-centric view of product development to a technology-driven product-centric view because of a mature requirements definition process but an immature development process and software architecture. While Herculean in scope, only a comprehensive approach would ensure long term success -- any deficiency in any one area would render the product line unsustainable. The following describes the pillars of this comprehensive SQA strategy.

4 Software Architecture

The highly coupled monolithic architecture of the DIAD application was the primary impediment to migrating to a concurrent development paradigm. To allow multiple development teams to work concurrently on the same software base would require a sweeping rewrite of the existing software - merging multiple source bases into one would yield minimal benefit because of the extensive component interdependencies. An analysis of the installed base defined the business boundary conditions of the DIAD software re-engineering effort -- while there was some degree of overlap in a majority of the DIAD functionality, each region and/or country had unique product, service, and data collection requirements due to legal, regulatory, cultural, pecuniary, addressing, and customer needs. It was therefore impractical if not impossible to develop, support, and maintain a single "kitchen sink" application that could be fielded anywhere on the planet. Instead, an intensive investigation of all fielded DIAD products yielded a "plug and play" Application Modularization Framework that would allow different products to be generated from a single source base for a specific region, country, state/province, operating center, or even a single UPS Service Provider. To that end, DIAD functionality was re-classified and re-implemented as either an *Application, Service Module, Service,* or *Panel*:

- An *Application* is a distinct thread of execution for a specific intended purpose. Using a Windows analogy, Microsoft Word and Excel are separate applications that are co-resident, interoperate, and yet independent of each other. Similarly, separate and distinct

functionality that were once cemented together in the monolithic legacy implementation could now be co-resident, interoperate, and yet stand alone. It was now conceptually possible to add separate and distinct functionality without "polluting" the original implementation.

- A *Service Module* is a distinct independent linkable context of an Application. An Application can have zero or more Service Modules. Service Modules are incorporated or deliberately not incorporated into an Application at product build time but announce themselves to the Application at initialization time. This allows us to introduce and remove Service Modules without changing an Application. If incorporated, Service Modules can be enabled or disabled by an external downloadable configuration or initialization logic, or user entry. This capability allows us to deploy functionality ahead of its actual introduction in the marketplace and/or remove functionality at run time if requisite data is not available

  A Service Module "owns" its constituent Services and Panels (see below). If the Service Module is incorporated into the Application, the constituting Services and Panels are automatically incorporated as well, whether the Service Module is enabled or disabled. Conversely, if the Service Module is not incorporated, neither are its Services and Panels. This feature allows us to optimize our utilization of DIAD memory. But most important, this organizational paradigm facilitates the componentization of our source base. We can now manage cohesive Service Modules rather than snippets of functionality spread over separate source bases.

- A *Service* is a distinct independent linkable context of a Service Module. A Service Module may have zero or more Services. Like a Service Module, it can be incorporated or removed, enabled or disabled. However, where Service Modules are disabled at configuration or initialization time, Services can be disabled and enabled at run time by the Service Module. This capability allows us the flexibility to enable or disable functionality based on user entry or Application state while still supporting a component organizational paradigm.

- A *Panel* is the logical input/output mechanism for an Application, Service Module, Service, or even other Panels. Panels encompass none, some, or all parts of the physical DIAD screen. Panels define menus, fields, scrollable regions, valid keys, navigation rules, validation rules, and action functions. An Application, Service Module, Service, and/or Panel can have zero or more Panels defined.

A series of enabling technologies made this architectural re-engineering possible. We introduced a platform independent hardware abstraction layer to facilitate porting to all handheld platforms, a commercial preemptive multitasking OS to facilitate decoupling and scalability, a system-wide event driven paradigm, an event-driven commercial user interface, string externalization, a pseudo-relational database, a re-entrant and restartable filesystem, a scaleable heap, and a host of other services. Furthermore, The Applications, Service modules, Services, Panels, fields, user interface, database, and OS execution behavior are all defined in a series of rule tables which are

easily expressed, modified, and maintained. While an in-depth discussion of each of these initiatives are beyond the scope of this paper, they are mentioned to demonstrate the totality of the architectural re-engineering effort and the extent of the infrastructure required to support the plug and play Application Modularization Framework

5 Configuration Management

As libraries of reusable software components continue to grow, the issue of retrieving components from software component repositories has captured the attention of the software reuse community. In order for the approach to be useful, the repository must contain enough components to support development needs, a pre-defined structure to categorize and place these components, and a retrieval mechanism to utilize them. Typically, structures required for retrieval are often static and unable to adapt to dynamic development requirements because of the fluid nature of changing technology, project dynamics, and customer needs. Once the structures are created, they are set in stone[5]. This is what happened to the legacy DIAD source base. Because of the separation of the product team into multiple project teams and the time-to-market requirements, the source base was replicated in several distinct and separately managed version control environments. Furthermore, the managed component was the constituent source file -- not the logical abstraction to which that source file contributed. As such, each product version label was applied to each source module within its version control environment.

In contrast, the Application Modularization Framework enabled multiple products to be generated from a pool of independently managed components that are largely release independent. By developing a series of value-added extensions to our version control system, we developed our "bill of materials" DIAD Software Configuration Management System (SCMS). Our DIAD SCMS allows us to generate different combinations of Applications, Service Modules, Services, and Panels from a true component base derived from a single source base. This mechanism is similar to a parts list or bill of materials used to generate a hard manufactured good. The major elements of the SCMS are the software version control system, component organization, product environment configuration, product component configuration, workspace management, and product generation.

There are 6 different component categorizations:

- *Platform Dependent* components are those logical entities which bind the hardware abstraction layer to the target hardware platform. There are a set for each handheld platform. These components are completely application independent and are typically managed and released independent of any planned product.

- *System Infrastructure* components are both platform and application independent. Operating System, Event Management, and User Interface services fall in this component category.

- *Application infrastructure* components are very similar to System Infrastructure components in that they are platform and application independent however they are application centric.

"Middleware" such as data, Panel, Service, Service Module, and Application management and support services are categorized as such.

- *Application* components are common to all applications while *Application Dependent* components (e.g. address management) are specific to a region, country, operating center, or user. However both encompass the elements of the Application Modularization Framework -- Applications, Service Modules, Services, and Panels.

- *Language Dependent* components are the externalized string index and text files associated with an Application or Application Dependent component. This is a key distinction. To support our plug and play mix and match strategy, applications no longer "own" language files, components own them. Where in the monolithic legacy application a translation was required for the entire application upon each release to create the language resource file, the translation effort here is on a component basis where the language resource file is generated automatically as a function of the product environment and component configurations.

The product environment configuration specifies the platform, tools, transformation rules, country, and language for a given product. Before that product is generated, the product environment configuration drives the selection of the Platform Dependent components, precompiles all of the Language Dependent components associated with the Application and Application Dependent components specified in the product component configuration, and defines the transformation rules to generate the product. Finally it launches the product level make.

The product component organization defines the components and hence source, header, language, configuration, and make files that define the generated product. The following is a sample portion of a product component configuration file:

```
COMP1_VER       = 19941003
COMP2_VER       = 19960404
COMP3_VER       = 19930923
COMP4_VER       = WIP
...
```

Note that instead of specifying source or application version labels, this product will be made up of separately managed and released constituent components. A separate workspace associated with this product configuration is created for each builder of the product so that updates may be made to components which are Work In Process (WIP). The flexibility of this organization allows a product builder to choose to incorporate released and/or beta (WIP) versions of components to meet the development need. In conjunction with the Application Modularization Framework, it is this capability that allows multiple concurrent teams to work in parallel and allows us to target functionality for a region, country, operating center or single user.

The extensive flexibility of the Application Modularization Framework, the DIAD SCMS, and the build, downloadable configuration, initialization, and run time configurabilty options provide the capability to generate, manage, and deploy applications around the globe. However this flexibility

comes at a severe price -- product certification. Herein lies the *raison d'être* behind our comprehensive SQA strategy -- applying the legacy manual certification process to a re-engineered yet unproved, untrusted, and immature software product will lead to disaster since the legacy certification process will have an inherent bias towards the proven, trusted, and mature legacy software product because of the same market pressures that spawned the re-engineering effort in the first place. Furthermore, the combinatorial configuration options across multiple languages and multiple hardware platforms rendered that already cumbersome process completely inadequate. To make the handheld re-engineering effort a sustainable success, test automation had to become an integral part of that same effort.

6 Test Automation

The DIAD is a hermetically sealed handheld data collector which has been ruggedized to withstand extended temperature, shock, and vibration over a commercially available device. Its external inputs are a matrix keyboard, an optical serial port, a signature pad, an integrated barcode reader, and power (for recharging). Its external outputs are an 8x40 liquid crystal display (LCD), the optical serial port, and speaker tone generation.

While similar in external I/O, the two DIAD product families - DIAD I and DIAD II are radically different mechanically and electrically. Each have very different microprocessors, internal peripherals, memory organization and distribution, power management, watchdog logic, glue logic, firmware, key layout, number of keys, and key functions. However, with the architectural re-engineering, the fundamental differences in platform are largely smoothed out by the hardware abstraction layer, operating system, and file system -- the same generated application will run on all three hardware platforms in both product families.

Because of the desire to provide rapid support for concurrent development and test, and since there was general uniformity in I/Os and application behavior, a platform independent test automation strategy was desirable. However, the generated applications had to be certified on each hardware platform. In addition, that testing must be electrically and operationally non-intrusive -- there should be no additional current drain when under test nor should the executing application "know" it is under test. Furthermore, the test platform should be able to test *anything* the DIAD can do in an unattended fashion -- closed-loop data collection and communication activities. Nevertheless, the sharp electro-mechanical differences in the product families and the hermetic design would make closing the loop very difficult. Furthermore there are no commercially available shrinkwrap automated test tools for custom handheld data collectors. We therefore needed a custom DIAD Automated Test Environment (ATE) to fulfill this strategy.

6.1 DIAD Automated Test Architecture

After clarifying the test automation platform requirements and issues, we formed an alliance with a major embedded automated test vendor to create the DIAD ATE platform architecture as shown in figure 1:

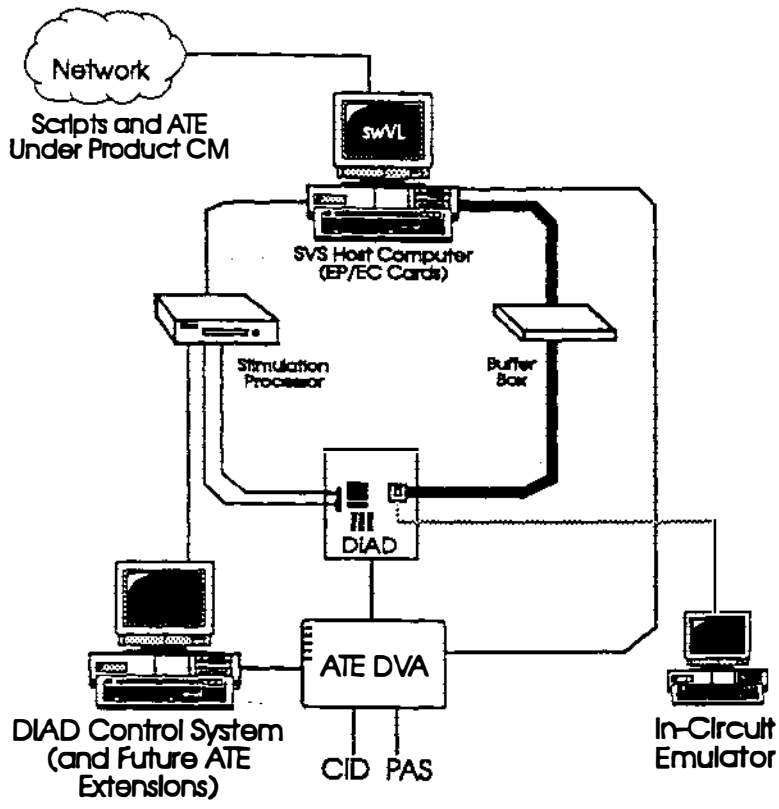


Figure 1 - DIAD Automated Test Architecture

The vendor's Software Verification System (SVS) [6] host computer is the nucleus of the architecture. It encompasses the script creation and execution environment that facilitates fully closed loop unattended testing of the DIAD application. The host interfaces with a custom designed embedded system, the Stimulation Processor (SP), which provides controlled, programmable electronic input stimulation of the DIAD (e.g. key input, signature injection, etc.) through a series of special connectors.

A microprocessor probe monitors all address, data, and control events generated by the processor due to internal and external stimulus. These events are queued in the Buffer Box for retrieval by the Event Capture and Processing Unit (EC/EP) -- a pair of boards that are resident in the host computer. The EC/EP is a special purpose card set with its own CPU, large FIFO and capture/buffering circuitry, I/O, and operating system. It also shares dual-ported memory with the SVS host computer. Event capture encompasses accepting and buffering large bursts of data from the DIAD's microprocessor, filtering raw data and separating events from unwanted data, and timestamping the filtered data and make it ready for processing. Desired events are processed by the Event Processing card to transform them into representations that facilitate "closing the loop." For example, all I/O activity to the display controller is captured by the probe, filtered through the EC, and transformed by the EP into the 8x40 text and attribute planes of the LCD. The state of the display can now be verified against the expected output based on the external

stimulus of the SP all under script control -- full visibility into screen contents and key activity -- and thus provides fully closed loop certification of all user activity. In addition, the SVS system allows script visibility into all raw memory and I/O port read/write events regardless if the are "preprocessed" by the EP. Furthermore, the SP provides simultaneous stimulus (but not monitoring) of the DIAD's host computer to automate the uploading and downloading of DIAD applications, configuration, and data fully under script control.

But as described at the beginning of this paper, the DIAD has evolved from a simple data collector to a integral component in UPS' real time data communications network. To fully close the loop, we need the ability to stimulate and verify the sending and receiving of messages as well. Because of our vendor's open architecture, we were able to customize the system to stimulate and verify all DIAD communications activity under script control:

- Insertions and extractions into and out of the DIAD Vehicle Adaptor (DVA) -- our data radio interface
- Visibility into data sent to the DVA by the DIAD (outbound messages)
- Injection of messages into the DVA destined for the DIAD (inbound messages)

Working closely with our vendor we were able to solve the conundrum of a platform independent test automation on radically different platforms. The SP and EC/EP facilitate the normalization of the sharp differences in the DIAD hardware platforms as the hardware abstraction layer did for the DIAD application. While the SVS host-SP and SVS host-EC/EP interfaces are the same for both DIAD product families, the SP and EC/EP functionality is DIAD product family specific as is the microprocessor probe.

6.2 Automated Test Scripts

The SVS host provides a Windows 3.1-based interpretive script development and execution environment. Low level scripts and SP and EP/EC script interface services are written in swVL™ or Software Verification Language. swVL is an extended "C-like" interpretive language/command set with library extensions. Scripts written in swVL and executed through the SVS interpreter facilitate the issuance of keys, signatures, resets, uploads and downloads by the SP; the execution of test cases, and the rules for comparing actual output provided by the EC/EP against expected output.

However, we significantly extended the swVL execution environment by creating a more DIAD-centric and script developer-friendly execution environment. The approach that was taken was to use Visual Basic to create a script generation, edit, execute, and debug environment and to use 'C' instead of swVL for the majority of actual script parsing and execution. This option provided us faster execution, more flexibility, and easier maintenance. The test scripts themselves are written in DIAD Test Language (DTL) -- a simple yet powerful interpretive scripting dialect that facilitates UI certification, functional verification, data validation, load testing, resource utilization, automated uploads and downloads, extensive logging, internationalization, platform independence, script sequencing and re-use, et. al. While an in-depth discussion of DTL and the extensive underlying support infrastructure is beyond the scope of this paper, it is important to

note that this most visible and utilized component of our test platform represents the culmination of our test automation strategy -- enable software test to be performed *concurrently* with product software development to find problems early in the development cycle. The best times to give active thought to verification and validation of the DIAD functional and architectural requirements -- what criteria we will need to test these requirements -- is while compiling *and* developing them [7].

The vendor platform in conjunction with our value-added components provided a mechanism for automating the testing of DIAD I and II based applications. By accounting for product family differences, the system has enabled engineers to generate re-usable test scripts to validate the DIAD applications generated by the DIAD SCMS. This platform provides fully unattended, automated, and non-intrusive closed loop testing of *anything* the DIAD can do. However, even 100% complete application coverage and certification does not necessarily mean that the product can be signed off as deployable. Because the application's execution is on a resource constrained battery operated handheld data collector we must also ensure that the application is well-behaved in its execution environment.

6.3 Beyond Test Automation

Every time we add functionality we typically worsen our execution performance, power consumption, and memory utilization. We have had on occasion to change application functionality that had been proven 100% correct in the legacy certification process but had either incurred excessive execution overhead, utilized improper power management, or had overutilized memory. It was therefore imperative that we go well beyond simple application correctness in our automated test strategy to also include resource utilization analysis -- CPU, memory, and battery. Using the DIAD ATE platform's capabilities, we are able to proactively monitor execution performance, power consumption, and memory utilization on all facets of the DIAD application independent of the functionality being certified.

Before, only an ethereal balance between functionality, execution performance, battery consumption, and memory utilization existed in the DIAD application. Through understanding usage models and our automated test platform we can now monitor, validate, propose, and converge on new optimal solutions to encompass all four criteria in an iterative fashion *before* the product is deployed.

Our automated test architecture and environment provides additional capability beyond its primary mission and its usage goes beyond test automation personnel. Because the microprocessor probe monitors all processor activity, it is possible for DIAD application developers to perform full code coverage analysis by corroborating "hits" in the memory space of the DIAD when scripts are executing against the locate map of the generated application before software is released for test. Furthermore, the probe design allows for an in-circuit emulator to be used instead of the target microprocessor. Therefore, if a script developer isolates a defect during script execution, the DIAD developer can set hardware breakpoints and monitor application execution to quickly isolate and fix the defect.

The DIAD application conducts extensive logging of system activity to allow event reconstruction in the case of catastrophic failure and to allow for the recovery of critical customer data. However our automated environment now allows us to non-intrusively replay a user's day from those logged events. With this capability we can:

- Determine if defects found in the field are common to the released application, that particular DIAD's software configuration, out-of-spec usage, or a hardware failure

- Generate a global usage model of driver activity to assist in the determination of the optimal balance of functionality, execution performance, battery consumption, and memory utilization.

- Certify new hardware configuration. Our vendors because of parts availability or reliability goals have changed ICs, consolidated ICs into ASICs, and second-sourced major functional components. Through our body of scripts and replayable days, we are able to field test a new hardware configuration before it hits the field.

The DIAD Automated Test Platform and associated scripts facilitate the non-intrusive certification of all DIAD application functionality --both UI and communications centric -- within the context of a resource constrained handheld data collector in an unattended automated fashion.

7 Documentation

In the legacy release-centric paradigm, documentation of DIAD functionality went no further than the functionality associated with a particular release. Therefore to determine the functionality prior to *N* but also incorporated in *N*, one had to look up the functional specification for release *N-1*, *N-2*,etc. The problem is exacerbated with the international releases. Functional specifications for an international release were first developed in English and then sent to a team of translators who would translate that specification into the language of the target market.

A massive consolidation and reconciliation of the functional behavior of radically different application functionality, operating languages, and data collection needs was required. However by moving from a release-centric view to a product-centric view what was once clear and unambiguous was now not only ambiguous but conflicting. Our first attempt at consolidation and reconciliation was the creation of a 400+ page DIAD Behavioral Specification (DBS) for just US domestic functionality. developers. For a short time it helped but serious deficiencies became quickly evident:

- There was no easy method to map the specified functionality to an Application Modularization Framework classification or configurability options. Furthermore the encyclopedic extent and linear organization of the specification made it difficult to use and maintain.

- Certain functionality was germane only to the domestic US -- how would we indicate that this functionality was not to be translated for non-US applications, or translated for a subset of international functionality?

- How do we document the content of multiple releases across the planet that have widely different functionality and operating languages? Even though products would be built from a common component base, how could we maintain a country-specific release-centric view? Moreover, how could we maintain a region-specific view that had multiple translations? How do we perform the translations? How should the test scripts be organized?

The answer was to transform the DBS from an unwieldy, unmaintainable, and unsustainable paper document to a manageable, maintainable, and sustainable electronic resource. The catalyst for this transformation was the UPS Intranet and the World Wide Web. Now, the content of the old DBS could mirror the inventory of Applications, Service Modules, Services, Panels, and the other components created by the architectural re-engineering -- a product-centric release-independent view. But more importantly, the organization of the new DBS could now reflect the release-centric application generated by the DIAD SCMS from that inventory of components. By employing HTML, RDBMS, and other technologies, we are able to tunnel through a document with a region specific view, a language specific view, a country specific view, a platform specific view, and a release specific view. We allow the user to see the deltas between release versions. We are able to organize the common, common configurable, and country specific functionality without polluting the other views. We can "exercise" the document as a user would "exercise" the DIAD application -- thereby evaluating the human factors and functionality of the proposed application organization before one line of code is written. We therefore also provide the context and the environment in which translations can take place.

8 Organization

Before the re-engineering effort, an assignment to test the DIAD application was viewed as potentially career limiting. There was little or no technical challenge, more time spent in manual test execution then in developing test strategy, and no sense of ownership of the product since it was the project teams that drove major design decisions -- not the independent product test group. After re-engineering, we changed the career development stereotype, created the technical challenge, and through automation allowed the script developer to focus on test strategy, not test execution. However, DIAD application, configuration management, test automation, and documentation re-engineering does not necessarily imbue a shared sense of responsibility and ownership.

To increase engineering productivity by enabling small cross-functional development teams to work *concurrently* on the DIAD product, to enable software test to be performed *concurrently* with product software development to find problems earlier in the development cycle, and to create that shared sense of ownership and responsibility demanded that we not only merge the domestic and international project teams but also to merge with the independent test group. From that pool, small teams were assigned to tackle specific DIAD functionality -- typically two application developers and two script developers with no assigned team lead. Task planning and schedule development is performed by the team. Script developers could now re-order the work of application developers to facilitate script development and execution, and vice-versa. Although the primary focus of the team was to tackle a specific business problem, the application and script developers were also to ensure the integrity of their individual product lines and product support

infrastructure. These small teams are at once both product-centric and release-centric -- addressing the needs of the marketplace without straining the architecture and addressing the needs of the architecture without shortcoming the marketplace.

9 The Re-Engineered Development Process

The conglomeration of these pillars of the comprehensive SQA strategy comes to fruition in the re-engineered development process as shown in Figure 2:

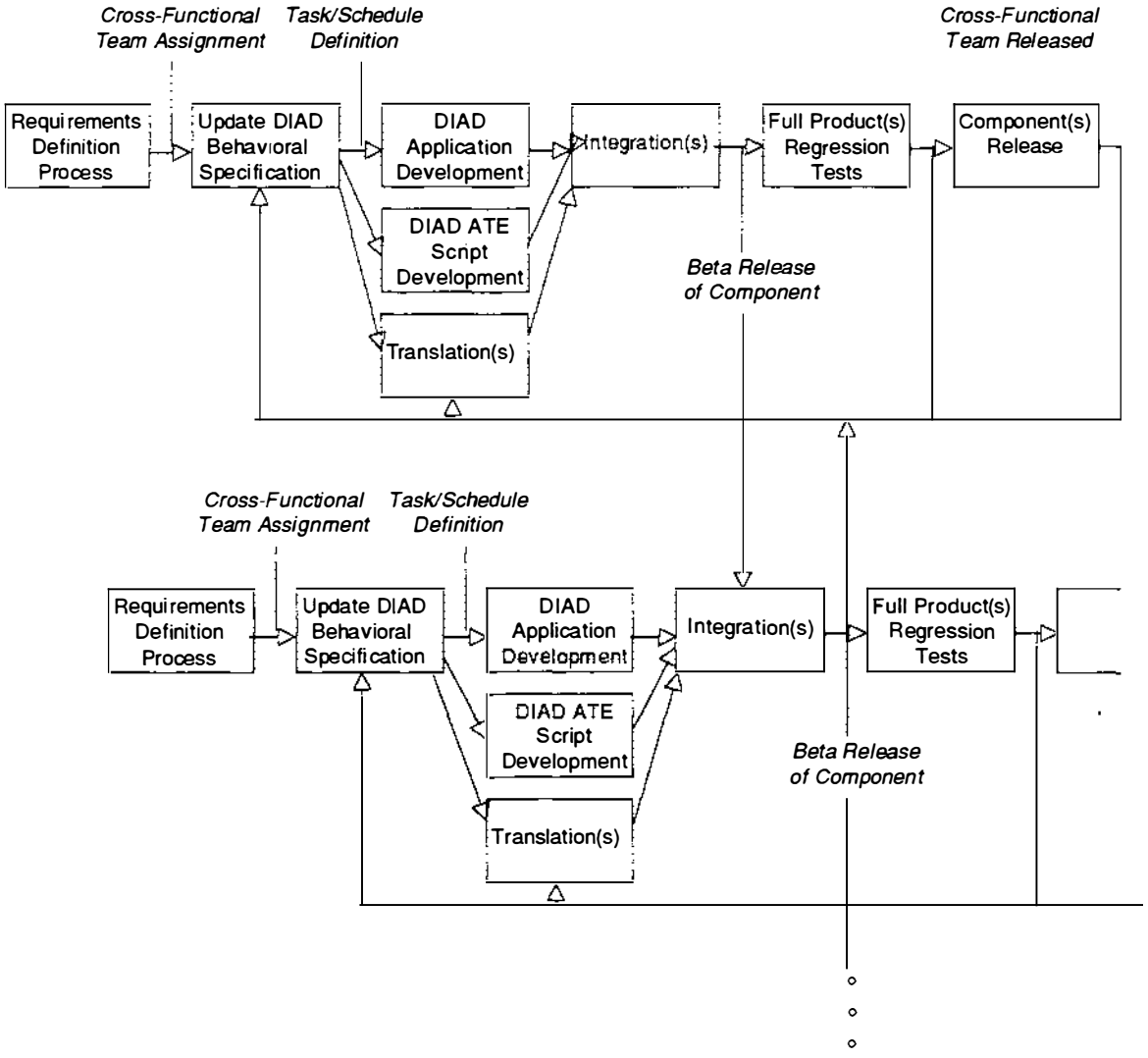


Figure 2 - Re-Engineered Development Process

No longer do the requirements from dissimilar business functions and the associated specifications have to be codified for a particular release before development can begin. As figure 2 shows, multiple concurrent tracks are now available -- the only limit to support a near random arrival rate of business requirements is staffing. The already strong requirements definition phase is now

enhanced to encompass both domestic and international analysis -- to define what target markets, operating languages, platforms, et. al. the resultant features, products, and services will impact.

Once approved, a cross-functional team of DIAD application and ATE script developers are assigned. Based on the scope of the requirements, potential Applications, Service Modules, Services, and Panels are identified. Data collection needs are defined. An impact assessment on the existing inventory of components and scripts is made. Configurability requirements are assessed. The DBS is then updated on-line but the associated functionality is in an electronic WIP quarantine -- it is not yet tied to any one release. Once the modularization framework, impact assessment, desired configurability, and DBS entries are reviewed and approved, the application/script design, development, and translation effort is defined and scheduled. Once accepted, the DIAD SCMS is employed to create the workspace for the effort and manage the affected component archives. The application, script, and translation development effort can then begin.

Every effort is made by the cross functional team to schedule an appropriate number of early integrations of DIAD and ATE functionality as a sanity check on the preceding phases, to perform partial automated regression tests of the affected components, and to release functionality to the entire product team to facilitate an in-house parallel beta test. The other development tracks may or may not integrate the functionality into their workspace dependent on the orthogonality of the efforts and/or the associated risk.

Once the functionality has been developed, reviewed, and certified under automated test, DIAD application product(s) for the identified target markets, languages, and configuration are generated via the SCMS. Each of these products go through the battery of automated tests to certify the product -- acceptance, regression, execution performance, battery consumption, memory utilization, et. al. Once passed, the constituent components but *not the generated applications* are released and the SCMS component inventory updated. The associated DBS entries are then moved out of quarantine but are still not associated with a particular release.

A product release encompasses the same certification effort as a component release except that the product encompasses the *entirety* of completed constituent components for a particular target market, language, and configuration at a desired time. Once released, the product's component configuration is archived in the SCMS and the DBS content is updated to allow navigation through the various views.

10 Summary

It is important to note that the creation of this comprehensive SQA paradigm and the subsequent sweeping re-engineering effort required intensive planning, capitalization, staffing, and foresight. Certain elements, such as the handheld re-engineering effort, took longer than planned. UPS has also had to support both the legacy product/process and the re-engineered product/process as it transitioned from one paradigm to another. But because of the power of this new paradigm, a great deal more flexibility and entrepeneurship is now permitted in the decision to define, develop, release, and deploy DIAD based products and services. Instead of being driven by the serialized

restrictions of the legacy architecture and process, release decisions are driven by more immediate customer satisfaction, volume development. and profit potential. The investment in architecture, configuration management, test automation, documentation, organization and development process has yielded a sustainable enterprise that will not only support our installed base but expansion as well -- whether in new products and services, operating languages, or even hardware platforms. UPS is now more nimble in its ability to rapidly respond to changing market conditions worldwide.

11 References

1. Arnold, R.S. *Software Reengineering* , IEEE Computer Society Press, 1993.
2. Pressman, R. *Software Engineering: A Practitioners Approach,* 2d ed., McGraw-Hill, 1987.
3. Hutchings, A., and Knox, S. "Creating Products Customers Demand," *Commun. ACM 38*, 5 (May 1995). 72-80.
4. Hammer, M., and Champy, J. *Re-engineering the Corporation*, Harper-Collins, 1993
5. Henninger, S. "Supporting the Construction and Evolution of Component Repositories," *Proceedings of ICSE-18,* March 1996.
6. *swVL™ Introduction and Overview.* BTree Verification Systems, Inc. 1994
7. Hatley, D. and Pirbhai, I. *Strategies for Real-Time System Specification*, Dorset House, 1988.

About the Author:

Bernard E. Gracy, Jr. is UPS' DIAD Product Manager. He received his BSE in Computer Science and Engineering from the University of Connecticut in 1985 and his MSCS from Rensselaer Polytechnic Institute in 1993. He is a member of the ACM - SIGOPS and SIGSOFT.

Exploring the Expanding Frontiers of the Software Beta Process

Sridhar Ranganathan
Manager, Beta Programs
Pure Software
1309 S. Mary Avenue
Sunnyvale, CA 94087

Phone- (408) 524-3654
Fax - (408) 720-0548
E-mail- sri@pure.com

Keywords:

Beta - Automated Software Quality - Pure Software - PureVision - Competitive Advantage - testing - verification - defects - information - cross-functional - software - deployment - feedback

Biographical Sketch:

Sridhar Ranganathan has over six years of experience in the software industry. He is currently employed as the Manager for Beta Program at Pure Software Inc., a leader in Automated Software Quality. Prior to joining Pure Software Inc., Sridhar was employed as a Senior Software Engineer at Xerox Corporation.

Sridhar graduated from UC Berkeley's Haas School of Business' MBA program in May 1996. He also holds a MS in Computer Science from Villanova University, PA and a BS in Computer Engineering from the University of Bombay, India.

Intended Audience:

This paper will appeal to both the technical and managerial personnel in a software business. The focus of this paper is directed primarily at the software product development, testing and support teams. The underlying ideas and concepts may be stretched to address similar issues in other businesses as well.

1.0 Abstract

This paper will endeavor to elaborate on the process and information flow and identify the stakeholders and their roles during a software beta process. The software business community is keenly aware of the need to do multiple pilot programs for a software product, before it is made available to end-users. This stems from the fact that it is typically not practical to complete all the multitudes of possible test combinations, in-house. These pilot programs are typically labeled as alpha, beta or gamma releases, where the software is sent to a select few customers, first in known environments and gradually eased into the unknown. (Clear definitions and distinctions will be provided for each of these phases in the paper. For the sake of discussion in the abstract, all the phases are collectively referred to as the beta process.) The intent during the beta process is to understand the potential of the product - to identify the strengths and weaknesses early enough - to create an opportunity to rectify the weaknesses and enhance the strengths.

Before the beginning of the beta process, the product is usually a completely functional product. The beta process helps the customers get early access to the software. It sets the stage for customers to evaluate the product for its features and their own needs for its functionality in their business. Conversely, the beta process provides an opportunity for the software producer to test-market the product and gauge the reliability and usefulness of new features. For instance - the demand for the product can be gauged by the number of customers willing to participate in the beta program. The beta process has grown in its significance to influence critical business decisions in the complex, growing and fiercely competitive software industry. Internal testing and verification, once thought to be the key in producing quality software is simply not sufficient. The growing inter-dependency of software packages from different vendors makes it absolutely essential to test any software product in as many diverse environments as possible. For a business to acquire all these diverse test platforms is not only unrealistic, but least to say, simply not cost effective. The same goal can be achieved by carefully soliciting and selecting the customers for the beta program.

The goals of a beta program can vary depending on the size of the company, the maturity level of the software product, the demand for the product, the reputation of the product, the market size, the demographic mix of the customer base, among other factors. The complex software marketplace, with its ever-increasing competitive pressures lends another dimension to the beta process. The feedback required of a beta program is usually very cross-functional, for instance - Engineering wants to know if the defects are solved, Marketing wants to know if the product is ready for release, Sales wants to know the demand for the product. This makes the goal-setting process a very important stage in the beta cycle. Regardless, accurate and timely information is the key to a successful beta program and subsequently, a successful beta program is very essential for the success of a product. From the competitive angle, the timely and accurate information during the beta program can create significant competitive advantage. Facilitating and managing the data flow between the various cross-functional departments and the beta customers becomes very important in shaping the future success of the product and the business.

In establishing the need for a well managed beta process, the paper will set stage to introduce an important concept in Automated Software Quality (ASQ) and the use of software quality tools during the process. The discussion of both ASQ and the tools primarily focus on the

need to address the serious lack of accurate and timely information flow during the beta process. The goals of ASQ, in its purest form, is simply to automate the necessary quality processes, to ensure consistent and standardized use, to help businesses achieve competitive advantage. Quality processes have evolved over the years, but are usually time consuming or inconsistent due to lack of tools. While the software industry, in its growth phase, primarily focused on bringing technological advances to the mainstream customers, it neglected to address quality issues. As the technology matures, more attention is now paid to these issues. ASQ allows businesses to align their processes to their goals in achieving superior quality for their products. The tools used during the beta process are - a defect tracking tool to help understand the readiness of the product, a call tracking tool to capture, log, understand customer feedback and PureVision, a remote run-tracking tool that captures all essential elements of the runs at the beta customer site and automatically sends it back to the software developer.

Effective management of the information flow is very essential for the success of the beta program. The feedback coming in from the beta customers has to be timely and reliable to make decisions for projecting the future of the product and, hence, the business. Currently, various personnel from various functional departments contact a beta customer for specific information. Synergies are lost due to this random and arbitrary collection of data. More so, it is often forgotten that by being in a beta program, customers are providing a valuable service and it is very essential to be sensitive to their time constraints. A solution to these problems would mean that the information should flow unobstructed and seamlessly from the customer to the appropriate functional teams. This solution should also allow various teams to look at different slices and combinations of data, to have access to the complete picture, rather than getting pigeon-holed to a certain cross-section of data. This solution should be automatic without the need to initiate any contact with the customer, other than signing them up in the beta program.

The paper will include results from the various experiments that have been done here at Pure Software Inc. These experiments used the remote run tracking tool (PureVision) in the beta programs of other products. The quality of feedback was immediately apparent, but the real value was seen when PureVision, in combination with the other tools, was able to present the incoming data in an easy and graphical manner. However, this paper is not about PureVision, but about the beta process. The main theme is to understand the need to automate, the need to exploit the information received during the beta process and gain competitive advantage.

2.0 Scope

The word ***beta*** is frequently used in the software industry, however, it is always not used in the same context. There is no single definition for this word and hence, it becomes quite necessary to define it, and understand the context in which it is used in this paper. Prior to defining beta, it is necessary to understand the software development process.

2.1 Designing, Development and Release of Software at Pure Software Inc.

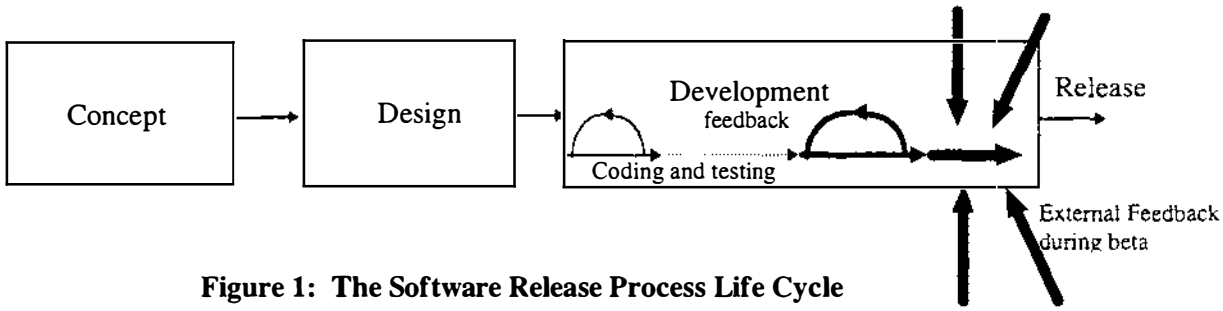


Figure 1: The Software Release Process Life Cycle

Software products are created by converting an idea into a concept, where it is stated it in terms of features and functionality that could be used by potential customers. The concept is then debated and refined to understand the implications of the product in the market place, which leads to the design phase. In the design phase, the requirements are separated in logical components and the pieces are put together in the most efficient manner possible. At the completion of the design phase, a functional specification document is written which helps the software team understand all the logical strands that connect the different pieces. The software team then proceeds to develop the software. Development of software is the process of writing and testing the code to ensure that the end product complies to all the requirements specified during design.

Testing of software, in its most elementary form, begins during the concept and design phases. During these phases, the underlying ideas and the various methods of implementations may be refined. However, software testing, as commonly known in the industry, begins only after a fair amount of code is written. Blocks of code are written and tested in an incremental fashion, before they are joined together to complete the software product. Figure 1 depicts the feedback stages during development. The increasing thickness of the feedback arcs signify the increasing complexity of the content in the feedback information and the increasing difficulty in incorporating the feedback information into the software product. This process of coding and testing is repeated until the software starts to resemble the desired end product. At this stage, internal testing has usually outlived its usefulness. Most software products are required to work in tandem with other software, in various hardware environments. Sometimes, they are required to be tailored and optimized for various business situations. Given the demands of the marketplace, in almost all cases, software products require external testing under a variety of diverse conditions.

2.2 Beta Testing at Pure Software Inc.

Alpha and beta testing of software products are the various stages of external testing. For the scope of this paper, the following definitions are used

Table 1: Release Definitions

Release Milestones	Definition	Target Customers
Alpha	• All feature integration complete. • Test in largely know environments. • Some previously unknown functionality may be included. • All changes to code require communication to project team and documentation of the changes are required to be made in the bug tracking system	Friends and Family - include customers with long standing relationships, include partners where the interdependency of product lines are important to mutual customer base.
Beta	• Fully functional as specified in functional requirements document. • Test in real and unknown environments. • Identify and target these environments based on earlier feedback. • Include environments that have evolved in the fast moving software marketplace. • All changes require peer code review. • Include all deployment functions to ensure smooth delivery of software at release.	Full internal deployment. Customers selected to ensure wide coverage of unknown environments. All alpha test sites to be included to verify changes made during alpha.

It is already quite apparent that the amount and the quality of ***feedback*** information during the alpha and beta stages of product development is very important. Collecting feedback during the alpha stage is not very difficult due to the nature of the targeted customer base. Not only that they are friends and family and hence, have close relationships, but the targeted customer base during alpha is also quite small. However, the scenario changes quite drastically as the product enters beta. Feedback becomes more critical and at the same time it becomes increasingly difficult to get it. The customer base gets to be quite diverse and it takes a fair amount of proactive follow-ups to get the required information. Some of the important issues at this stage that directly affect the quality of feedback and hence the effectiveness of the beta program are:

- identifying the customer base (can't be too many, yet needs to be quite diverse)
- timeliness and content of feedback
- focus of feedback
- cost effectiveness of feedback received

3.0 Introduction

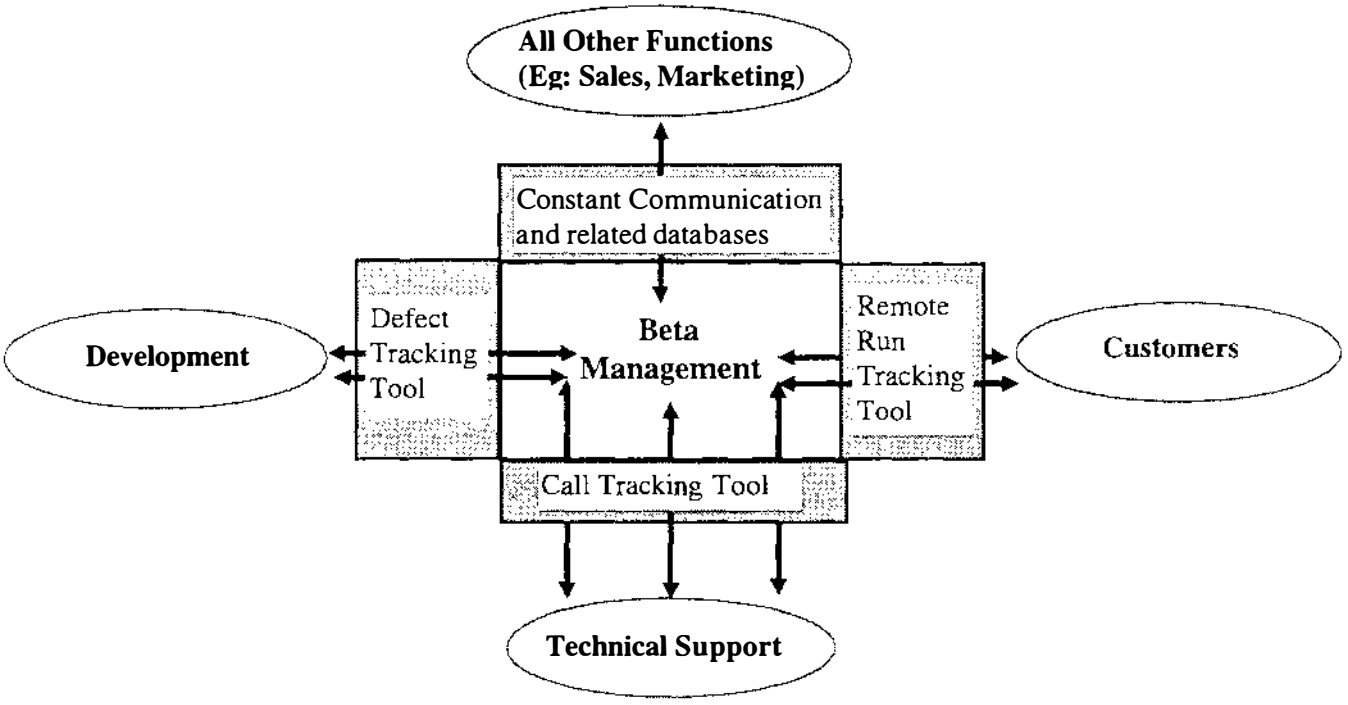
3.1 Automated Software Quality

The underlying theme of this paper is about the use of automated tools that help in streamlining these processes. The beta process is one among the various processes in a software industry that is of vital importance to ensure the success of a product in the highly competitive marketplace, but the general framework may be used to redefine/reengineer other processes, as well.

The purpose of beta testing is to get the product out to customers and get useful feedback that could be incorporated in the product before release. In most cases, the customers are not very well identified, the feedback is not solicited with any degree of focus and the beta testing process usually becomes a very loosely connected series of events, sometimes even chaotic. Like any process, beta testing involves interaction between people, procedures and tools. These tools are referred to as the ASQ tool kit.

Figure 2 shows the interacting functional organizations and the automated tools used to communicate and facilitate the process. The tools and the process will be described in more detail in the next section.

Figure 2: Process Interactions and use of automated tools



4.0 Beta process

4.1 Traditional Model

Based on either the state of the software or the need for release of the product, the product is sent out to several beta sites. These beta sites are usually selected by

- the software engineers developing the software. Consequently, these sites are usually their contacts, personal and professional. Not much thought is given to the usefulness of the product in their environment. Feedback is quite limited as beta software is usually perceived to be unstable and hence, there is a reluctance to use it in the intended environment.
- the product marketing manager or the person responsible for the financial accountability of the product. The beta sites selected are usually existing customers from other related product lines or potential customer contacts made at trade shows and other marketing events. Not much thought is usually given to the technical viability or capability of the beta site.

Beta software is sent out, occasional follow-up is attempted and the beta feedback is completely driven by the beta sites and the entire process ends up being very reactive in nature. This model can suffer from the following problems:

- Insufficient feedback due to the lack of diversity in beta sites.
- Inability to send significant feedback due to lack of need for the product.
- Timeliness of the feedback.
- Deployment tasks are not involved until after the beta is "successfully" completed.

This chaotic process can best be described as "shot in the dark". The feedback information is either not adequate or not focused enough to make any decisions.

4.2 Streamlining the Traditional Model

4.2.1 Philosophy

The intent to streamline the beta process is

- to increase the usefulness of feedback information. For example: to be able to identify and prioritize the problem areas in the product.
- to get the feedback information in a timely manner.
- to ensure incremental improvement in the quality of the product after every beta release.
- to be able to make informed decisions that affect product release.
- to ensure product delivery organizations are well connected and ready to handle the release of the product.

Feedback information is considered to be the key during the beta program. The product team and the beta management personnel ensure all internal issues to be resolved before a product enters the beta testing phase. The following sections describe the actual process that is used at Pure Software Inc.

4.2.2 The People

The beta programs are managed by the Beta Management team. The team consists of a Beta Manager, who reports to the vice-president of World Wide Technical Support and 2 beta administrators. The team is involved in project life cycle planning even before the product reaches the beta stage. Access to the product team and the product well in advance of the beta program helps the beta management personnel to get trained and pre-plan the profile of a beta customer.

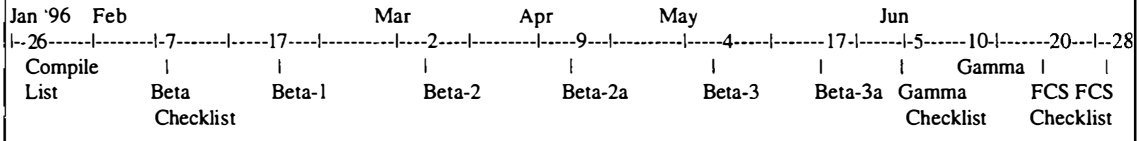
4.2.3 Process Description

Beta Plan for Purify

Goals:
* Ensure backward compatibility with Solaris 2.4 and below
* Bug Fixes, Solaris 2 5 support & PureDDTS Integration
* Ensure all internal functions are in place for product deployment.

Specific Testing Coverage Goals:
+ Verify bug fixes for customers with <previous product release> problems.
+ Test PureVision + Test Japanese Localization (Need feedback)
+ 3 DCE customers, Solaris threads + Sites that have Purify and PureDDTS.

Beta Schedule



Overall Testing Coverage
Operating Systems:
+ 20 Customers on Solaris 2.5
+ 20 Customers on SunOS/ Solaris 2.4 and below (Ensure backward compatibility)

Number of Customers
Targeted 50
Have 15

Beta Tasks:

Pre Beta

Who	Due	What
SR	1/29/96	Solicit Customer Sites
PR	1/29/96	Provide additional key sites
SL	1/29/96	Update targeted Problem Reports for this release.
AA	2/4/96	Send Tester profile. Receive environment. configuration from installed sites
AA	2/7/96	Create customer/environment matrix, Create and update beta site list
RW	2/7/96	Beta Checklist

During Beta

SR	Weekly	Follow ups on defects, Beta Report
RW	6/5/96	Gamma Checklist
AA	6/13/96	Send Beta Survey
AA	6/18/96	Survey results consolidated
SR	6/20/96	Final Beta Report
RW	6/20/96	FCS Checklist

Post Beta

AA	6/18/96	Thank you notes
AA	6/28/96	Upgrade notice to beta customers (includes all customers who have reported bugs in the previous release) First Customer Shop (FCS)

The beta plan contains the details of the necessary components that are required to ensure comprehensive testing in unknown environments. The targeted customer base is explicitly stated and the beta tasks are well defined. Responsible personnel for these beta tasks are identified and the target completion dates of these beta tasks are specified. See Figure 3 for a sample beta plan. The beta plan signals the start of the beta process and summarizes the steps that define the beta process.

4.2.4 Customer Solicitation

The first task of the beta process is to find the right mix of beta customers. This task is very important to ensure the success of the beta program. Beta customer sites are identified and solicited to fill a particular need during the beta program. Table 2 lists the types of customers and their purpose during the beta program.

Table 2: Beta Customers

Target Customers	Their purpose during beta
Verification Customers	Customers who have reported problems with the earlier relea es of the product. These customers provide feedback on verification of the defects that are fixed in the current release.
New Customers	Potential customers identified either by the sales team or potential customers with diverse operating environments. These customers provide feedback on the stability of the product in new, diverse environments.
Development Partners	Other software vendors who may sell to common customer base. If the software products are used in the same environment, it is essential to test the interdependency between the products.
Loyal Customers/ Fans of the product	Customers who absolutely have to have the product and have a distinct liking for the product. These customers usually are the first ones to install the product and provide feedback. These customers also are the easiest to solicit, as they ask for the product.

The solicitation letter contains an invitation to the beta program and includes a questionnaire asking the prospective beta customers for the details on their environment configuration. An environment profile is developed as the replies come in. This report helps cross reference the beta sites with the coverage goals, as stated in the beta plan.

See Table 3 for the customer environment profile. This information is very useful in understanding both the coverage of important factors and the need for focus in testing specific environments.

Table 3: Beta Customer Environment Profile

EMail	Operating System		Compilers				Debugger				
	Solaris 2.x (2/3/4/5)	SunOS 4.1.x (1/2/3/4)	SWx.0(2/3)	gcc/g ++	acc	/bin/ cc	xbd	dde	dbx	gdb	
Customer 1	2	4	3			x			x		
Customer 2	2	4.1.3_U1, 4.1.4	3			x	x	x			
Customer n		4	2						x	x	

4.2.5 Beta Feedback

The customer environment profile helps in qualifying beta sites per beta release. It also helps in targeting resources towards select customer sites for gathering feedback. Feedback obtained is communicated to the product team by publishing a weekly beta report. Figure 4 shows a sample weekly beta report.

Figure 4: Sample Beta Report

	5/22	5/29	6/6
Beta-3 sent to	295	313	432
Total sites who have installed	80	120	192

PureVision data.

Total runs since Beta-3a	4388 runs
Total crashes	309 crashes
Crash Type 1	118
Crash Type 2	191

Active Sites per day (on PV)

5/17	34	5/24	26	5/31	21
5/18	7	5/25	2	6/1	2
5/19	4	5/26	1	6/2	3
5/20	34	5/27	6	6/3	22
5/21	28	5/28	24	6/4	23
5/22	40	5/29	21	6/5	35
5/23	32	5/30	22	6/6	18

Defects' list

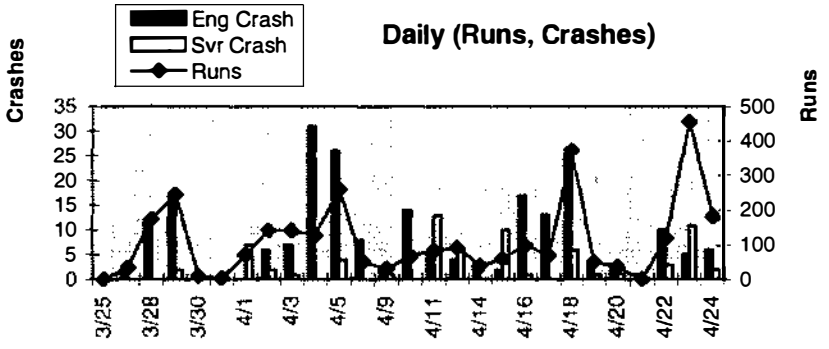
open **Must** Problem Reports

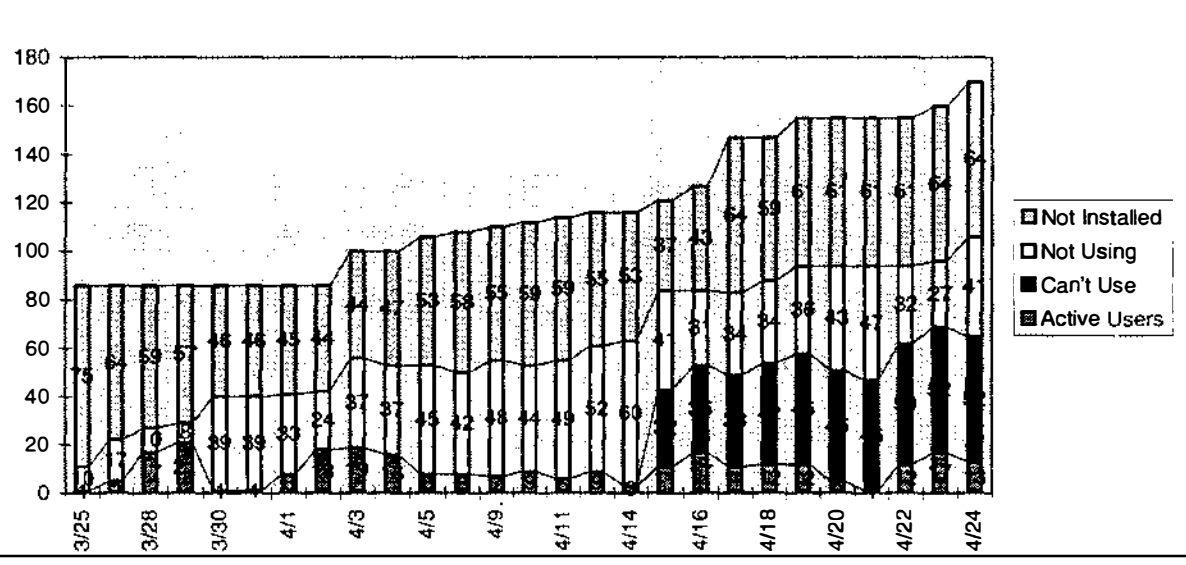
Identifier	Engineer	Platform	OS Version	Description
PSIps19536	lanning	solaris	2.5	window crashes upon entry
PSIps11119	fulmer	solaris	2.4	instruction relocation incorrect
.....				

Problem Reports to be verified

(Resolved between release of previous version 04/15/95 and 10/10/95)

Identifier	Platform	Customer	Date	Description
PSIps00001	solaris	custl @company1	960127	mainwin program crashes
PSIps00002	multiple	cust2@company2	960210	copy sequence failure due
.....				





The data presented in the beta report is gathered from reports generated by tools used by the interacting organizations. See Figure 2 for data flow between organizations. The beta site information is gathered from the remote run tracking tool (data from the customers), the defect list information is collected from the defect tracking tool (database for the development team) and the qualitative information about the runs are collected from the call tracking (data from the support organization) where the customers calls or remote runs that result in abnormal exit are logged.

4.2.6 Data flow

The defect tracking tool is used to log defects as found in internal testing and as reported by customers using the previous version of the product. The defects are prioritized into ***must***, ***should***, ***could*** and ***not*** categories. A product does not enter beta, until all the ***must*** defects are resolved. Consequently, a product in beta is not released until all the ***must*** defects are resolved.

When the product enters beta, the remote run tracking mechanism provided by the remote run tracking tool is enabled. This allows the product to collect and transmit data regarding each run back to the product team. The beta team uses this information to record the following statistical information

- # of customers, # of active customers
- # of runs, per customer
- # of crashes per site, per beta version

This information combined with the information collected during the solicitation process provides a detailed insight on the usage of the product in beta.

The beta team then proceeds to interpret the data and infer qualitative information. Some of the qualitative information that are useful during beta are:

- a large percentage of beta sites active would indicate that the product has high demand in the marketplace.
- testing coverage details can be inferred by cross-referencing the environment grid of the active beta site and the run tracking data transmitted.
- the run tracking tool also returns all state information that occurred during a run. If the run happened to terminate abnormally, the state information would help in duplicating the error and resolving the issue.
- the state information also helps in categorizing the abnormal exits (defects). This helps in prioritizing the defects depending on either the number of beta sites reporting the error or the severity of the error.
- For a new product in beta, if the information provided indicates a high demand, it would be worthwhile to pass the information to the sales and marketing organization. If the product is sold during beta, identifying demand may help in forecasting revenues.

The call tracking database logs all customer reports and errors reported by the run tracking tool. These reports provide a great deal of insight in analyzing, debugging and resolving the defects.

5.0 Results

Some of the experiments conducted during several beta programs, using the above outlined beta process revealed some extraordinary gains. These are summarized in two main categories:

5.1 Timeliness

The turnaround time for customer feedback improved from an average of 1-3 days to a few minutes. The remote run tracking tool transmitted the details of the beta site and the runs of the product in beta as soon as the run was completed at the beta site. More importantly, it always gave an accurate number of active sites using the beta product.

5.2 Cost Savings

To understand the effectiveness of the tools that help automate the beta process, an estimate of the expenses incurred without these tools is presented.

Assumptions

- 400 beta sites
- 4 beta releases
- $2 for each follow-up via phone or fax
- 4 follow-ups for each beta

The total expenditure for just the follow-ups is $12,800. This does not take into account the other hidden costs in the traditional approach, such as:

- feedback may not be timely.
- feedback received is not accurate or complete, resulting in the defect not getting resolved.
- no accurate count of beta sites, runs or crashes.

These hidden costs are bound to impact the quality and time to market of the product in beta. The benefits of using these tools in the beta process are quite apparent. However, these tools are not quite useful without a process in place. Collecting, interpreting and presenting the data plays a very important role in making informed release decisions for the product in beta.

6.0 Conclusion

Streamlining the beta process at Pure Software Inc., using the methodology discussed in this paper, has proved to be very successful. The various organizations that participate in the beta process have acknowledged the improvements. In the past, the technical support personnel used to manage the beta process and the creation of a separate beta organization has taken away a lot of unrelated tasks from them, leaving them focused in their customer support function. The development team is left to focus on the technical details of the product, while the beta management personnel receive and manage the feedback information during the beta period. The testing team is not frustrated in its inability to test all possible combinations. Apart from that, this new process has brought about a sense of ownership to different pieces of information that is critical during the deployment process of a product. Various related databases are now connected. Information flow and exchange has become more standardized.

The ultimate winner has been the customer. Beta announcements are usually made by sending an electronic mail. The announcements originate from beta management personnel and the customers have a single contact point for all beta related issues. The beta management personnel act as the front line contact for beta customers. However, the expertise of other organizations is used to ensure both quick and accurate responses. The benefit for the customer is a single point of contact and clear communication. The schedule of the release process is communicated well in advance and all the important milestones are highlighted. The beta customer is often contacted by Pure Software when the run data shows any discrepancy. This is in contrast to the customer making frantic calls to technical support. Customer relationships have improved tremendously over the last year. One indicator has been the post-beta survey. The surveys being returned have exceeded 20% of the total beta customer base and is increasing. The responses to the questions in the survey has had a positive trend, not only for the beta program or the product, but also for related areas such as documentation and technical support.

The primary reason for the success of this process is the use of automated tools. The various tracking systems used in identifying, managing and interpreting the feedback information have added immense value to the process. The defect tracking, call tracking and the run tracking tools help facilitate the information flow between the organizations to be able to make well informed decisions that are critical to the success of a product. The defect tracking and call tracking tools helps in assessing the quality of the product prior to, during and after the beta period. The run tracking tool is used only during the beta period and it helps in linking the data from other tools and customers. From useful debug information that helps in fixing defects to valuable information about the installed beta customer base, the run tracking tool provides the backbone to the infrastructure developed by this new process. Finally, the process has fostered a great environment for building cross-functional product teams. This process has a lot of hand-offs and functional teams rely on each other for communicating related events. The automation has achieved these critical events to seamlessly fit into the cogs of the day to day business events.

7.0 Bibliography

1. Pure Software Product Bulletin, PureVision on Unix
2. Pure Software White Paper, Remote monitoring using PureVision
3. Pure Software Product Life Cycle, Internal Presentation
4. Regis McKenna, Marketing Tools, Talk about leveraging the beta process to successfully market technology/products.

Incremental Process Improvement

Rick Clements
FLIR Systems, Inc.
16505 S.W. 72nd Ave.
Portland, OR 97224
rclement@flir.com

Abstract

Many new companies find themselves in the same position our team was in. They have been developing a good product but their success makes change necessary. Success means more products need to be developed in less time.

This paper looks at the process of moving from an ad-hoc testing process to a process to a defined process. In an ad-hoc process, there isn't time to collect data for metrics. How can the process be improved without data or metrics? The good news is that since there are big problems the problems are easy to identify.

Key words and phrases: process improvement, software / firmware testing, getting started, embedded real-time systems

Biographical Information

Rick Clements is a Software Quality Assurance (SQA) Engineer with 17 years of experience in the software industry. He is currently an SQA engineer at FLIR Systems, Inc.. He has 7 years in SQA at the Color Printing and Imaging Division of Tektronix. In addition to SQA, he has worked in real-time systems development, device drivers and software to test hardware. He has a B.S. in Computer Engineering from The University of Michigan.

Summary

The initial process was an ad-hoc testing procedure. There was continuous testing during the duration of the project. The test cycles were about two weeks long.

The first improvement was written test procedures in the form of a Test Item Specification (TIS). This resulted in being able to test the printer in a single week leaving a week for the vendor to fix defects. It also made defect easier to recreate. It has the long term advantage of being able to reuse some of the tests.

The next improvement was up front planning. Up front planning came in two steps. The first was test plans. Test plans allowed planning of resources. Before tests were written, we could plan what areas would be tested, what areas would not be tested and the risks involved. This allow management to accept the risks or provide additional resources.

The other up front planning tool came later. That was the work break down structure. The work breakdown structure (WBS) described each task, how long it would take, the resources required, other tasks it depended on and how to tell when it was completed. This detail of planning was important with multiple overlapping projects. It allowed the essential tasks to be completed on the first project. Then, the resources could be shifted to the second project to complete the essential tasks. The resources could then be moved back to the first project to give better coverage by completing additional tasks on the first project.

The better planning allowed better use of personnel. This was also an incremental improvement. First, test fixture operators were brought into run tests. This freed engineers to do more planning and test design on following projects. As the process became better defined and the test operators became more experienced they could take on additional tasks. Several of the operators became test technicians. These technicians became capable of analyzing tests logs, test output, submit some defects and flag other defects for the engineer to analyze. They also began modify some of the manual test procedures under an engineers guidance.

There were mistakes made. One mistake was moving the writing and maintenance of the interface specification out of the QA group. The QA group spent more effort in getting enough detail to be testable and in keeping the specification up to date. Fortunately, changes to the specification were recorded in FAX's until the specification was updated. FAX's are a benefit of working with an outside vendor.

Introduction

This paper takes an chronological description of the process improvement. This is done to highlight the fact the improvement was incremental and not a single step.

This division of Tektronix produces color printers for the office, commercial artist and engineering environment. Part of the print engines are developed by other companies. The print engine is part of the system that puts marks on paper. The team discussed in this paper tests the software that controls the real time processes in the print engine.

Each version of software arrives in the from of firmware burned into PROM's. The software is refereed to as firmware because new software can't be shipped to the customer on a floppy if a problem is found after it's released.

This paper focuses on the team testing the print engines from OEM vendors. Their approach was somewhat different than the team testing internally developed print engines. For example, the team testing the internal print engine firmware could get access to sensors and could talk with the design engineers but the specifications they got were less complete. A description of the internal print engine firmware team's approach can be found in "Mother2 and Moss: Automated Test Generation from Real-time Requirements" by Joe Maybee in the 1993 proceedings of the Pacific Northwest Software Quality Conference.

Ad-hoc Testing

The printers were doing well very well against the competition. This success brought two problems. First, selling more printers made it too expensive to send a technician to each customer to replace the PROM's containing firmware. Second, more products needed to be tested more quickly.

The process, at this time, was to push a print engine into an engineer's cubical. Testing would continue until a new firmware release was ready in two weeks. Nine months later you have a brand new printer product. This engineer wrote the interface specification, interfaced with the vendor and did the testing. No time as allocated to write the tests before the printer arrived. This meant all the tests was done ad-hoc.

Ad-hoc testing has a number of problems. One problem, it's sometimes hard to recreate failures. Without the testing sequence written down, it's easy to leave out a step that got you to that failure. A second problem is there is no test procedure to build from for the next printer. A third problem is with no

preplanning, you can't optimize the testing strategy. This made for a very long testing process.

Define a process

This long testing cycle was something that the manager very much wanted to change. The question was how. There was never time to improve the process. In fact, there wasn't an organized process. Most of the team were mechanical engineers responsible for evaluating potential print engines in different technologies. There was only person with software testing experience in the team. He was too busy testing to improve the process.

During a re-organization, the print engine team and system software QA teams came under the same manager. He added half an engineer from the QA team to the engine firmware team to speed up the testing process. The engineers took the opportunity to design the tests before testing began.

The Test Item Specification (TIS) used in the software QA team was used. This had three main sections what will be tested, a list of test cases and test procedures. Each section could be reviewed before proceeding to the next. This allowed corrections to the earlier sections of the TIS before time was wasted on the more detailed later sections. Also, the earlier sections were shorter making it easier to get the design engineers to spend time from their schedule to do a review. The TIS format is shown in Appendix 1.

The first section of the TIS described what was going to be tested. In the software QA team this section was reviewed by the design engineer who wrote the code that would be tested. Since the design engineer for the engine firmware was in a different company, the design engineer who was the customer for the engine firmware reviewed this section and the other QA engineer reviewed this section.

The second section of the TIS was a list of the test cases. The test cases referenced requirements in the interface specification. The engine firmware QA team was fortunate in having one of the best specifications. This was partly because we were dealing with an external company. It was also do to the fact that we wrote the specification. This specification meant designing the test cases could go more quickly. It also meant we could tie the cases back to requirements in the specification.

The third section of the TIS was the actual test procedures. The procedures were optimized to team tests with like configurations together. For example, detecting letter size paper, being able to feed the letter paper though the printer and printing on letter size paper in the correct location could all be tested with a single step. While, being able to detect the absence and

presence of the second feeder was done in two different steps because all the single feeder tests were teamed at the end of the procedure.

The first benefit was the testing could be done in a single week. This gave the vendor a week to make changes before sending a new version. As the code became more stable and had fewer failures, the test cycle was reduced to two days.

A second benefit was isolation was easier. Each procedure step was self contained and written down. This meant the failure could almost always be duplicated by repeating the procedure. A few failures were timing related and were hard to reproduce manually.

A third benefit came when we started the next project. We could re-use some of the work we had done on the last project. Even though the new printer was a different technology from a different company a large part of the areas to be tested and test cases could be used as a starting point for the new project.

The fourth benefit we hoped to get was freeing up more of the QA engineer's time by having test fixture operators run the tests. We exceeded our expectations. We designed the test procedures for a test fixture operator with a good attitude and minimal computer experience. However, we often found people who were much more capable. As the test operators became more experienced an became test technicians, the QA engineers were freed to work on additional tasks. While we had hoped that would give the QA engineers a chance to work on better tools, it often meant it gave the team a chance to do more overlapping projects.

As the test operators became familiar with the tests could soon check the output and log files. Anything that they weren't sure of or that needed investigation, they could flag for the engineer to look at. The next logical step was for them to submit the defects they found.

Up Front Planning

On the first project, the test planning had been done by the software QA team. This had one major problem. They did the test plan based on their schedule. When they put together the test plan, the engine firmware QA team had finished test design and was already testing.

Having the test plan done before test development began. Gave the chance to get the necessary resources lined up ahead of time. The biggest benefit was being able to show the need for a test operator. This would allow one of the two QA engineers to start on the next project earlier. The advantage

management saw was some of the work could be shifted to less skilled people. An example of a test plan is in Appendix 2.

The test plan described what areas would and wouldn't be tested. This listed what tests would or wouldn't be developed compared to the TIS which listed individual test cases that would or wouldn't be tested. (A test here is a test procedure or a set of test cases. A test case is check which can either pass or fail.) The test plan also included what equipment and personal was required for the level of testing described. This allowed management to compare the coverage against the cost.

An other important section was the risks section. This described what could go wrong and how it would be dealt with. For example, the risk that an other project schedule slide which would cause a conflict in resources would be handled by putting the resources on the highest priority project and delaying the other project.

The importance of up front planning is it allows holes in the process to be identified early enough to solve. A section of the test plan lists personnel and the training they will need. This identified the need for documentation for the new test fixture operators to learn the process. This was a more effective and complete method than having the lead test technician tell the new operators everything they can think of when they arrive. Our test technicians were a logical choice to do that documentation. Having completed the testing on a project, they had the best idea what the new operators needed to know. Also, the more tasks the QA engineers didn't have to do, the more they could concentrate on improving the process.

Tools Are Part of the Process

The TIS's and test plans had eliminated enough problems that the effort to port the tools from one project to the next became the biggest stumbling block that management saw. The engineers saw the lack of test logs and the scripts ability to handle unexpected events as a major problem.

The new test fixture is shown in figure 1. The fixture was easier to move to new printers because the logic on the minimal image processor (MinIP) was in an LCA instead of discrete logic on a wire wrapped board. Log files were added to test fixture. The assembler like format was retained in the scripting language. Branching, symbolic constants and macros were added.

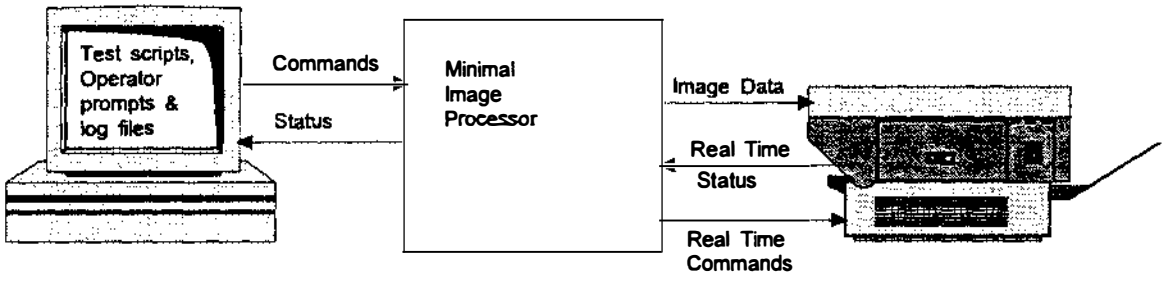


Figure 1 - Test Fixture

The ease of modifying will be a benefit on future projects. The additional features in the scripting language made it easier to handle complex conditions and errors. The tests were able to report failures to the log then attempt to return the system to an expected state. This made it easier for the test operators. For example, they didn't need to run an other script to eject paper from the printer.

When this tool was being developed, the engineers wanted to be more aggressive and make bigger changes to the tool. Management wasn't sold on the benefits were worth the risk of that radical a change. At this point in the team's evolution, it would have been effective to present data quantifying the benefits. There was now sufficient process and data to support a more systematic approach. But out of habit, the team took the what they could get. The incremental approach had severed them well up to now. They Knew that they could add the MOSS tool after the test fixture was proven. As it turns out the biggest reason for taking the more aggressive approach wouldn't have shown up in the data. The move aggressive approach would have made the tools used by the internal engine firmware QA team and external engine QA team more common. It wasn't know at the start of the project that the two teams would be combined by the end of the project.

The MOSS tool takes requirements in the form of a state table. Allows the specification of the initial state, stimulus, response and the terminal state. Being a real-time system, the specifications include timing information. For example, a complex response may list several statuses and the time interval for each. A complete description of the system can be found in "Mother2 and Moss: Automated Test Generation from Real-time Requirements" by Joe Maybee in the 1993 proceedings of the Pacific Northwest Software Quality Conference.

Multiple Projects

About the time the test plan was completed for two new projects, the team was moved from the engine group to the same group the internally developed engine firmware was tested in. The new leader expected more formal

processes instead of allowing it to happen. He wanted a work breakdown structure document (WBS). The new leader wanted to use the WBS because of it's effectiveness in planning and because it gave him a way to learn how his new team worked. The processes and tools developed by his two teams were significantly different.

The first section of the WBS is the test taxonomy. This section describes the different phase of the project and lists the tasks in those phases. The remainder of the WBS describes each task, how long it would take, the resources required, other tasks it depended on and how to tell when it was completed. An example of the WBS can be found in Appendix 3.

The WBS proved its self most valuable when trying to make tradeoff between two simultaneous projects. The most important tasks on the highest priority project could be identified and people applied to them. The important tasks on the other project could be worked on when resources freed up.

The people assigned to the tasks in the WBS estimated the time required. Because the project schedules shifted in relationship to each other, the people originally assigned to the tasks weren't always available. When other people were assigned to the tasks, the times had to be adjusted. For example when an inexperienced engineer from an other group was assigned to a task that was originally estimated for an experienced QA engineer, the task requires more time. When one project was put on hold, the resources could be shifted back to other projects. They could then pick up the WBS for the other project and have a record of what tasks still needed to be done.

Better But Not Complete

A lot of progress has been made. Tests, scheduling and processes are now documented. This allows tasks to be done by people with the best skill set for that task. Test development and testing is taking less time. However, improvement in the process is still occurring.

With test procedures documented, it required less of the QA engineer's time to develop tests for new project because they could leverage off of past projects. However, QA engineers are in short supply and projects aren't. Could some of the tests be modified by the test technicians? Yes. On the manual tests, the engineer could provide them with a list of things that needed to change for the new project and review the tests when they were complete for simple changes. On tests which required more changes, additional direction was required during the work on the tests. However, tests were now being developed by an engineer and a technician that had required two or more engineers in the past.

This freeing of the QA engineers by the Test Technicians, gave the QA engineers to start using a more data driven approach. For example, the metric which tests are finding the most bugs produced useful data. It pointed out the functional tests find the most errors in early testing cycles while the user based tests find more errors when the product becomes more stable. This allowed the amount of functional testing to be reduced freeing test fixture operators to work for an other team within the QA department.

Appendix 1 - TIS Format

This is an example of a TIS. Each section provides a short sample from the TIS.

XXX
Engine Firmware
Test Item Specification

Printer Name

Test Suite (Test Set): Devices (Printer)

Author: XXX

Reviewers: XXX, XXX

Date of last revision: 28 August, 1996

For use during testing:

Tester's Name _____

Project ID _____

Engine ID _____

Code Version (PS/Eng) _____ / _____

Date: Test Run / Results Entered _____/_____

Time to: Setup / Run / Evaluate _____/_____/_____

Checkpoints: Total / Run / Failed _____/_____/_____

TEST ITEM OVERVIEW

This introductory section makes it easier to identify which TIS covers which features.

TEST CONDITIONS

The engine has a serial command and status protocol. Tests of the protocol include error conditions.

The engine commands are tested **except** for the executive size, black legal request set, black legal request reset, dummy print and dummy print reset. The excluded commands aren't used by the image processor.

Error status and the status commands are tested.

NOT TESTED

Most of the aaa and bbb work is now being done in the image processor instead of the engine. The images that must be created are being defined by the design team. These features will be tested as it is being designed.

TEST CASES

Numbers of the form c# are case numbers. Numbers of the form p# are procedure step numbers. Numbers of the form s# are references to sections in the EIS containing the specified requirement being tested. [The EIS is an engineering specification that defines the product electrically and mechanically. It also defines the interface between software designed by Tektronix and software designed by the vendor.]

c1.(p1, s6.4.1, s6.4.3.3.1) To check command protocol, the firmware version is requested.

c2.(p1, s6.4.1, s6.4.3.3.1) To check command protocol, the firmware version is requested with a parity error on the first byte.

Reference	Command	In Tray
c3.(p2, s6.4.2, s6.4.4)	Letter Size	letter paper
c4.(p2, s6.4.2, s6.4.4)	Letter Size	A4 paper
c5.(p2, s6.4.2, s6.4.4)	A4 Size	letter paper
c6.(p2, s6.4.2, s6.4.4)	A4 Size	A4 paper

TEST RESOURCE REQUIREMENTS

The following equipment is required:

- 1 PC (AT or better) with a serial port
- 1 XXX printer with optional feeder and a MinIP
- 1 Each type of media tray
- 1 each color Empty toner cartridge
- 1 each color Mostly empty toner cartridge

The person running this test needs to be familiar with DOS PC applications, and cabling between PC and printer. A second person is needed to move the print on and off the optional feeder unit. A QA engineer needs to run the tests the first time because some

tests will require the runner to determine what is the correct response, and provide input to the EIS.

TEST PROCEDURES & CHECKPOINTS

Place a √ in each checkpoint the passes. Place an X in each checkpoint that fails and indicate near the checkpoint the condition that failed. (This can be a status message, description of the print or other indication.) Generally, passing checkpoints will be indicated by a White background box saying "Match" while failed checkpoints will be indicated by a Red background box saying "mismatch". If a mismatch occurs, note the 1st 2 digits and the last 2 digits in the green background box that follows the Red background mismatch box.

Note: Each procedure step is independent of other procedure steps. However, each procedure step should be treated as a unit.

p1.(c1) Run the script **version** in *min_ip*.

_____ No errors are reported.

_____ The version number matches the version on the ROM.

Appendix 2 - Test Plan Format

This is an example of a test plan. Each section provides a short sample from the TIS.

XXX (Phaser XXX):
Engine Firmware Testplan

Rick Clements

Test plan for release of
the XXX Project.
Based on IEEE/ANSI
Std 829-1983

This is a Configuration Management Item

1. Test Plan Identifier

Project ZZZ

2. Introduction

This test plan covers the ZZZ project for the Phaser XXX. It doesn't include PostScript controller or network cards; they are covered in the *XXX PostScript Test Plan*.

2.1 Objectives

This software test plan is intended to support the following objectives:

- Detail the activities required to prepare for and conduct the software acceptance test.
- Describe the areas of functionality to be tested.
- Communicate the responsibilities, tasks and schedules to the concerned parties.
- Define sources of information used to prepare this plan.
- Define the test tools and test environment needed to conduct the software test.
- To define the human resources needed to conduct the software test.

2.2. Background

This section describes the major features and changes for the project. It also lists the priorities from the test management document. This allows the QA engineers to make tradeoffs based on what's important on this project.

3. Test Items

The items to be tested are:

- List of TIS's

4. Features To Be Tested

This section provides a brief overview of what printer features will be tested. See section 6 for more information.

5. Features Not To Be Tested

This section provides a brief overview of what printer features will not be tested. See section 6 for more details.

- The print engine itself (mechanical reliability, etc.) will not be specifically tested. The OEM engines team and ORT test the engine.

6 Approach

Each testing cycle will contain the following steps.

1. When the firmware is received, one copy of the firmware will be burned. The *Smoke Test TIS* will be run to verify we have a good build. Time: 1 hr target, 2 hr maximum
2. Additional copies of the firmware will be burned. The *Low Level Engine Firmware TIS*, *High Level Engine Firmware TIS*, *Engine Diagnostics TIS*, and *Speed and Pipelining TIS* will be run. Time: 2 days ramping down to 1 day.
3. The firmware is made available to be distributed to the design team and solutions QA.

6.1. Tests

- The Smoke Test TIS verifies the firmware is ready for general testing.
- The *Low Level Engine Firmware TIS* covers the signals and timing. These tests require the MinIP, a logic analyzer and an oscilloscope.
- The *High Level Engine Firmware TIS* the interface commands and status. These tests require the MinIP.
- The *Speed and Pipelining TIS* test the increased speed and error handling in a pipelined environment. These tests require the MinIP.
- The *Engine Diagnostics TIS* test the engine's ability to detect and indicate service level information on various engine problems. These tests require the MinIP.

6.2. Tools

The following test tools are used by OEM Engine Firmware QA:

- The MinIP is used send engine commands, read engine status, and provide stimulus for timing and signals.
- Logic Analyzer and an oscilloscope are used to verify timing and signals.

7 Test Pass/Fail Criteria

Each test in the TIS passes if, and only if there are no classification 1, 2 or 3 defects discovered. (See table 1).

Classification	Description
0	Dangerous - Cause injury to a person or damage to equipment! (Example: not stopping moving parts when a cover is opened.)
1	Critical - Catastrophic and unrecoverable! (Example: system crash or lost user data.)
2	Severe - Severely broken and no work around. (Example: can't use major product function.)
3	Moderate - A defect that needs to be fixed but there is a work around. (Example: user data must be modified to work.)
4	Minor - A defect that causes small impact. (Example: error messages aren't very clear.)
5	Enhancements, suggestions and informational notes.

Table 1 - Failure Classifications

8 Suspension Criteria and Resumption Requirements

The testing of a test item will be suspended if:

- there is no valid specification for the item
- the item fails in such a way that further testing will provide little or no new information.

Testing will resume when the reason for suspension no longer applies. The QA Smoke Test is designed to find such catastrophic errors before each testing cycle begins.

9 Test Deliverables

OEM Engine Firmware QA will generate the following:

- Report per release that indicates the number of test checkpoints planned and run for each test item, the number of bugs submitted, resolved, and postponed, and the 5 worst bugs, according to the gut feel of the QA lead. (Submitted bugs will be categorized by priority.)
- Defect reports will be submitted into the DDSs tracking system under the project names XXX.engfw and XXX.mech. In each report, QA will attempt to describe the user-visible symptom, the technical problem, and the estimated customer impact. High priority defects will be summarized by the OEM vender communications liaison then Faxed to the supplier to be fixed.

10 Testing Tasks

This section lists the tasks required.

11 Equipment Needs

This section shows the equipment needs.

12 Assumptions, Risks, and Contingencies

- Documentation - QA's test preparation depends heavily on all the Section 6 of the XXX EIS. If it is finished late, the quality of our testing will decrease.
- No Impact From Other Projects Rolling releases on YYY will cause testing on XXX to be late or incomplete. This is further impacted because XXX test development is now competing for resources with YYY.

Appendix 3 - WBS format

This is an example of a WBS. Each section provides a short sample from the TIS.

XXX: Engine Firmware QA Work Breakdown Structure

Rick Clements, Lead XXX Engine Firmware QA Engineer

A concise description of Print Engine Firmware QA tasks for the XXX Project.

1.0 Introduction

This document is intended to specify Print Engine Firmware QA tasks for the XXX project in sufficient detail to allow planning with a reasonable degree of confidence.

This document outlines:

- Task Taxonomy - A listing of task groups in outline form.
- Detailed Task Specifications - A detailed description of the tasks, the resources required for the task, entry and exit criteria and estimated time to accomplish the task.

This document is not placed under configuration management, it is under the control of the Print Engines QA group.

2.0 Task Taxonomy

The outline below does not imply an order to the tasks.

1000 XXX QA project planning

- 1100 Engine Firmware QA Inception Documents
  - 1101 Engine Firmware QA Plan
  - 1110 Engine Firmware QA Work Breakdown Structure
    - 1111 Preliminary Work Breakdown Structure
    - 1112 Review Work Breakdown Structure
    - 1113 Final Work Breakdown Structure

2000 Definition Phase

- 2100 Engine Firmware Requirements Specification
  - 2101 Review existing requirements
  - 2102 Analyze for missing requirements
  - 2110 Modify requirements database
    - 2111 Fixing existing requirements
    - 2112 Create new requirements for next generation

3.0 Detailed Task Specifications

Task:	**1101 Engine Firmware QA Plan**
Task Definition:	Describe the documents that go into the XXX Engine Firmware QA schedule.
Deliverables:	"XXX Engine Firmware QA Plan".
Exit Criteria:	Final version of document.
Resources Req'd:	Purpose and names of documents.
Assumptions:	None.
Estimated Effort:	1 week.
Constraints:	None.
Assigned To:	XXX

Task:	**2101 Review Existing Requirements**
Task Definition:	Verify that the existing requirements are correct for XXX.
Deliverables:	List of requirements that need correction.
Exit Criteria:	List of all requirements that need to be corrected.
Resources Req'd:	Requirements.
Assumptions:	None.
Estimated Effort:	1 week.
Constraints:	None.
Assigned To:	XXX

SYNOPSYS®

Documenting Software Development Process Information on the World Wide Web

Abstract

The world wide web (WWW) provides a flexible and easily accessible method for viewing software development process information. The web allows information to be layered, provides a way to hyperlink to related documentation, and allows readers to chart their own paths through the information. This paper describes the advantages and the issues associated with putting process and quality information on the WWW.

Leslie A. Dent
Synopsys, Inc.
700 East Middlefield Road
Mountain View, CA 94043
http://www.synopsys.com
(415) 694-4468
email: ldent@synopsys.com

Keywords: Software Development Process, Best Practices, WWW page design

About the Author: Leslie A. Dent is a senior software quality engineer at Synopsys. After graduating from UC Berkeley with a degree in Computer Science, she found her niche in studying software quality issues and has spent most of her career in software quality assurance, process improvement and testing.

Introduction

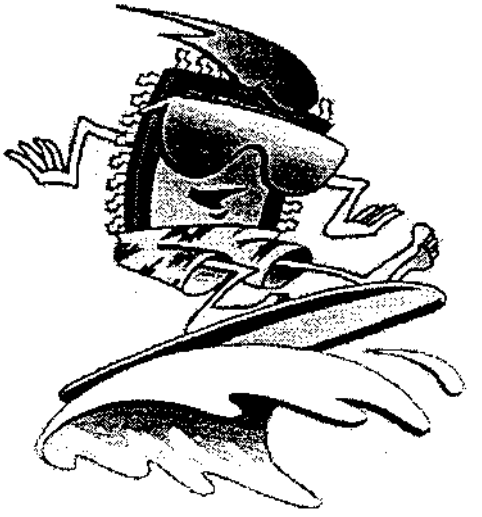
Synopsys, like many other companies, has moved from viewing the world wide web (WWW) as just an advertising medium, to an essential corporate tool. We are developing an extensive intranet[a] as a way to communicate information within the company. My department, Metrics and Processes, is on the forefront of this movement. We have a web tree[b] for each of our software process improvement efforts. This paper illustrates how to develop and publish software development process information on the WWW.

Why Choose the World Wide Web?

We have chosen web pages[c] as the mechanism to document and publish our software process improvement projects for a number of reasons. At Synopsys we have a platform incompatibility problem caused by engineers using UNIX workstations and managers using Macintosh machines. PCs are also becoming prevalent as we start developing and selling products on the PC. Documentation developed in an application on one platform is difficult to access from the others. Any platform can run a web browser[d] and provide users with easy access to the documentation from their desk top computer or terminal.

Another reason to use web pages is that the most recent version of the documentation is always readily and instantly available. This is especially important since our process improvement efforts are evolutionary. After we develop a web page for the high level concepts, we can start on the web pages for the next level of detail and continuously add levels or layers of information. We can publicize the addition of each layer and users can browse the new information when they are ready. With more traditional methods of publishing, our group would need to wait until the document was finished and send out copies to our process users. Not only does using web pages saves us money in terms of copy and mail expenses but presenting the information in coherent sections can be less intimidating than a bound notebook. Instead of filing the document on the shelf more people are likely to read parts of the documentation and to look things up in it. The intranet also allows us to hyperlink[e] to related documentation. Instead of having to search for a reference, a user can activate a hyperlink and the web browser will take them to the reference.

Documentation on the web can be used for both education and reference. Readers can chart their own path through the documentation taking either a "breadth-first" by reading the top level page for each subject, a "depth-first" approach by reading all of the information for a particular subject, or going to a particular section. There can be significant productivity gains for both the reader and the web page publisher. Readers can browse material at their own pace when it fits their schedule or when they need to lookup a piece of information. Publishers don't have to answer the same questions from different people but can provide the answers which can be referred to again and again. A judicious addition of graphics jazzes up the documentation and makes it fun to read.



a. intranet: a network that can only be accessed from within a company, as opposed to the internet which is public.

b. web tree: a hierarchial group of web pages.

c. web page: a document that can be displayed by a web browser.

d. web browser: an application which allows you to view graphics, text, and HTML (hypertext markup language) files and navigate hyperlinks.

e. hyperlink: a pointer to another web page or a section of a web page; the hyperlink gives address information to the web browser.

What Needs to be Done

The remainder of this paper lists and describes the steps necessary to develop and publish documentation on the web. Two example projects are used to illustrate how we executed these steps. The steps are:

- Getting Started
- Designing a Web Structure
- Establishing Conventions
- Selecting an Authoring Tool
- Developing the Web Pages
- Conducting Usability Testing
- Marketing the Web Pages

Example Project - Software Testing Guidelines: This documentation provides guidelines and standardized procedures for planning and conducting testing. It describes six different types of testing and the procedures associated with each.

Example Project - Software Development Process Framework: The framework defines a process model that is a cross between a waterfall and an iterative model[1]. There are phases which represent a time-based view and activities which represent a function-based view. Activities are general names for the types of tasks completed during the process. The activities occur in all phases to a certain degree. Each task may also have best practices which are specific methods to perform a task. The framework defends a set of phases and activities which can be tailored for a specific development effort.

Getting Started

"Users of multimedia computer documents don't just look at information, they interact with it in novel ways that have no precedence in paper document design."[2] Developers of web pages need to consider all of the issues that are involved in publishing a hardcopy document and more. They need to understand graphics and user interface issues so they can successfully guide the user through the documentation.

The first step is to have a clear idea of what you want to present. Are you documenting a known process? Are you trying to change the way people do things? Who is your audience? Knowing the answers to these questions, will enable you to structure your documentation appropriately for the WWW.

Example - Software Testing Guidelines: The purpose of the testing guidelines is to educate the development groups. Testing at Synopsys is ad hoc. The guidelines are intended to change behavior by providing the information which will enable people to make choices about what kind and how much testing to do for a particular project. The audience for the guidelines is all members of the development groups including:

- Research &Development engineers
- Application Engineers
- Technical Writers
- Marketing Managers

Example - Software Development Process Framework: Each development project is expected to have a process, to document it, and to follow it. The framework changes the way people conduct projects by formalizing the process. The Software Development Process Framework serves as a reference document and facilitates the customization of a software development process for an individual project.

Designing a Web Structure

Structuring web documentation involves several decisions. First, decide how to break the documentation into web pages. Will the documentation be contained in a single web page with hyperlinks into specific sections? Or will there be a separate web page for each topic?

Studies have shown that users get frustrated by too much scrolling.[2] Web pages do not have page numbers, so users they can feel lost when scrolling through a long web page. Users also find it frustrating to follow several hyperlinks just to get to a web page that has only one or two paragraphs of information. Being too deep in a document can also make users feel lost. Having too much information on one web page or having too many web pages can frustrate the user. A balance between these extremes should be the goal. The approach should be based on the content of the document. Each page should be able to stand alone as well as fit into the document flow. Our guideline is that the user should not have to scroll more than five times within a single web page.

Next, decide how to put the web page files into a physical directory structure. Aligning the physical file directory structure with the web structure makes the web tree easier to maintain. For each layer of information, there should be a separate directory. The disadvantages of a flat structure versus a deep structure are similar to the web page structure itself. A flat tree puts too many files in one directory. A deep tree creates too many directories.

Example - Software Testing Guidelines: The top level of the guidelines has web pages which discuss the background and motivation of the project, give an explanation of the sections found in each testing definition web page, define testing terminology in a glossary and provide references. Each type of testing (and sub-types if applicable) has its own web page within its own directory. There are also web pages for any tools that can be used for the testing. See figure 1 for a graphical view of the web structure of the testing guidelines. (Note: index.html is the standard designation for the root or main page of the web tree.)

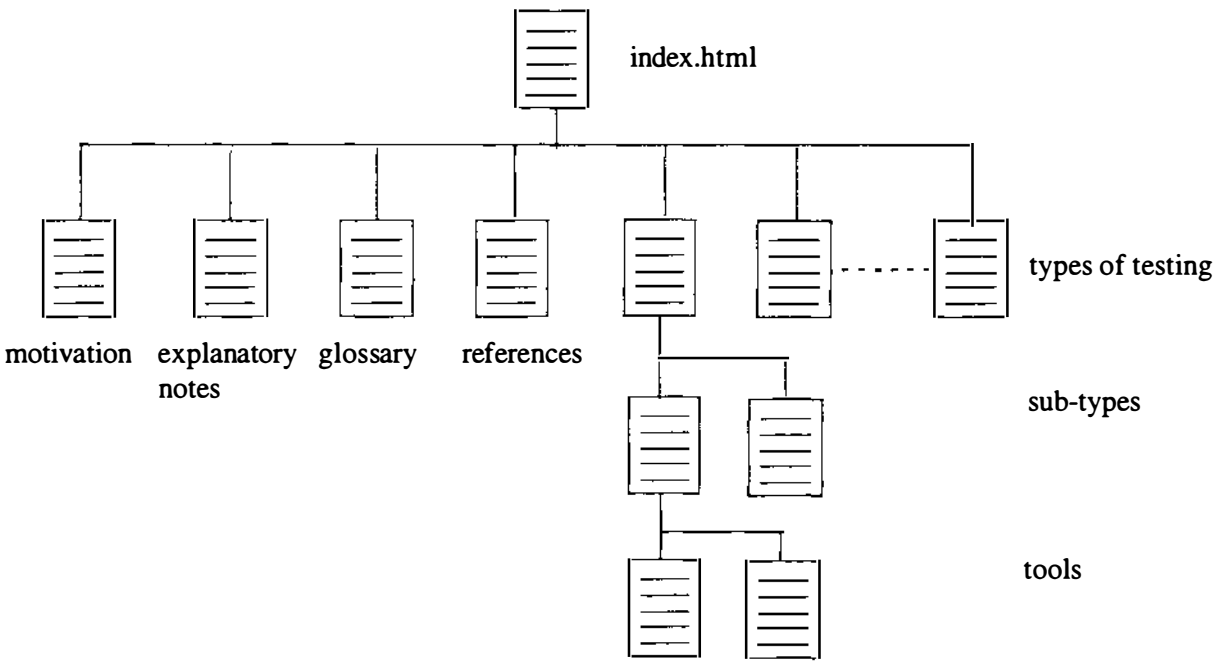


Figure 1: Software Testing Guidelines Web Structure

Example - Software Development Process Framework: The framework can be traversed from either the activities or the phases. Both paths lead to the activities within phases and the tasks. The general phase descriptions are kept in one directory. Each activity has its own directory with a sub-directory a separate directory for each phase. There is a best practices directory within each phase under the activities with a web page for each best practice. See figure 2 for a graphical view of the web structure of the framework.

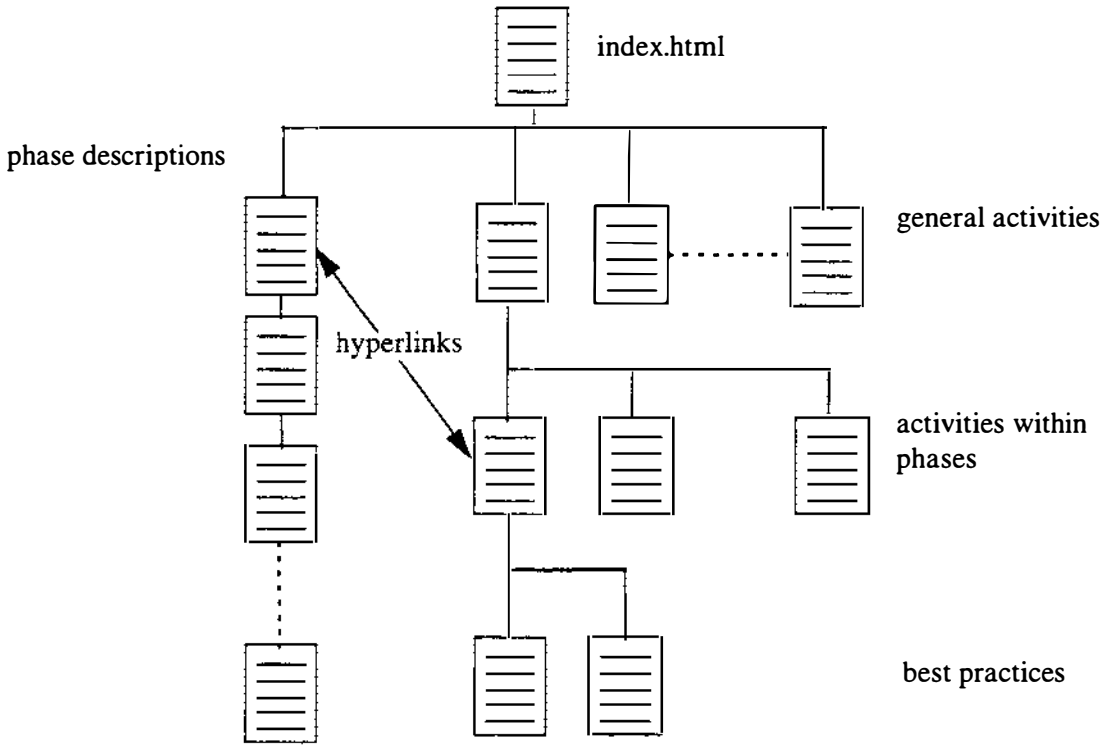


Figure 2: Software Development Process Framework Web Structure

Establishing Conventions

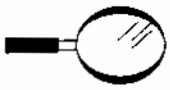
- Page Layout
  - Header
  - Footer
- Content Layout

"When it comes to building web pages, your job as a designer is to aid the eye in recognizing salient features or elements, and in locating the items of interest quickly and efficiently."[3] Standard layouts and conventions ensure the web pages are consistent and make it easier for the reader to discern the main points of the documentation and to navigate the web pages. The conventions can be put into templates to make web page to guarantee consistency.

Page Layout

Like a published document, each web page should have a header and a footer. The header should contain sufficient information to allow a reader to hyperlink to it from somewhere else without becoming disoriented or confused by being in a different location. Figure 3 illustrates the header for the Software Testing Guidelines web pages. The header should contain:

- Any standard company banner information
- The title of the document or project
- The title of the section
- A section icon (optional)
- A table of contents (if the web page does not fit on the current screen)



Software Testing Guidelines

Static Analysis Testing - Code Analysis

Overview	1.0 Introduction 1.1 Rationale 1.2 Timeframe
Activities	1.3 Planning 1.4 Developing 1.5 Executing
Inputs and Outputs	1.6 Prerequisites 1.7 Deliverables 1.8 Metrics
Additional Information	1.9 Tools and Techniques 1.10 Examples
Additional Information	1.11 Limitations 1.12 References 1.13 Recommendations for Future Work

Figure 3: Software Testing Guidelines Sample Header

The footer should contain publishing information about page and provide the reader a way to navigate to other pages in the tree by hyperlinks. Figure 4 illustrates the footer of the Software Testing Guidelines. The footer should contain:

- The title of the section
- Navigational buttons[a] to:
  - The top of the document or web tree
  - The previous section of the document
  - The next section of the document
- The author's name and contact information, including an email address
- The date the documentation was last modified
- Copyright or company information

a. navigational button: a graphical element which contains a hyperlink.

The navigational buttons for the previous and next sections denote a recommended reading order. Occasionally, this order may seem arbitrary but it should be consistent and every page should be included. (The recommended reading order is the order in which the pages would appear in a hardcopy document.)

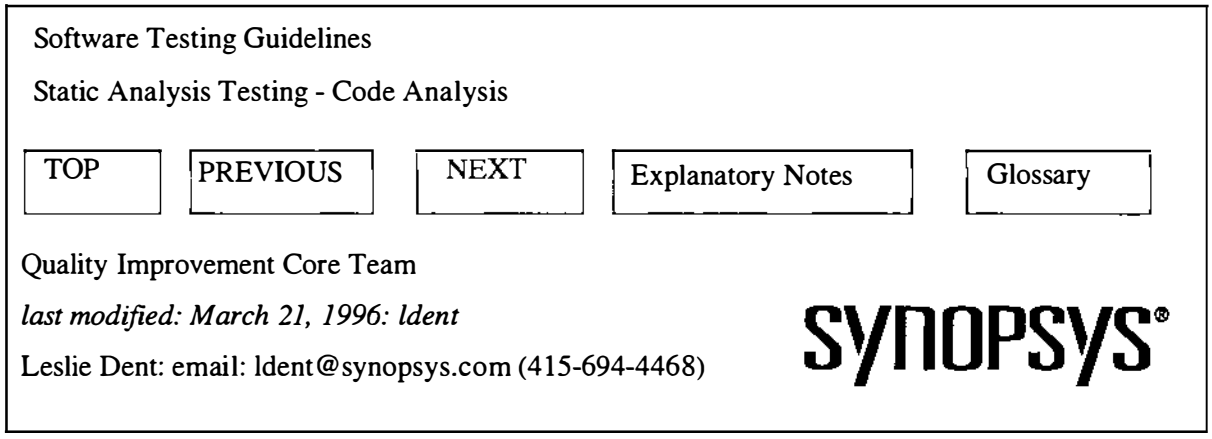


Figure 4: Software Testing Guidelines Sample Footer

Content Layout

Each web page should have a consistent "look and feel" in its content as well as its page layout. Suggestions for achieving a consistent look and feel are given below. The suggestions were compiled after informal usability testing of web trees and researching web style issues discussed in several style guides.[2,4,5]

- *Put a short hyperlinked table of contents at the top of any web page that extends beyond the current screen.* This lets readers know what to expect on the web page and allows them to activate a hyperlink to jump to a particular section if they desire.
- *Standardize section headings and use them liberally.* Let the reader know what is discussed in each section. This allows the reader to skim a document and determine which parts to read. Well designed headings can grab the reader's attention and contribute to a positive perception of the document.
- *Use horizontal rules to help mark sections.* A simple horizontal rule breaks up a page into distinct sections. Use the rules to set off the header and footer sections.
- *A single paragraph should contain fewer than 8 lines.* Too much text is intimidating. Information divided into smaller pieces and the inclusion of white space makes it easier for the reader to absorb. Bullets can also be used to break up the text when items are listed.
- *Use meaningful hyperlink names.* Give enough context to links so that the user doesn't need to go to the hyperlink to figure out what information is there. The user may choose not to go to the hyperlink, so the text should be understandable whether or not the hyperlink is traversed.
- *Test all hyperlinks that are put in the page.* There is nothing more annoying than invoking a hyperlink that is broken. Broken links are defects in a web page. They give the user an impression of poor quality. There are tools which can automatically traverse a web tree and report any broken hyperlinks. If the document referenced by the hyperlink doesn't exist yet, make a page for it which includes at least a header and states that it is under development.
- *Include at least one graphic per page.* Text-only pages are boring. Use a few appropriate graphics to illustrate concepts or act as a visual cue for the page. Too many graphics can cause the web page to take too long to load. A web page should load in less than 8 seconds.

Selecting an Authoring Tool

Many people find it difficult to compose documents in "straight" HTML[a]. They often get distracted by inserting the syntax and neglect the content. There are many alternatives to composing in HTML. Compose the documents in ASCII and after they are finished insert the appropriate HTML commands. WYSIWYG (What You See Is What You Get) HTML processors are available and shield the writer from the HTML syntax. Documents can also be written using a standard word processors and then converted to HTML. Many popular word processing programs have built-in HTML converters. Freeware conversion programs are also available.

Graphics work best developed in another application and then converted to GIF[b] (or JPEG[c]) files. Many freeware programs exist to convert postscript files to GIF files. Built-in word processing converters and freeware conversion tools also convert graphics to GIF files.

When selecting a tool for authoring, consider the medium in which the web pages will be maintained. Will the document be written in one tool, converted to HTML and maintained in HTML? Or well it be maintained in the original source and be converted each time the source is changed?

Another issue is version control. Html files are ASCII files and thus can be managed under a standard version control system such as RCS[d]. Most configuration management tools also have add-on features to control word processing documents.

Example - Software Testing Guidelines: The software testing guidelines were written as chapters in Adobe's FrameMaker. We used a freeware version of webmaker to convert them to HTML. We weren't completely satisfied with the resulting HTML because the sections were too disjoint. After editing one chapter (web page) to our liking, we wrote editing scripts to modify the other HTML files. We also added graphics from a FrameMaker presentation by converting the graphics to GIF files using webmaker. The web pages will be maintained in HTML and the revisions will be controlled by Atria's ClearCase tool.

Example - Software Develop Process Framework: Some web pages were composed in straight HTML, while others were written in ASCII and then converted to HTML. Templates were designed for each of the layers of pages. The pages are stored in RCS.

Developing the Web Pages

Now you are ready to develop your pages. Be sure to take advantage of any writing style guides that have been adopted by your Technical Publications department. Also utilize their editing services. Since the web pages will be visible by your whole company, they should be polished.

Be careful to check how different web browsers or versions of the web browsers that are in use at your company display the web pages. The web page may look fine in one browser but be unreadable in another due to a syntax error or non-standard constructs. Tools such as weblint help perform syntax checking and give warnings for non-standard constructs such as the center command.

Example - Software Testing Guidelines: Each chapter was put through a technical review. The whole FrameMaker document was reviewed by an editor from the Technical Publications group. Weblint was run on every page. See table 1 for an example of the content of one the testing guidelines web pages. Each testing web page uses section headers in the left margin.

---

a. HTML (HyperText Markup Language): a language which provides formatting commands that are interpreted by the web browser.
b. GIF (Graphical Interchange Format): a commonly used compact file format for graphics.
c. JPEG (Joint Photographic Experts Group): a commonly used file format for storing photographs.
d. RCS (Revision Control System): a standard UNIX utility for controlling revisions of ASCII files.

Overview

1.0 Introduction

Definition	Code analysis is the use of external tools to automatically process source code and return an list of real and potential errors.
Goal	Catch code errors before the code is executed.
Audience	Code analysis only applies to code and therefore will be most relevant to R&D
Basic Considerations	The tools detect certain types of errors or adherence to standards. Examples of tools are lint, call graph generators, code formatters and a variety of complexity analysis/measurement tools.

1.1 Rationale

Benefits	Lint locates real errors and potential problems and points the user to the ...
Consequences	Code can often be compiled and run, even if it has errors in it which lint can ...
Cost	Code analysis is very cheap to perform and is easily automated. Usually it ....

1.2 Timeframe

- Static code analysis should be done periodically during code development.
- ...

Activities

1.3 Planning

- Determine which code analysis tools will be used.

....

1.4 Developing

Ensure all Makefiles have necessary targets for the code analysis tools.

1.5 Executing

For each code file or module:

...

Table 1: Software Testing Guidelines Web Page Sample Contents

Example - Software Development Process Framework: Each page was reviewed by one or more team members. Weblint was run on every page.

Conducting Usability Testing

Web pages are essentially a user interface, so ease of use or usability tests should be conducted on them. The testing can be as informal as asking users for their feedback or as formal as watching the users navigate the documentation while trying to find specific information.

Example - Software Testing Guidelines: Experienced web users were asked for their feedback on a sample web page. Their comments resulted in dividing the sub-sections into four major parts and displaying the hyperlinked table of contents using the HTML table command.

Example - Software Development Process Framework: This was an early effort. After we put together a few pages we did an informal usability test. We watched while people navigated the pages and asked them to give us their feedback. The list of conventions discussed earlier was derived in part from this effort. The items that stood out are listed below.

- Users would click any and every hyperlink.
- They got lost easily if there was no context information on the page.
- They didn't like to scroll more than a few times.
- They loved icons and graphics.

Marketing the Web Pages

Now that you have a full set of web pages for your process improvement effort, publicize the web pages in your organization and make everyone want to read them. At Synopsys, we have an intranet web page that we can use to advertise new pages. We can also submit our pages so that they get listed in the appropriate table of contents allowing someone looking for such information to find our web pages. For the official intranet that is under development, we are planning to have contests related to the web pages including a scavenger hunt with prizes as a way to encourage people to use the web pages.

Example - Software Testing Guidelines: A presentation focused on the content was made to the Research & Development directors. The web pages were hyperlinked into related software process web pages.

Example - Software Development Process Framework: A presentation focused on the content was made to the group managers. The web pages were hyperlinked into the groups department web page. Additional input was solicited from the group.

Future Work

As our knowledge about the technology available when using the WWW increases, we will be exploring some additions to our pages. The WWW supports multimedia. We have talked about recording well-respected engineers discussing their best practices for software development and including the audio in the pages.

Some groups have used the forms[a] capability to do on-line reviews of documents that are displayed in HTML. Reviewers can fill out their comments for each section, push a button to activate the form and send those comments to the document's author. Other groups use the forms capability to solicit feedback about a web page or project proposal.

We will also be adding a search capability for every web tree as soon as our intranet department purchases a search engine. Sophisticated search capabilities act like indexes which require you to decide what words should be entered in the search table. Simple word search capabilities which conduct word searches without regard to context are also available.

a. forms: a method of sending data supplied by the user to a script which processes it.

Conclusions

Software development process information delivered on web pages provides many ways for the users to access the information. The user can chart their own path through the information or be guided by the publisher. They can read it all at once or use it to look up specific information. High level information can be made available immediately and the information can be layered to present a snapshot of the right amount of detail without being overwhelming. References can be easily integrated by hyperlinks.

References

1. Andrew Topper, Daniel Ouellette, Paul Jorgensen; *Structured Methods,* McGraw Hill, 1993
2. Lynch, Patrick. *Yale C/AIM WWW Style Manual.* http://info.med.yale.edu/caim/StyleManual_Top.html
3. Tittel, Ed, and James, Steve. *HTML for Dummies.* IDG Books, 1995, p. 60
4. Levine, Rick. *Guide to Web Style.* http://www.sun.com/styleguide
5. Allison, Bob. *The Web Master's Page,* http://gagme.wwa.com/~boba/masters1.html

Cleanroom and Organizational Change

Shirley A. Becker
Q-Labs, Inc.
College Park, Maryland
sbe@q-labs.com
(301) 864-0840

Michael Deck
Cleanroom Software Engineering, Inc.
Boulder, Colorado
deckm@cleansoft.com
(303) 494-3152

Tove Janzon
Q-Labs, Inc.
College Park, Maryland
tja@q-labs.com

Keywords: Cleanroom, Quality Improvement, Process Improvement, Quality Management, Risk Management, Incremental Development

Abstract

Cleanroom software engineering is a managerial and technical approach to software development that emphasizes quality through design, measured through testing. This paper addresses some of the challenges inherent in organization-wide adoption of Cleanroom software engineering. A short section surveys pilot-project techniques that have been used to introduce Cleanroom at the project level. The rest of the paper describes an organizational infrastructure that has been successfully used in a large-scale (400 programmers) application to support aspects of Cleanroom software engineering.

1. Introduction

For many years, the Cleanroom approach was a well-kept secret. It was used primarily on smaller projects – with impressive results, to be sure, but always with the concern about how to scale up to the next level. As more and more projects begin to use the Cleanroom practices, the need to address scale-up has become a serious concern. This paper will describe an organizational infrastructure and other management techniques that have been applied successfully in a large-scale application to initiate the use of Cleanroom.

The keynote phrase for the large-scale adoption of Cleanroom hinges on the idea of an "expectation of quality." Cleanroom represents a fundamental shift away from the notion that errors are likely and frequent. With this cultural shift come important organizational and managerial changes that will be described more fully in the following sections.

1.1 Overview

This paper is intended for the reader who has some familiarity with Cleanroom as applied on a project level. We begin with a brief introduction that describes the goals and practices of Cleanroom. Then, we look at practices for introducing and using Cleanroom in small-scale pilot projects. The bulk of this paper presents organizational-change practices that are used in the next stage: when large-scale Cleanroom adoption is underway in an organization. We conclude with some observations about future directions and research.

The general topic of software process improvement is covered much more thoroughly elsewhere. The most notable example is the Capability Maturity Model for software (Paulk et al., 1995). This paper will focus more closely on some of the specific organizational structures needed to support the Cleanroom approach. However, the other aspects of capability maturity, including estimation and metrics, are also important to the success of any software process improvement effort. The link between Cleanroom and the CMM is described by Arnold (1995).

1.2 Cleanroom Principles

Cleanroom software engineering (Mills et al., 1987; Cobb & Mills, 1990; Dyer, 1992) is an approach to software development that combines an engineering process with statistical quality control. It takes its name from the clean rooms used in chip manufacture. There, a scrupulously dust-free environment is maintained because it is much more expensive to remove defects from the chips after fabrication than to prevent their introduction during the process.

Cleanroom has a track record of quality and productivity improvement in a wide variety of settings (see, e.g., Linger, 1994). Cleanroom is based on two principles. A *design principle* is motivated by the mathematics of functional verification. In Cleanroom, design teams strive to produce systems that are error-free upon entry to testing. This is accomplished by replacing ad-hoc developer testing with team reviews. Reviewers use the idea that a program defines a mathematical function to argue the correctness of the design. At the same time reviewers evaluate the design qualities of the software such as maintainability and reusability.

The *testing principle* directs Cleanroom teams to focus their efforts on certifying the reliability of the software product, rather than trying to remove as many bugs as possible from the code. The

testing techniques are guided by statistical (also called stochastic) testing based on expected system usage. The Cleanroom testing approach also provides a framework for statistical process control (Becker et al., 1996).

1.3 Cleanroom Practices

Mills defined a set of core practices to be used by teams that implement the Cleanroom principles. These techniques are:

- Formal specification and design of intended behavior.
- Incremental development process model.
- Stepwise refinement of specifications to code.
- Correctness verification of developed code.
- Statistical certification of compiled software products.

Some of these techniques may be found in other formal methods of software engineering. For example, incremental development can and is performed by many who do not use the Cleanroom method. It is important to note, however, that each of the techniques was defined by Mills and his colleagues to play an integral role in the successful development of high-quality software systems.

Although Mills originally proposed that all of the preceding practices be applied in every Cleanroom project, recent experience has suggested that most successful Cleanroom projects pick and choose from practices that are appropriate to their particular needs. Deck (1994) surveyed several such projects. Hausler, et al., (1994) and Deck (1996) have both suggested that a phased and tailored approach to technique selection is necessary. We will discuss technique selection in more detail below.

2. Cleanroom Process Piloting

For most organizations, the path to wide-scale adoption of Cleanroom begins with a pilot project. Ideally, the pilot project is chosen because it is particularly representative of projects in the organization. More often, pilot projects begin because they contain "early adopters" in the managerial or technical ranks, or because the members of that project are frustrated with the status quo and want to achieve higher productivity and quality goals.

2.1 Goals of the Pilot

The goals of the pilot will include one or more of the following:

Evaluation of the Cleanroom process within the specific organization's environment and domain. For example, how does Cleanroom apply to the particular kinds of products that the organization develops? Or, how must Cleanroom be tailored to use a specific skill set?

Selection of Cleanroom practices. Most projects will choose from among the Cleanroom practices depending on their specific needs. In a pilot project, this selection must take into account the eventual needs of the larger organization.

Measurement of the pilot experience. What benefits were achieved through the pilot adoption of Cleanroom? What were the costs of implementation? What is the return on investment?

An organization could go through a sequence of several pilot projects, with each focusing on specific evaluation criteria. The work at the NASA Goddard Space Flight Center Software Engineering Lab (Basili & Green, 1994) is an example of a highly controlled sequence of pilot projects. There, each piloting step was part of a larger evaluation process based on the Goal-Question-Metric paradigm.

2.2 Organizational Change and the Pilot Project

A pilot project must also demonstrate that the Cleanroom "culture change" challenges can be addressed on a small scale prior to organization-wide rollout. There are three aspects to this culture change: introducing a "non-error" attitude in development, breaking the cycle of *ad hoc* developer testing, and defining the new role of testing.

Mills' phrase, "non-error as part of the company culture," neatly sums up the difference between the Cleanroom philosophy and the management mind-set still prevalent in software engineering. Cleanroom attempts to foster an environment in which software errors are treated as unlikely and rare, and in which software perfection is often attained. The culture change has numerous ramifications: bug counting, bug tracking, and bug hunting become less prominent activities (though they do not vanish altogether, humans being fallible) while team review and systematic reuse become more important. Thus the most important change that accompanies Cleanroom, and the one that is most pervasive, is the need for the company culture—including its incentive systems—to reflect the movement toward an expectation of quality.

The most challenging step in moving toward an expectation of quality is the replacement of ad hoc developer testing with team verification-based review. Most software developers are trained from their earliest experiences to create programs through a process of "iterative hacking." An initial design is created, and perhaps reviewed, but after that the programmer enters a private world dominated by the compile-test-debug cycle. New tools make this cycle rapid, but they do not necessarily improve the end product of it. That end product is unknown to other developers, is almost always larger in size than originally planned, and is usually delivered later than expected. Simply put, this process is not engineering. Programmers on Cleanroom teams are encouraged to forego this hacking mode of development, instead putting forth a succession of improved designs to be reviewed in small teams. While some projects have taken the step of trying to prevent hacking (e.g. though limited access to the compiler or operating environment), in most cases it is sufficient to ask programmers to try avoiding it for a single increment, after which they (typically) do not revert to their old ways.

A pathology exists in many software development organizations today. This pathology is fostered by the problem of bug-counting and its attendant incentives. Consider: if the purpose of testing is to find bugs, then a good tester is one who finds lots of bugs. The result: if the testers don't find very many bugs in a software system, is that good or bad? In addition, if testers are rewarded for finding lots of bugs, what is the incentive to the developers to reduce the bugs through inspection? The Cleanroom principles turn this around. The purpose of testing is primarily to measure system quality, so the burden of finding bugs falls primarily on the developers' team reviews. This achieves the numerous benefits of team review (cross-pollination, the best ideas

emerging, and so forth) and eliminates the adversarial relationship between development and testing.

2.3 Phased Introduction

Any examination of the Cleanroom historical experience cannot fail to perceive that Cleanroom is implemented in very different ways by each project that uses it. Some projects may combine functional correctness verification with traditional partition testing. Other projects will use inspections together with statistical testing.

Several papers have suggested how projects and organizations should go about selecting Cleanroom practices that are appropriate to their needs. Deck (1996) describes a process for using risk assessment techniques to choose those practices that mitigate the highest project risks earliest. A project that perceives reliability to be its highest risk will select more rigorous verification and testing techniques. On the other hand, a project that perceives time-to-market as its highest risk might use inspections rather than full verification.

Hausler, et al., (1994) proposed that a project or organization go through three phases in its adoption of Cleanroom With each phase, not only is rigor increased but so is the level of formal process documentation.

In either case, it is clear that no one size fits every Cleanroom project. Furthermore, the track record of Cleanroom is one of flexibility and adaptability rather than rigidity and orthodoxy.

2.4 Pilot Project Results

Pilot project results are numerous, and have been well-reported in the literature. Although few of these studies can be considered controlled experiments, they form a pattern of success both in terms of productivity and quality. Some projects (Linger & Mills, 1988; Selby et al., 1987; Green et al., 1989) have adopted a "strict" definition of Cleanroom. Other projects (Hausler, 1992; Tann, 1993; Johansen & Karlsson, 1996) take a more liberal view of what Cleanroom "is."

3. Organization-Wide Cleanroom Adoption

Many of the pilot projects may be fairly criticized as not representing large-scale projects because they were limited in scope and/or size, not on a critical path of development, developed by "high performance teams", or other factors that might not scale-up (i.e. human factors). The underlying issue is whether the quality aspects of Cleanroom hold when using it to develop large projects (dozens to hundreds of developers) that are inherently complex due to size, scope, and the involvement of many personnel. One of the focal points of this paper is the feasibility of using one or more Cleanroom techniques to improve the quality of software systems. The information that is provided is based on the experiences in the development of large projects that required many teams.

Based on our experience with the use of Cleanroom, pilot studies are important for gathering feedback on the practical application of a Cleanroom techniques such as incremental development. However, there are limitations in transferring the technology and lessons learned from a pilot study to large projects. For example, a pilot study may not be on a critical path in the overall

development of a system. As a result, there is more flexibility in the use of a formal technique because changes can be made without impacting the overall schedule of the system. This may not be the case when applying a formal technique in a large-scale environment. Typically, there is little room for schedule delays and changes may have major impacts on quality, resource utilization, and performance.

In large project development, the organization may not have the resources for developing "high-performance" teams that are typical of pilot studies. High-performance teams may be characterized as teams that feel empowered, committed, and have a sense of purpose (Mears, 1994). On a smaller scale, it is feasible to allocate resources in order to develop one or more high-performance teams. Typically, this is achieved in a pilot by allocating resources for training, mentoring, and monitoring a small number of teams. The same type of support may not scale-up when dealing with large projects because of geographic dispersion, distributed and parallel work, and the sheer magnitude of support required by the diversity of skills, experiences, and backgrounds of the organization's human resources.

3.1 Formal Processes and Organizational Goals

An organizational process encompassing formal techniques such as Cleanroom is necessary for achieving organizational goals (i.e. goals). Otherwise, the organization may experience a *bureaucratic, chaotic*, or *wishful thinking* environment where goals or a formal process may be lacking. What is needed to achieve the "Goal Attainment" state is a strong relationship between selected processes and organizational goals (e.g., Cleanroom and quality). This is shown in Figure

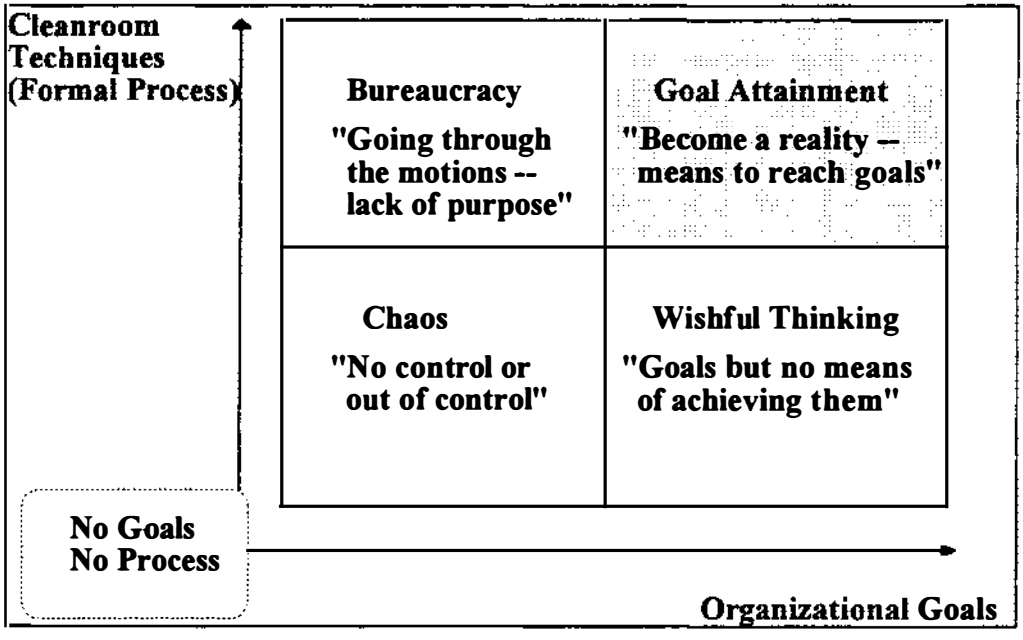


Figure 1. Organizational "States" Based on Goal Setting and Process Adherence

1 where the goals of the organization become a reality through established organizational processes.

Introducing Cleanroom does not just affect the development process but has major impacts on the organization structure. As a result, we have to move beyond pilot studies and take a look at organizational issues associated with large projects typical for the development of many software systems. The organization must have the capability for team-based development thus placing requirements on project management and the line organization. All the organizational issues that are inherent to teamwork apply to the introduction of formal methods such as Cleanroom because it is a team-based development process (Curtis, et al., 1994).

3.2 An Organizational Framework For Cleanroom Support

In a large study on the use of Cleanroom incremental development and teamwork, it was found that an organizational framework must be in place for the effective use of Cleanroom. The study was based on a large, multi-site telecommunications project involving approximately 500 people (approximately 400 team members and 100 project and line managers). The project involved

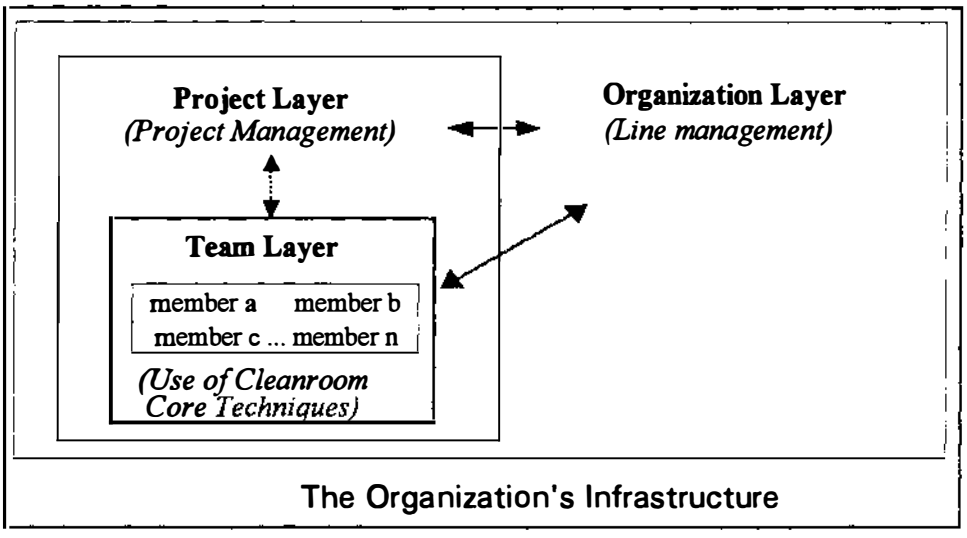


Figure 2. An Organizational Framework for Cleanroom Support

development work on a legacy system with real-time system components. The project life-span was two years during which it was incrementally developed with eight internal delivery points. Each increment expanded the previous one and as a result the functionality of the system evolved. A significant amount of the project effort was focused on integration/verification work. The Cleanroom approach was selected for this project to meet its goals of time-to-market reduction and quality improvement.

The study identified the need for a common organizational framework in order to promote a quality culture. The framework that was used as a basis for defining roles and responsibilities of

all parties involved is presented in Figure 2. The objective was to ensure that the organizational framework was in place in order to provide a common process that would integrate Cleanroom techniques into project development. The project, organization, and team components in the framework with defined organizational boundaries had to be considered individually and as a whole in order to support common goals.

The Cleanroom team and its members is viewed as the software engineering component of the framework. This component has been the focus of much of the Cleanroom research to-date and as a result is perhaps the best understood in terms of formal processes and techniques. What hasn't been emphasized, however, is that team "buy-in" of Cleanroom techniques is crucial to the success of the project in terms of meeting schedules and quality goals. When teams do not understand the objectives of a Cleanroom technique, the implementation of the technique may become added work with no improvements. For example, a team member may not understand the team review process and its impact on quality. Then, the review becomes meaningless (no attempt for semantic/syntactic understanding) and adds labor hours to the team's efforts.

The Project component is shown to be connected to both the Organization and Cleanroom teams suggesting two important concepts --the oversight role project management must play in the effective use of Cleanroom and the need for an interface across organizational boundaries. Project management must understand the relationship between Cleanroom process adherence and its impact on quality. This means that the project must allocate sufficient resources and time to support aspects of Cleanroom process adherence (e.g., team reviews).

The organization (i.e., line management) is an integral component of this framework as line responsibilities include ongoing support of project and teams in terms of process improvement, organizational maturity, and personnel development among other factors.

Both of these components must understand the benefits of using Cleanroom; otherwise, the formal techniques of Cleanroom are paid lip service but are not fully supported by the organization. This lack of support trickles down throughout the organization.

3.3 The Organization Infrastructure

What was used as the basis for obtaining "buy-in" by all parties is the infrastructure of Figure 2. This provided a foundation upon which Cleanroom techniques were supported to attain quality goals. An infrastructure that has been proposed by Janzon et al. (1996) and used in the study is a combination of four organization improvement (OI) concepts that fit together in a synergistic fashion. Each OI concept is addressing a different aspect of the infrastructure that is typical for complex software projects. These OI concepts are important because Cleanroom provides a means for performing development activities but does not address organizational aspects of team-based development. The OI concepts are:

Organizational soft skills - Management and teams need to be trained in effective group dynamics including decision-making, communication, participation, and problem-solving, among others.

Team process model - Teams need a common framework (i.e. team methods, roles, and responsibilities) in order to work together as high-performance teams. In the study, teams were

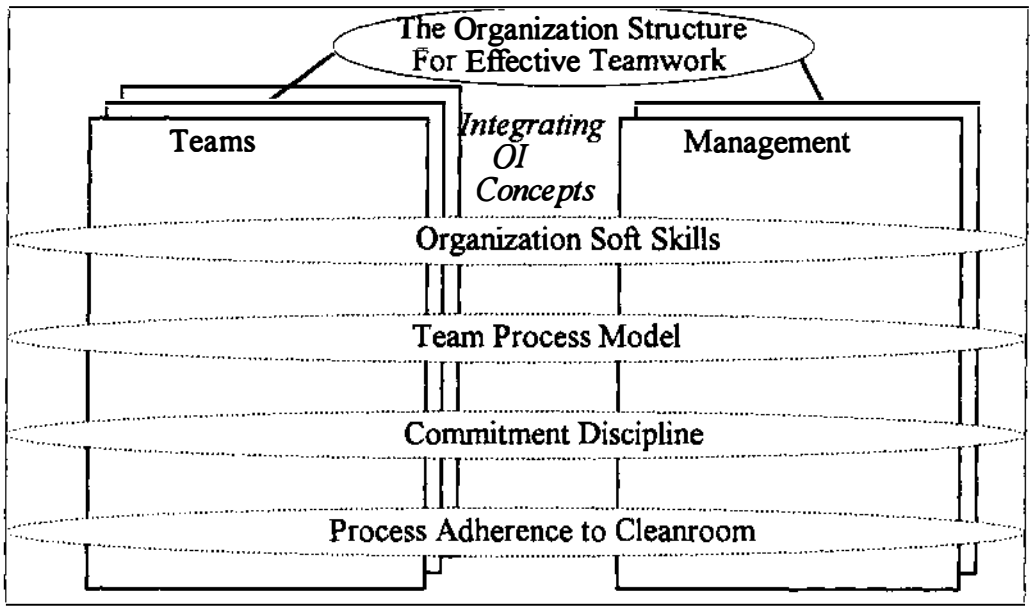


Figure 3. The Organizational Improvement (OI) Concepts

provided an assignment process in which team activities were performed in a predefined order (Becker et al., 1996)).

Commitment discipline - To achieve high-performance teams that work effectively with management, work allocation must be based on commitments in order to ensure responsibility and accountability. Both management and teams must be able to make commitments to produce deliverables as agreed upon in terms of quality, cost, and time and to adhere to organizational processes.

Process Adherence - It is meaningless to introduce Cleanroom -- or any type of formal process without an organizational commitment to its use. Thus, it is crucial that the organization selects what formal processes are to be followed by the teams (i.e. one or more Cleanroom techniques) based on its organizational goals and abilities. These types of process agreements must be made visible to all parties in the organization and then process adherence is an organizational issue not only the responsibility of the teams.

Figure 3 is descriptive of how each OI concept supports the common goals of the organization.

3.3.1 Organizational Soft Skills

The soft skills training is important in ensuring that management and teams work together as effectively as possible. Management soft skills include coaching team members on the use of formal methods in order to use Cleanroom techniques and supporting technology in a team environment. Project management soft skills include motivation and commitment to the successful completion of the project; whereas, line management soft skills include long-term mentoring of individuals in the organization. This dual focus of project-specific and long-term organizational development ensures that resources are available to satisfy existing and future requirements.

There are team-related soft skills that are required for all participants within the whole organization. These skills typically include conflict resolution, consensus-building, goal-setting, and others. To effectively apply Cleanroom, it is important that team members are able to problem-solve as individuals and work together to find feasible solutions. Soft skill training includes effective communication, for example, so that team members are able to give and receive review comments on their work in order to hold effective Cleanroom reviews.

3.3.2 Team Process Model

A Team Assignment Model was used in the study by approximately 400 developers as the underlying, organizational process necessary for attaining quality goals. The objective of this model would be full support of the Cleanroom development techniques. Figure 4 is an illustration of Cleanroom development support using the model. However, the Cleanroom techniques that were initially supported included incremental development and team reviews. This "phased approach" allows for a more controlled environment of introducing Cleanroom as a team-based method. Continued efforts have initiated the introduction of formal specification into the team assignment lifecycle. The implementation of the Team Assignment Model played an important role in:

- Stabilizing teams, projects, and organizations because roles and responsibilities were well-defined.
- Establishing a communication network for effective top-down, bottom-up, and cross-functional information flow.
- Empowering teams, project and line managers and the organization as a whole to meet quality, performance, and productivity goals.

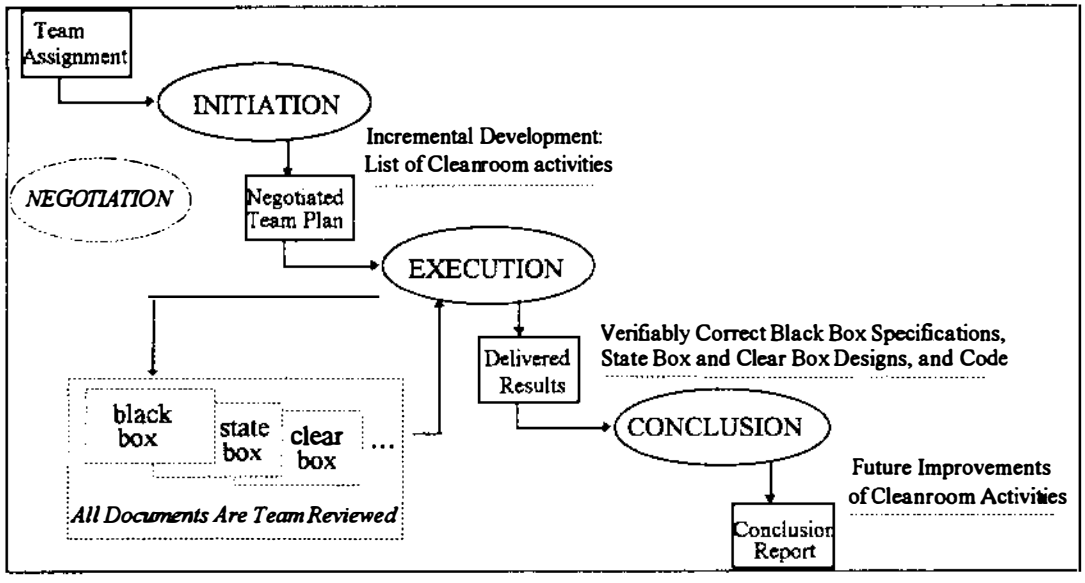


Figure 4. The Team Assignment Model

A team assignment is performed in three phases: initiation, execution, and conclusion. These phases are referred to as the *Team Assignment Model.* They are defined as:

Initiation - The purpose of the initiation phase is to start work on an assignment. The team assignment activities are launched at a "kick-off" meeting at which time the team receives its assignment. The team members analyze the requirements of the assignment, define common plans and goals, and establish work methods and team routines for assignment execution. Based on the outcome of this activity, the team may need to negotiate with management the terms of assignment execution due to a lack of resources, time, expertise, or other factors. It is during this initiation phase that internal and external commitments to the team plan are made.

Execution - The purpose of the execution phase is to begin work according to the Team Assignment Plan and the process agreed upon, following the schedule of tasks, and using resources effectively in order to achieve the goals. The work is actively controlled by tracking and assessing task completion. Renegotiation on the Team Assignment Plan may be initiated when management or teams deviate significantly from the agreed upon plan.

Conclusion : The purpose of the conclusion phase is to compile experiences of the team effort and communicate them to management for future improvement. The assignment is brought to closure by documenting team experiences and improvements regarding the effectiveness of technical, process, and quality aspects of activities that were performed. The conclusion phase provides an effective means for communication between teams and management which is necessary for ongoing performance evaluation and improvement.

3.3.3 Commitment Discipline

The commitment discipline provides a foundation for ensuring purpose and accountability throughout the organizational structure. The underlying concept is that teams have control over their work assignment by developing a team assignment plan that is mutually agreed upon by management and teams. The team is responsible for providing input on the team's capabilities of meeting deadlines, producing the functionality and maintaining high-quality given their team resources. This decision-making ability also means that teams are responsible for their input, execution, and completion of the assignment as defined by the Team Assignment Model. This a very different approach than management giving teams no decision-making power or flexibility in completing team assignments. This concept of teamwork is necessary for developing an organizational commitment culture from bottom-up.

In the study, one aspect of the commitment discipline was demonstrated by the negotiation process that is part of the Team Assignment Model. Teams had an opportunity to negotiate their team assignment plans with management in terms of functionality, resources, time, and process adherence (including the Cleanroom team review process). The impact of process adherence on the organization became readily apparent as each team scheduled a team review for approximately every 40 hours of work. It became an organizational commitment to ensure that these reviews were not reduced or eliminated even though the overall project hours could be reduced.

3.3.4 Cleanroom Techniques

Ideally, the application of all Cleanroom techniques would result in high-quality software systems if practical guidance was available for handling the complexity of large project development. What was discovered during the large-scale implementation of the Team Assignment Model is that the use of Cleanroom techniques requires a practical approach to the use of Cleanroom in the team assignment lifecycle. The use of team reviews, for example, required training, mentoring, and monitoring the team review process. Initially, it was assumed that the teams understood a team review process. As a result, team reviews ranged from a syntax checking session (particularly for new employees that had little or no teamwork experience) to high-quality reviews. Once an organizational policy was established for holding team reviews (i.e. every 40 hours of development work), the size and complexity of the material being reviewed allowed for intellectual control by the reviewees. Table I illustrates the review process that was recommended to the teams to be used during the team assignment lifecycle.

Review Components	Description
Team Review Strategy	Team establishes a policy for when a review is held. Includes participant preparation time and the terms for holding a review (size, complexity, functionality, quality).
Author Preparation	Describes what preparation steps are taken by the author of the reviewed material (includes a policy for preparing an agenda, scheduling the review, inviting external resources etc.).
Participant Preparation	Describes the preparation steps that are taken by the reviewees.
The Review Meeting	Policy for reviewing the material. Same policy as a structured meeting where the author presents the material for a fixed time period and the participants identify issues, errors, etc. that are recorded in a review log.
Review Cancellation	If the participants are not prepared, the author cancels the review and reschedules it.
Review Log	All review comments, issues, and errors are recorded in a review log that is used to ensure follow-up on completeness and correctness issues.

Table I. Team Review Guidelines

At the higher levels of specification and design, the intention of using team reviews was not to provide "proofs of correctness" which was not feasible -- but to achieve other organizational objectives. The team review process was integral in ensuring that all material produced by a team had been reviewed by other team members to focus on error prevention early in the development process. Other objectives included promoting team commitment by ensuring "buy-in" by the team on the quality of its deliverables. Also, in the early stages of teamwork, holding team reviews provided an opportunity for all team members to gain an understanding of what needs to be done to successfully complete the assignment, to maintain intellectual control throughout the team assignment lifecycle, and to develop technical skills within the team (which became very important for supporting cross-functional teams).

These OI concepts may be viewed as organizational building blocks for successful team-based projects and for long-term development of an organization's personnel. The organizational objectives for using such an approach include shared technical competence, improved communication network, strengthened commitment, and process ownership, among others.

3.4 Common Barriers in Team-based Development

Table II identifies common barriers found during the study when an organization moves towards team-based development. These barriers are typical of an organization that has "good intentions" of introducing teamwork but has not developed the mechanisms for its effective implementation.

Common Barriers	Description
Team Leader Role	Team leader role must be clearly defined as a team coordinator not a supervisor or administrator.
Inefficient Team Meetings	Teams must establish effective team meeting routines in order to use their time and resources effectively.
Lack of New Member Integration	When a new member is added to a team, the team will need to integrate the team member by establishing new team routines, roles, and responsibilities.
Lack of Reward and Recognition	The organization infrastructure must be established for supporting both individual and team reward and recognition.
Lack of Two-Way Communication	Communication channels must be developed among all parties in the organization.

Table II. Common Organizational Barriers in Team-based Development

The next step in integrating the Cleanroom techniques within the proposed organizational framework is to select additional techniques and evaluate an effective approach for their introduction. For example, the formal specification of system behavior and stepwise refinement could be introduced together in order to provide a foundation for design and certification. But, before this can be done effectively, there are issues that need to be studied further. These include:

- *Practical guidelines for using a formal specification language that meets the needs of the organization.* The specification language may be Mill's box structure notation (Mills et al., 1989) or it may be more feasible to use an existing organizational approach to minimize the learning curve and to support reuse of existing specifications. (For example, graphical notations, sequence diagrams, etc.)
- *Standards in the use of the specification language/format, accessibility, etc.* Standards are needed to support the expansion of specifications into designs and code. When integrating a Cleanroom technique, such as formal specification, it must be integrated into the existing work processes. This may not be an issue in pilot studies when the short-term focus is on the successful completion of the pilot in isolation of organizational work processes. (An example

of this has been when the specification was placed in a separate document that was not readily linked to the design document making correctness verification a difficult activity).

- *Identification of existing tool support.* The organization should review existing tools as it may not be necessary to develop new tools to support Cleanroom techniques. The main issue is not to add complexity to the existing development environment -- but to try to focus on "added value" of Cleanroom techniques to the work processes. (An example of this is to develop templates with supporting version control in an automated environment to support a natural progression from specification, design, redesign, to code.)
- *Reusability support.* For legacy system, guidelines must be established for using Cleanroom techniques on the existing components. The project may not be able to support the cost and resource requirements of applying Cleanroom techniques to all existing components. (An example guideline would be -- if more than 50% of a specification/code unit is modified, then it is reversed engineered using the Cleanroom techniques.)
- *Long-term strategy for implementing the Cleanroom techniques.* Experience has shown that the benefits of using a phased approach to the introduction of Cleanroom techniques is useful in providing a controlled environment for planning, tracking, and assessing the organizational impact. Without a long-term strategy for the use of Cleanroom techniques, the benefits of one technique may be limited by the problems encountered when another one is introduced.

4. Summary and Future Research

In our experience with using Cleanroom in both large and small projects, we have noted certain critical success factors.

- *Risk/Goal evaluation*: It is important for a project or organization to consider what their goals are in selecting Cleanroom and in choosing individual practices. These may be stated as business goals (e.g., improve time-to-market) or as risk mitigation (e.g., reduce the risk that the product will have unacceptable quality). Only after the overall business goals have been identified and prioritized can an adequate job of process tailoring be done.
- *Metrics and measurements*: Although metrics are a significant part of any software process improvement effort, there are no metrics that are specific to Cleanroom. Cleanroom teams must integrate metrics into the phased adoption of Cleanroom practices, in order to measure and justify the return on that investment. Critical to this process is the collection of baseline (pre-Cleanroom) metrics.
- *Integration*: Cleanroom must be integrated into an environment of existing tools, techniques, and processes. It is important to leverage prior investments by adding key Cleanroom practices to that environment.
- *Long-range planning.* It is very difficult for any individual project manager to think beyond the current deliverable. However, for organization-wide adoption of Cleanroom, it is important to invest in the infrastructure to support gradual but consistent movement toward the Cleanroom goal. This investment cannot be justified at the project level, but must be supported by a long-range plan for process improvement at the organization level.

We have suggested some techniques for piloting Cleanroom and, from there, moving toward organizational adoption. These techniques are based on the experiences of projects, but there is a great need for more projects to report on their results.

5. References

Arnold, P. (1995). Capability Maturity Model for Software Goals Mapped to Cleanroom Software Engineering Process, *STARS CDRL C012-001*, STARS, Washington, DC.

Becker, S., Janzon, T., & Nilsson, B. (1996). Establishing Effective Team Routines for Cleanroom Support, *Proceedings of the 1st Cleanroom Workshop, ICSE 18,* Berlin, Germany.

Cobb, R.H. & Mills, H.D. (1990). Engineering software under Statistical Quality Control. *IEEE Software*, (November), 44-54.

Curtis, B., Hefley, W. E., Miller S., Konrad, M., & Bond S. (1994). Increasing Software Talent. *American Programmer: Peopleware Part I.*, Cutter Information Corp.

Deck, M. (1996). Cleanroom Practice: A Theme and Variations. *Quality Week '96*. Available in softcopy from http://www.csn.net/~deckm.

Dyer, M. (1992). *The Cleanroom Approach to Quality Software Development*, Wiley.

Green, S., Kouchakdjian, A., & Basili, V. (1989). Evaluation of the Cleanroom Methodology in the Software Engineering Laboratory. *Proceedings of Fourteenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center*, (Greenbelt, Md., November), 1-22.

Hausler, P.A. (1992). A Recent Cleanroom Success Story: The Redwing Project. *Proceedings of the Seventeenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center*, (Greenbelt, Md., December) 256-285.

Hausler, P.A., Linger, R.C., & Trammell, C.J. (1994). Adopting Cleanroom Software Engineering with a Phased Approach. *IBM Systems Journal*, 33(1), 89-109.

Janzon, T., Nilsson, B., & Becker, S. (1996). A Team-based Organization Structure For Support of Cleanroom Development. To be published in *Cleanroom Software Engineering Practices,* IDEA Group Publishing, PA.

Johansen, Ø., & Karlsson, E-A. (1996). Experiences using Cleanroom in Arendal, Norway.

Linger, R. (1994). Cleanroom Process Model. *IEEE Software,* 11(2).

Linger, R. & Mills, H. (1988). A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility. *Proc. 12th International Computer Science and Applications Conference* (October).

Mears, P. (1994). *Organization Teams: Building Continuous Improvement.* St. Lucie Press, Delray Beach, FL.

Mills, H.D., Dyer, M., & Linger, R.C. (1987). Cleanroom Software Engineering. *IEEE Software* (September), 19-25.

Mills, H., Linger, R., & Hevner, A. (1986). *Principles of Information Systems Analysis and Design*, N.Y., Academic Press.

Paulk, M., Curtis, B., Weber, C., & Chrissis, M.B. (1995). *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison-Wesley, Reading, MA.

Selby, R., Basili, V. & Baker, F. (1987). Cleanroom Software Development: An Empirical Evaluation. *IEEE Transactions on Software Engineering*, SE-13(9), 1027-1037.

Tann, L-G. (1993). OS32 and Cleanroom. *Proceedings of 1st Annual European Industrial Symposium on Cleanroom Software Engineering* (Copenhagen), 1-40.

Holistic and Business Approach to Software Metrics

Sandhiprakash Bhide

SQA Engineering Manager
Transmission Test Software Engineering
Tektronix Inc.
Beaverton, OR 97077.

Keywords

Software Metrics, Top Level Objectives, Process Metrics, Product Metrics, Objective/Goal - Strategy-Metrics derivation.

Abstract

Software Metrics have been used in the software industry for many years, but upper management has failed to see the impact of instituting Software Metrics Program on company's bottom line, market share, revenue, perceived quality or efficiency. The industry has focused only on software rather than software products and services as a whole. Software Metrics have been developer-centric rather than customer-centric. The industry has also focused on creating generic software metrics rather than choosing metrics suitable to a given product, project, or market segment. The result has been a complete disconnect between Software Metrics and the customer's need for quality product/services and company's need for profitability.

The paper proposes a method by which metrics are derived in a top-down fashion from a combination of customer needs, market needs, company needs, and so on. Each individual working in the system, from junior engineer to the CEO, is able to see clearly the link between the metrics they have chosen and whatever end-result they expect. The outcome is result-oriented metrics that are of value to both the customers and the company. Instead of sub-optimizing only software process, this approach provides a system, business, and holistic approach and optimizes the entire business process.

Brief Biographical sketch

Sandhiprakash (Sandhi) Bhide is presently working as SQA Engineering Manager for the Transmission Test Software Engineering group at Tektronix Inc.. Sandhi has done extensive work in the area of testing, software processes, metrics, SEI/CMM, and ISO9000 for software. He has over 16 years of experience in the area of hardware and software. He has worked for several high-tech companies in the Portland area and has helped them to establish software processes. He has also published and presented several papers in the area of software quality and software metrics. His research interest includes strategic planning, software process, metrics, system requirements engineering, and software tools. He is a member of IEEE Computer and Communication Societies and holds a Bachelors of Engineering from University of Pune, India and Master of Science from University of Wyoming, Laramie, WY.

The author can be contacted at: Tektronix Inc., P.O. Box 500, MS: 50-310, Beaverton, OR 97077-001.
Phone: (503) 627-5407, Fax: (503) 627-3072, E-mail: bhides@mdhost.cse.tek.com.

Holistic and Business Approach to Software Metrics

Sandhiprakash Bhide

"You cannot improve, what you cannot measure" - Lord Kelvin

"Why measure if you don't plan to improve or know what to improve"

"What good is measurement if process is not stable?"

"Why are you in business, if not for satisfying customers and making money"

"Everything is Connected" - Ancient Wisdom

1. Introduction

The ultimate goal of producing software products or for that matter any product is to provide solutions to customer's problems and in doing so generate handsome profits to support people, infrastructure, operation, processes and so on. This, in turn can help create even more products and services to solve customer's problems. In a competitive market where many companies are trying to solve customer's problems, the company that really wins is the company that can sustain its performance in keeping customers satisfied and continually generate handsome profits.

In order to achieve this goal, a company must consistently out-perform the competition in the following ways:

- Meet or exceed customer's as well as user's expectations in solving their current and future problems (or at least provide an easy transition path to solve future problems),
- Provide solution at a price that truly represents the value of the product,
- Provide solution when it is needed, or in another words provide solution in a time window when the perceived value is highest, and
- Keep highest operational efficiency by using appropriate and optimized processes for producing products and services[1].

(The first three bullets deal with software business strategy while the last bullet deals with software policy, processes, methodology, and tools. More than often, software businesses focus on part of the last bullet related to software process, while the first three bullet items do not get much attention at all. The paper focuses on both measuring business strategy or process as well as measuring all software related processes in a holistic fashion.)

I will call these 4 bullets the "Top Level Objectives". These factors stated above should be the prime drivers for creating or choosing metrics. This means the metrics derived from these 4 bullets should provide the right information to people so proper decisions can be made to improve processes. A constant focus on these Top Level Objectives will allow a company to always out-perform. Furthermore, the people at the lowest level of the organizational hierarchy should be able to see a clear connection between the metrics they are using and the intended

[1] Software Engineering Institute (SEI)'s Process Maturity Model [1] addresses only software process. The model focuses on five levels of software process maturity evolution, and it is an important first step in the improvement of business process. At the same time, it is important to note that this is only a partial solution to the real problem. For example, it does not address software documentation process, software update process, or support process. The point is that it is software products that sell not just software.

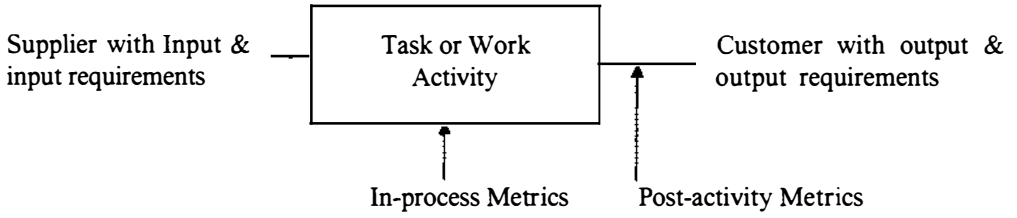
result of outperforming the competition. Finally, metrics cannot be generic from project to project, but rather tuned to the end result expected which might be different for different projects. The metrics also need to be tuned to the organizational level so they have meaning for the people using them.

1.1 Definitions

Following are some of the definitions used in this paper. They are presented here for consistency and clarification.

1. **Strategic Key Issue:** These are those 2-3 key issues that prevent a company from reaching its Top Level Objectives. These are the not low level issues. These are the high level issues that affect the entire company. These are the issues which when resolved can show a dramatic or break-through improvements in company's performance or in meeting the Top Level Objectives. These are also issues that affect multiple divisions, departments, or disciplines (cross-functional impact).
2. **Process:** A set of conditions, or a set of causes, which work together to produce a given result. In case of software process. code; software training; user documentation are the results. Alternately, the process can also be viewed as a sequence of tasks or work activities (some performed serially and some performed in parallel) that produce a given result. Since the word process means a *system of causes*, the process we choose to work with can be a person, a team, a way of carrying out measurements on the process, a machine that does some work, or a mental activity of visually checking the code. *A business is the highest level of process abstraction.*
3. **Process Stage:** A process stage is defined as a process partition where each stage is viewed as a state of evolution of the product. In case of software, design; test; and build are few examples of process stages. A process stage can be further decomposed into further process stages in an iterative fashion. A process stage and process are used synonymously. Following is an example of an elemental process stage.

Process Stage Diagram



This process stage model is very similar to the ETVX (Entry-task-validation-exit) model described by Radice et. al. [2].

- **Suppliers:** These are entities providing inputs to the process in a form that a process stage can make use of. This is a general term that can be used for both internal as well as external suppliers.
- **Customers:** These are entities that receive the output or services from the process. The customer can dictate in what form the output is required. Post-activity metrics define how faithfully the customer's requirements were met. Customer is a general term that can be used for both internal and external customers.
- **Task or Work Activity:** The transfer function that converts input to output. It is the efficiency of this conversion that is measured with the In-process Metrics.
- **Metrics:** There are two type of metrics. Post-activity metrics and in-process metrics.

- **Post-activity Metrics:** These define the effectiveness with which customer's expectations are met as a result of executing the process. The customers can be internal or external.
- **In-process Metrics:** Define the key characteristics of the process. These give you insight into the process. These are measurements done on the process within a process stage. Measuring these metrics provide the necessary data to plan further improvement.

1.2 Current State of Metrics Research

The metrics presented in the literature today deal only with software code, design, testing and so on. Yet, the software that gets in hands of the users is more than just code. Other externally visible items include documentation, physical media, services, after-sales phone and modem support, follow-on products, updates, maintenance contracts and so on. Internally, there are various other processes such as consulting, marketing, sales, advertising which are also very important and are necessary. But only in rare cases, these are considered as the candidates for collecting metrics. It is quite possible and in many cases it is indeed true that these factors are sometimes more important than the software code itself[2] and yet no metrics are captured for these processes.

Following are the issues with metrics currently used.

1. The metrics presented in the literature today are very low level (nearer to the code), used locally (scope is limited to a module), and do not present any direct or intuitive connection to becoming more competitive in the market place. For example, McCabe's Cyclomatic [3] Complexity deals at a module level based on number of paths from entry to exit. Henry-Kafura [4] information flow metric deal with information flow between modules. The scope is quite local, and at most between two modules. For example, a person who works in the process cannot possibly see how keeping low cyclomatic complexity can support the Top Level Objectives or lead to better competitiveness and company success.
2. These metrics tend to bring local sub-optimization with almost no regard to what the product is trying to do as a whole. Optimizing metrics at a module level does not necessarily mean the product is going to do better. In short, the module level quality does not guarantee quality of the product as a whole. Furthermore, it does not guarantee more user satisfaction or making the company more competitive. For example, reducing the complexity of a module has a very little product-wide impact and may represent only a small portion of the Top Level Objectives.
3. They do not provide any reasonable connection to the Top Level Objectives.
4. In many cases, they are chosen because they are easy to calculate, but have a limited use, e.g. defects per NCSL - non-commented source lines of code. NCSL has done more harm to software engineering than good as it is easy to measure but thoroughly useless measure.
5. The currently available metrics only relate to a subset of life cycle activities such as coding, testing, and in some cases design. What is missing from the current scheme of metrics is the cohesive structure or a framework that binds these various metrics together. Currently available metrics do not fit together as pieces of an overall puzzle.
6. Most metrics implementations rarely solve a macro-level problem. For example, internally the macro problem might be that of improving the operational efficiency of the entire software life cycle. Most metrics deal with a specific problem such as module complexity.

[2] There are several real life examples, you may have encountered. I will present one such example to illustrate my point. Few years ago, I came across a case in the Electronic Design Automation (EDA) industry's in the Integrated Circuit (IC) market segment. Over and over the customers told us that their #1 priority was to get more and newer functionality so they could produce denser chips (even if the software was not very robust and the quality of the software was not up to the mark). Similarly, for enterprise software, supportability and portability sometimes override other attributes such as functionality.

1.3 What Do We Really Need for Metrics?

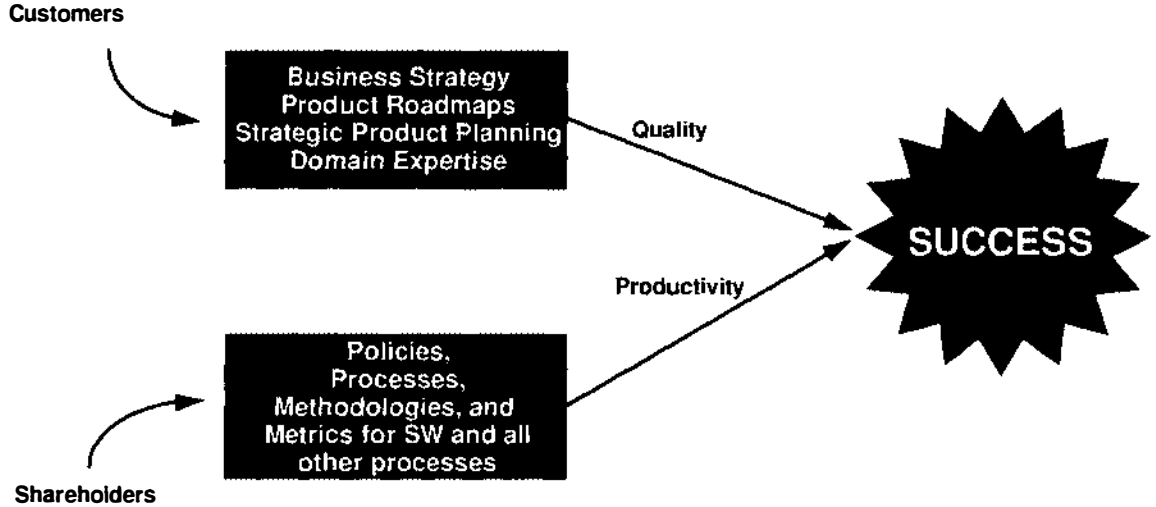
The industry uses various development processes and product life cycle models. Since the processes used are inherently different, the associated metrics must be generalized. Thus, the same metrics cannot be used in all situations. It also makes sense to integrate these metrics into the process since they define the road-map for evolution of products or services. Following are the desired aspects of metrics:

1. They become an integral part of the life cycle process and form a cohesive structure. In that context, they are not unique and depend on individual processes. The bottom line is to use metrics which make sense to the needs of the business.
2. Metrics presented at various levels of organizational hierarchy must make sense at that level. This would allow various levels of management to get metrics data that is relevant to them and not data printed on stacks of paper. If needed, detailed information should be available.
3. The identity of individuals in the process should remain local which in turn would help in gaining metrics acceptance. This would avoid metrics being used for performance appraisals & allow people to become responsible for their own actions. It will also build trust and means for self-improvement.
4. The metrics need to be integrated into the process and in a framework so they can be successively derived and driven by the Top Level Objectives.
5. Two types of metrics are needed: one set that defines the process characteristics (in-process metrics) and a second set of metrics that define the outcome of the process (post-activity metrics). When used in a cause-effect relationship, these metrics would allow information for process improvement.
6. Tools should be built around the cohesive process-integrated metrics so they can talk to each other.
7. All metrics when used together as a set and when connected with each other would provide enough information to improve operational efficiency and company's competitiveness.
8. One should be able to roll up metrics data and provide metrics suitable at that organization level.
9. Only a few and key metrics need to be captured because they make sense and support the Top Level Objectives. If the metrics do not support the Top Level Objectives, then they should not be captured.

2. Software Business Models

To understand the concepts presented in this paper, two models have been used. They are not orthogonal models and they represent the same thing in two different views. Each model presents the business view. These are models presented simply to provide a context for holistic approach presented in this paper.

2.1 Quality and Productivity Model for Software



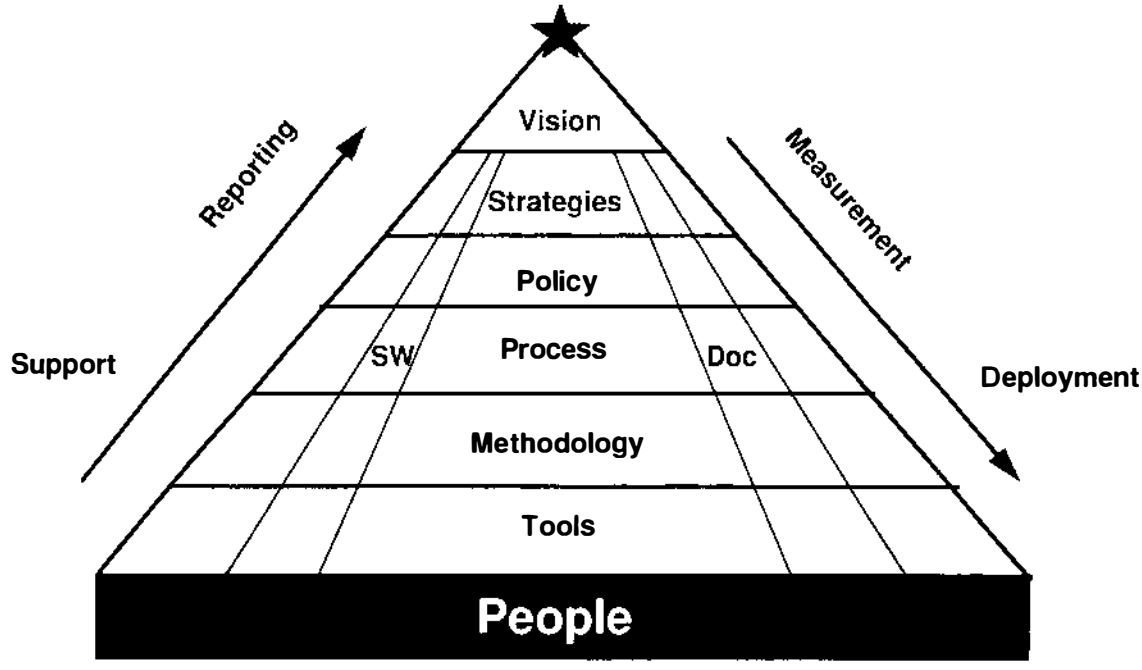
The Quality and Productivity Model is a very simplistic model but provides a clear view of what is needed to succeed in the market place. It has two components. Typically, customer's needs and problems drive the Business Strategy, Product Road-maps, Strategic Planning and so on. This information enables companies to choose and to define market segments they want to play in. If a company executes well in this area and solves customer's problems, it means that the customers are receiving a quality product.

The second component is typically driven by the shareholders who look for efficiency and cost reduction in the processes, whether they are software processes or any other processes used in the software business. If they are under control and optimized, it means that the processes are productive and healthy.

For a software business to succeed, both components need to operate well. This means both areas need to have metrics in place which could be measured, tracked, and used for improvement. Typically in software industry, the focus is only on the second component and even there, only software process receives the attention. Even though software is an important piece of the solution, it is by no means a complete solution to achieve the Top Level Objective.

As the picture shows, both strategy and all other processes supporting that strategy are needed to make a company successful. It indicates that there needs to be a complete holistic view of how metrics are utilized and deployed. Metrics also need to be in place for all the processes used in the company.

2.2 Pyramid Model or Hierarchical Deployment and Support

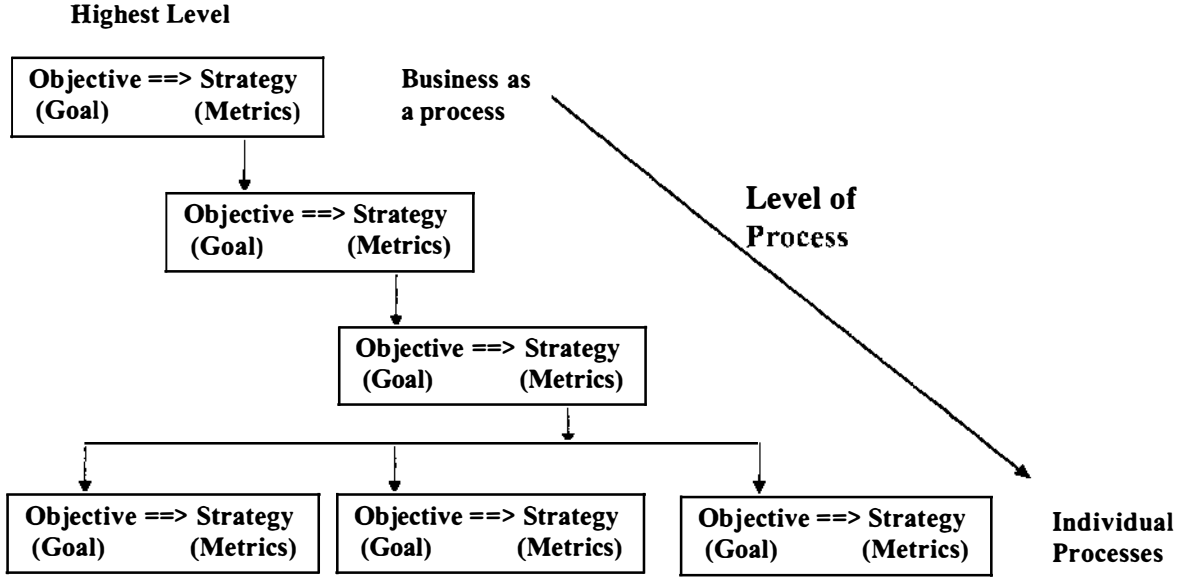


At the highest level of the pyramid, a company vision is developed and then associated strategies are developed to achieve that vision. Policies, processes, methodologies and tools are then developed or fine-tuned to support the vision and the strategies. In this framework, software is only one process and usually metrics are developed for this process. At the same time, there are several other processes for which metrics need to be developed. The measurements needs to be performed at each layer of the pyramid. The following model presents a view of how the metrics at each level need to be connected including software metrics.

2.3 Objectives/Goals and Strategies/Metrics:

Following diagram shows the Strategy and Objective linkage that is very key to the discussion of the holistic approach of defining metrics. With this approach, metrics at each level are generated from the metrics at the layer above.

Refer to the Pyramid model previously discussed and consider it as unified business process. The vision tends to be a global statement that cannot be measured. What can be measured is a clear set of objectives associated with the vision and the corresponding strategies. Individual strategies for each function, department, discipline, or person can be derived from this highest level business process through successive derivation. The advantages of this scheme are very clear. The metrics at each level of process abstraction are linked and directly relate to the business vision and strategies and therefore would be key in achieving the Top Level Objectives.



Objective is a statement of the result expected, while the Goal is a numerical value or quantification of the result expected. Strategy is a description of the means by which the objective is achieved[3], while metrics is a measure of how well the strategy was executed or they measure the success of the strategy. The strategies at the highest level and the associated metrics, become the objectives and goals at the next process stage in the hierarchy. At that level of process, strategies and metrics are developed which become the objectives and goals of the next process stage. This scheme continues from the top of the organization to the bottom level of the organization and can be deployed in different areas of the processes, such as software, documentation, and so on.

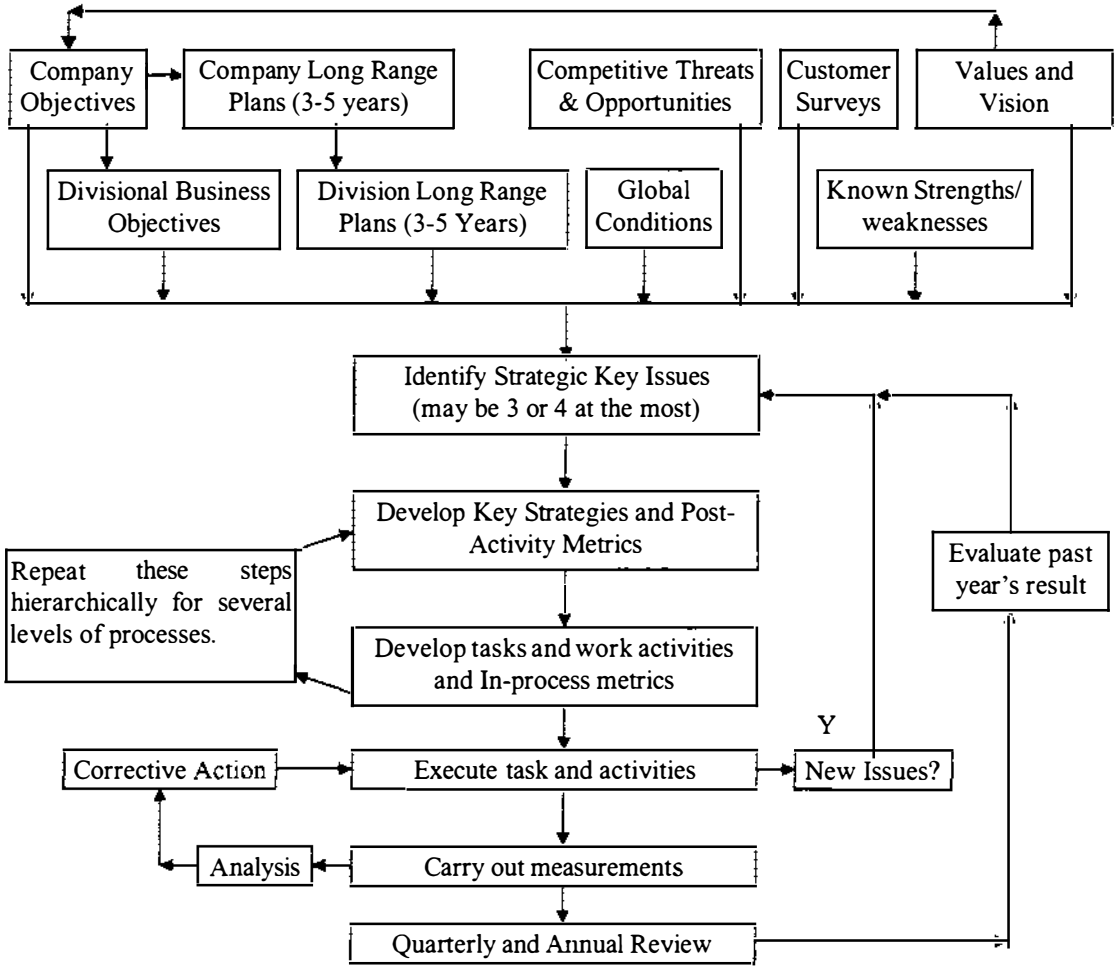
Not only the strategies, polices, and processes are developed from the company's top vision, but the metrics at each level are also derived from the Top Level Objectives.

[3] For the purposes of this discussion, the word *Strategy* is used here in a generic manner (the means by which objectives are achieved). Normally, the word is used at higher levels to mean business strategy. In this paper, the word *Strategy* means strategy at high level, but could also indicate other means such as policy, processes, methods, and tools. Although initially confusing, it allows visualization of the Objective-Goal and Strategy-Metrics hierarchy (since the terms used at any organizational level are still the same). In real life, Management By Objectives (MBOs) are used in a similar fashion. Every individual irrespective of the organizational level he/she is at, defines a set of objectives and means to achieve those objectives. They are successively translated from the highest level of organization to the lowest one.

2.4 Key Concepts for Creating Metrics

Following are some of the key concepts used in creating metrics in a hierarchical fashion:

1. Metrics defined in the process-integrated metrics framework are derived from Top Level Objectives.
2. A business is the highest level of process abstraction or process stage.
3. Metrics at lower level are derived from metrics at the higher level of process abstraction through a process described in the previous section.
4. The focus is to have only a limited number of key metrics that relate to the Top Level Objectives. The number of metrics is strictly a function of what are you trying to achieve (i.e. Top Level Objectives).
5. The success of metrics deployment depends on how the cause-effect relationship between the in-process metrics and the post-activity metrics is used.
6. Metrics and the metrics definition must be updated based on the business model and the business. The metrics are not generic. They change with changing needs.
7. Metrics must be constantly monitored and used to improve the process.
8. The metrics defined through this process primarily reflect user's/customer's needs (instead of just developer's needs). The developer's needs are taken into consideration, as one of the customers.
9. Objective/Goals and Strategies/Metrics are chosen based on Strategic Key Issues facing the company and the customers/markets. Following diagram shows various factors that can be used to define the strategic key issues and a simple process flow that shows how the post-activity metrics and in-process metrics can be defined. This process can be repeated several times to come up with the metrics that are appropriate at various levels of process abstraction.



The strategic key issues are defined by taking into consideration several inputs as shown in the diagram. Through brainstorming, all the issues are distilled to 3-4 top key strategic issues that could prevent the organization from achieving the Top Level Objectives. Once they are clearly defined, the associated Objectives and Goals are defined. For these Objectives and Goals, Key strategies and Post-Activity Metrics are developed. These Key Strategies are further developed into tasks, work activities and In-process Metrics. The same Key Strategies and Post-Activity Metrics become the Objective and Goals for the next level of process abstraction and this process repeats till it reaches the lowest level of organization.

There are two important aspects of the method described above:

- Metrics review should be conducted regularly to make sure no new issues have surfaced. If new issues come up, then they need to be addressed. New metrics may also be needed at this stage. In this case, the change would also permeate through the entire process chain. The review can occur quarterly, yearly or as often as needed. It depends on how fast the market changes.
- Both in-process metrics and post-activity metrics need to be analyzed regularly so that tasks and activities can be changed and corrective actions can be taken.

3. Examples of the Strategy -> Objective Linkage

Following sections describe an example that elucidates how metrics can be derived from highest level of organization to the lowest level of organization (hierarchically) using the Top Level Objectives. It is important to note that in reality, several such tables would exist based on the group you are involved with and your role in supporting the Top Level Objectives. All the metrics are connected and person at the lowest level of organization can see how his/her metrics support the Top Level Objectives.

The approach used here is holistic simply because of the following:

1. Just because we are dealing with software, we are not jumping on declaring defects per thousand lines of code as our metric for example.
2. There is a clear link from top to the bottom of the organizational hierarchy. All metrics are therefore related and connect to the Top Level Objectives.
3. Metrics are defined at every stage of the process abstraction.
4. All metrics are created by people, divisions, group, or functions based on their role in supporting the Top Level Objectives.
5. All the areas, e.g. Software Development Group, Software Documentation Group, and the Corporate Quality department in this example are involved in creating their own strategies and metrics to support the Top Level Objectives.
6. There is a high likelihood that the company will succeed if all areas execute together as a team in their individual areas.
7. This focuses on the business strategy as well the processes and their associated metrics.

3.1 Vision Statement - Example

As an example, let is assume that the company has the following vision:

- "To be the Dominant Supplier of the Dental Software in the USA".

3.2 Key Strategic Issues - Example

As an example, let us assume that the company is facing the following key strategic issue (this is for illustrating the example):

- Company's financial performance has been weak and market share has been declining at the rate of 10% per year.

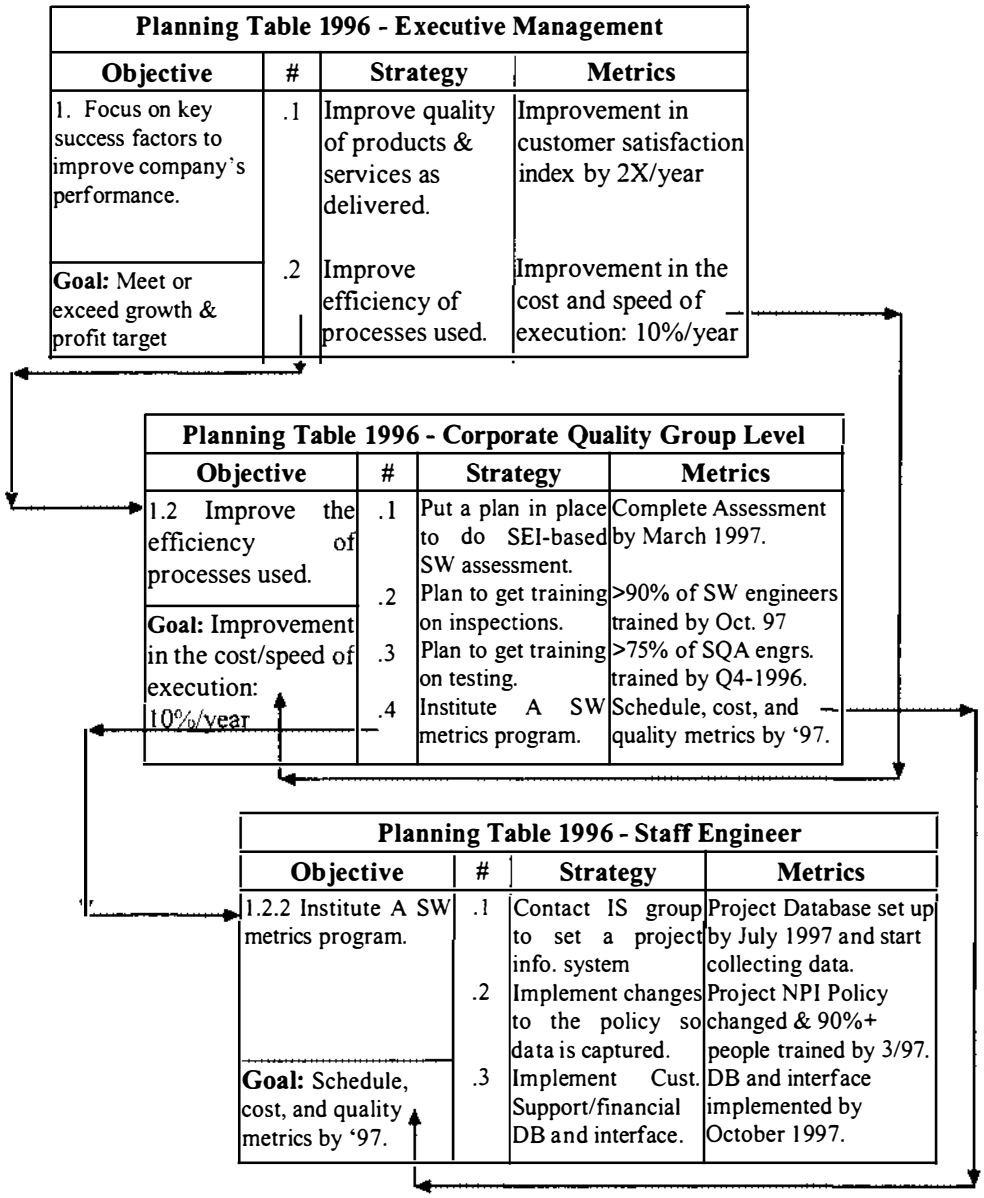
3.3 Examples of Objective/Goal and Strategy/Metrics Linkage

Following three pages describe how three different groups can derive their metrics from the highest level of the organization. For brevity, I have chosen only three examples to illustrate the point. The same process should be repeated for all

The focus of these examples is to provide a methodology by which individual organizations can derive their own metrics suitable to their organizations. The examples and the metrics chosen are for illustrating the methodology. There is no specific reason why only those metrics are chosen.

Part of the same top level information is repeated on the next three pages. This is intentional for the following reasons. The top level objectives, goals, strategies, and metrics should be visible to everyone in the company. The examples also show each individual's metrics sheet as it relates to what the company is trying to do. Secondly, seeing the linkages from the bottom to the top and to the peer groups is very important. Finally, the figures emphasize the key point that everything is connected.

3.3.1 Corporate Quality Group's Derivation



Planning Table 1996 - Executive Management

Objective	#	Strategy	Metrics
1. Focus on key success factors to improve company's performance.	.1	Improve quality of products & services as delivered.	Improvement in customer satisfaction index by 2X/year
Goal: Meet or exceed growth & profit target	.2	Improve efficiency of processes used.	Improvement in the cost and speed of execution: 10%/year

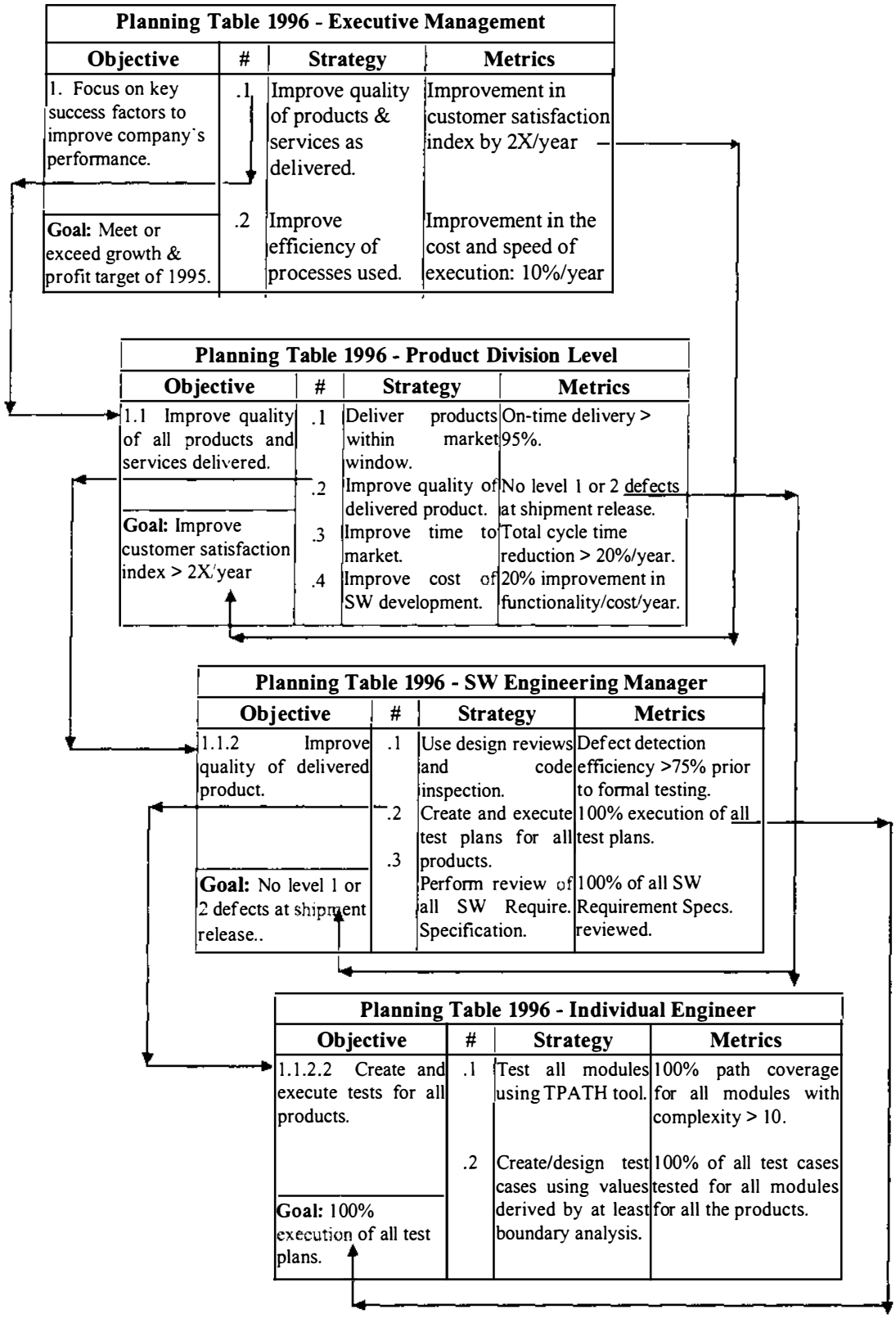
Planning Table 1996 - Corporate Quality Group Level

Objective	#	Strategy	Metrics
1.2 Improve the efficiency of processes used.	.1	Put a plan in place to do SEI-based SW assessment.	Complete Assessment by March 1997.
	.2	Plan to get training on inspections.	>90% of SW engineers trained by Oct. 97
Goal: Improvement in the cost/speed of execution: 10%/year	.3	Plan to get training on testing.	>75% of SQA engrs. trained by Q4-1996.
	.4	Institute A SW metrics program.	Schedule, cost, and quality metrics by '97.

Planning Table 1996 - Staff Engineer

Objective	#	Strategy	Metrics
1.2.2 Institute A SW metrics program.	.1	Contact IS group to set a project info. system	Project Database set up by July 1997 and start collecting data.
	.2	Implement changes to the policy so data is captured.	Project NPI Policy changed & 90%+ people trained by 3/97.
Goal: Schedule, cost, and quality metrics by '97.	.3	Implement Cust. Support/financial DB and interface.	DB and interface implemented by October 1997.

3.3.2 Software Development Group's Derivation



Planning Table 1996 - Executive Management

Objective	#	Strategy	Metrics
1. Focus on key success factors to improve company's performance.	.1	Improve quality of products & services as delivered.	Improvement in customer satisfaction index by 2X/year
Goal: Meet or exceed growth & profit target of 1995.	.2	Improve efficiency of processes used.	Improvement in the cost and speed of execution: 10%/year

Planning Table 1996 - Product Division Level

Objective	#	Strategy	Metrics
1.1 Improve quality of all products and services delivered.	.1	Deliver products within market window.	On-time delivery > 95%.
	.2	Improve quality of delivered product.	No level 1 or 2 defects at shipment release.
Goal: Improve customer satisfaction index > 2X/year	.3	Improve time to market.	Total cycle time reduction > 20%/year.
	.4	Improve cost of SW development.	20% improvement in functionality/cost/year.

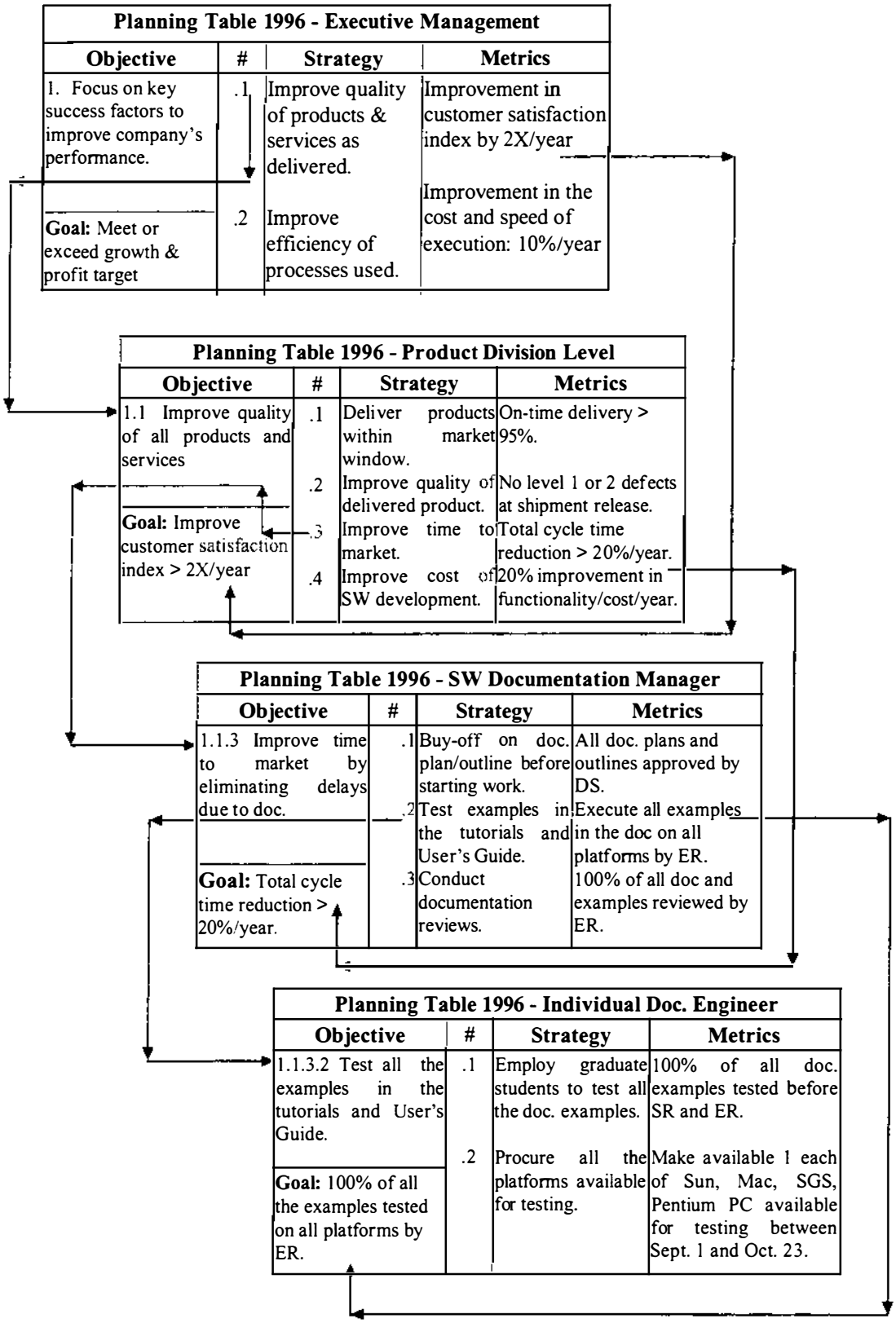
Planning Table 1996 - SW Engineering Manager

Objective	#	Strategy	Metrics
1.1.2 Improve quality of delivered product.	.1	Use design reviews and code inspection.	Defect detection efficiency >75% prior to formal testing.
	.2	Create and execute test plans for all products.	100% execution of all test plans.
Goal: No level 1 or 2 defects at shipment release..	.3	Perform review of all SW Require. Specification.	100% of all SW Requirement Specs. reviewed.

Planning Table 1996 - Individual Engineer

Objective	#	Strategy	Metrics
1.1.2.2 Create and execute tests for all products.	.1	Test all modules using TPATH tool.	100% path coverage for all modules with complexity > 10.
	.2	Create/design test cases using values derived by at least boundary analysis.	100% of all test cases tested for all modules for all the products.
Goal: 100% execution of all test plans.			

3.3.3 Software Documentation Group's Derivation



Planning Table 1996 - Executive Management

Objective	#	Strategy	Metrics
1. Focus on key success factors to improve company's performance.	.1	Improve quality of products & services as delivered.	Improvement in customer satisfaction index by 2X/year
Goal: Meet or exceed growth & profit target	.2	Improve efficiency of processes used.	Improvement in the cost and speed of execution: 10%/year

Planning Table 1996 - Product Division Level

Objective	#	Strategy	Metrics
1.1 Improve quality of all products and services	.1	Deliver products within market window.	On-time delivery > 95%.
	.2	Improve quality of delivered product.	No level 1 or 2 defects at shipment release.
Goal: Improve customer satisfaction index > 2X/year	.3	Improve time to market.	Total cycle time reduction > 20%/year.
	.4	Improve cost of SW development.	20% improvement in functionality/cost/year.

Planning Table 1996 - SW Documentation Manager

Objective	#	Strategy	Metrics
1.1.3 Improve time to market by eliminating delays due to doc.	.1	Buy-off on doc. plan/outline before starting work.	All doc. plans and outlines approved by DS.
	.2	Test examples in the tutorials and User's Guide.	Execute all examples in the doc on all platforms by ER.
Goal: Total cycle time reduction > 20%/year.	.3	Conduct documentation reviews.	100% of all doc and examples reviewed by ER.

Planning Table 1996 - Individual Doc. Engineer

Objective	#	Strategy	Metrics
1.1.3.2 Test all the examples in the tutorials and User's Guide.	.1	Employ graduate students to test all the doc. examples.	100% of all doc. examples tested before SR and ER.
	.2	Procure all the platforms available for testing.	Make available 1 each of Sun, Mac, SGS, Pentium PC available for testing between Sept. 1 and Oct. 23.
Goal: 100% of all the examples tested on all platforms by ER.			

3.3.4 Review Table

The metrics should be reviewed on a regular basis (quarterly, annually or as often as needed) with the help a simple table as shown below:

Objectives and Strategies	Actual Performance	Summary of Analysis of Deviation	Trend Status	Correction Actions
1.	Goal Actual			
.1	Metrics Actual			
.2	Metrics Actual			

This simple table can be used look at both in-process and post-activity metrics, analyze and track corrective actions.

4. Issues with this Method

The method presented here of taking a holistic, system, or business view is very powerful, yet it has several drawbacks. Those drawbacks could be overcome though.

1. This method involves lot of hard work which the teams may not be interested in doing or may not have time to work on. For example, crystallizing the vision, 2-3 key strategic issues, and strategies is not an easy task. It requires lot of team work from various cross-functional groups. Defining measurable attributes: the metrics can also be very time consuming process.
2. This is a process with long term impact which means there are no easy solutions. The approach being holistic is more likely to win over the conventional one, yet it involves lot of had work to identify issues and come up with solutions. It also requires careful work in terms of choosing metrics.
3. The derivation trees can grow arbitrarily large or lengthy and can become difficult to manage. A tool support would help, but there are no tools that can support this methodology that I know of.
4. Reviews on a regular basis (quarterly, yearly, or as often as needed) are a must. Without them and ongoing corrections to the plans, this benefits of having metrics cannot be achieved. Such a program is likely to get off the upper management's list very fast if the benefits are not shown quickly.
5. The method assumes that people involved in this process have extensive knowledge of the business, market place, and the domain. The metrics must be carefully chosen so they can truly represent the Top Level Objectives. It also assumes that people involved in the metrics definition process understand the intuitive connection between various organization levels and functions. In that context, this is not a generic metrics development method. This requires an extensive knowledge of the interplay of various business component.

5. Summary

Metrics definition is driven in a top-bottom fashion and addresses the strategic issues facing the company in terms of customers and business. This type of metrics definition is easy to sell to the upper management as they can see a clear connection between what the company is trying to do and what is being measured at all levels of the organization layers. The resources required to carry out measurements are easy to justify as they are directly related to the business process. It prevents the process of metrics collection from becoming laborious as only metrics that are necessary to address the Top Level Objectives are collected. The metrics are not collected just for code but for product as a whole. This means other areas such as support, documentation, training etc. that are necessary to make business successful and win in the market place are also addressed. This allows people to focus on things that are

important for success. The metrics in this scheme are an integral part of the life cycle process and thus form a cohesive structure. The localization of metrics at the process stage level makes individuals responsible for their own action. The method of defining post-activity metrics forces you to talk to your customers so you can understand what their needs are.

The problem with this method is that the planning tables can get very lengthy and can become difficult to manage. If regular reviews, corrective actions and monitoring are not implemented, the benefits of instituting this method can be lost.

6. References

1. Watts Humphrey, *Managing the Software Process,* Addison Wesley Publication, 1989.
2. A. Radice et al.. A Programming Process Architecture. *IBM Systems Journal 24*. 79-101. 1985. This entire issues is an excellent source on software programming processes.
3. T. McCabe, A Complexity Measure, *IEEE Transaction on Software Engineering,* December 1976.
4. S. M. Henry, and D. G. Kafura, Software Structure Metrics Based on Information Flow. *IEEE Transaction on Software Engineering,* September 1981.

Software Quality Management by Responsibility Driven Software Evolution

Kingsum Chow
MicroComputer Research Labs
Intel Corporation, JF3-359
2111 NE 25th Ave
Hillsboro, OR 97124, USA
kingsum@ichips.intel.com

Keywords

Software quality, software maintenance, asynchronous software evolution, responsibility driven software evolution, program restructuring.

Biographical Sketch

Kingsum Chow received his Ph.D. from the Department of Computer Science and Engineering, University of Washington, Seattle, Washington. His research interests are software evolution, design and specification of software systems and object-oriented design and programming. Currently, Kingsum Chow is with Intel Corporation.

Chow received an MS in computer science and an MS in chemistry from the University of Washington, Seattle, Washington, and a BS (First Class Honors) in chemistry from the National University of Singapore, Republic of Singapore. He is a student member of ACM and IEEE.

Software Quality Management by Responsibility Driven Software Evolution

Kingsum Chow
MicroComputer Research Labs
Intel Corporation, JF3-359
2111 NE 25th Ave
Hillsboro, OR 97124, USA
kingsum@ichips.intel.com

Abstract

Software quality degrades gradually throughout the lifetime of the software due to changes that are applied to it for maintenance and evolution [Belady & Lehman 1985]. Since a typical software application uses library code as well as its own code, the quality of a software application depends not only on the application code but also the library code that the application uses. In many cases, a library maintainer takes on the responsibility of the library code and the library interface, while each application maintainer takes on the responsibility of the application code and using the library interface. Problems arise when the library interface is changed and productivity suffers when each application maintainer has to adapt their code to the new interface. We identified patterns of interface changes from several software systems. We developed a method to shift part of the responsibility of updating the application code in response to library interface changes from application maintainers to library maintainers to reduce the productivity loss. Our method is based on change specifications by library maintainers and change propagation by our tools. Our results show promises in handling common interface changes.

Keywords: Software quality, software maintenance, asynchronous software evolution, responsibility driven software evolution, program restructuring.

1 Introduction

Libraries provide leverage in large part because they are used by many applications. As Parnas [1972], Lampson [1984], and others have noted, stable interfaces to libraries isolate the application from changes in the libraries. That is, as long as there is no change in a library interface's syntax or semantics, applications can use updated libraries simply by importing and linking the new version.

This is a great idea, but not completely practical. Pancake, citing Ungar, recently summarized the practical problems of this approach:

> Ungar cautioned that "the notion of interface is much more of an illusion than people give it credit for. The problem is that nobody really knows how to formally write down the definition of an interface in a way that can be checked and will guarantee the thing will really work when used by different clients... You try to get all your design decisions to hide behind interfaces so that changes percolate through as few levels as possible. But every honest programmer will admit there are times when you have to change your mind, and the interface changes." The problem of how to manage such changes has not been solved, particularly if components are in use across a number of organizations or if the components have been modified or built upon in any significant way [Pancake 1995, p. 35].

While how to formally write down the definition of an interface is clearly a problem, the bigger problem, as Pancake pointed out, is how to manage interface changes, particularly if components are in use across a number of organizations.

Our goal is to make progress towards solving this particular problem. In addition, it also provides a practical programming model and an effective solution to at least one aspect of this problem. We provide a solution for both the component providers and component users with respect to the many tedious adaptations to interface changes in the life cycle of a reusable component after its initial release. We address an approach to reduce some of the costs of updating applications when a library, upon which the applications depend, changes in unstable ways – when the syntax and/or the semantics of the library change. Section 2 describes the problem. Section 3 describes the case studies of interface changes. Section 4

describes our approach. Section 5 describes the limitations of our approach. Section 6 concludes this paper.

2 The problem

To decrease costs and increase reliability, today's software products are often created or assembled by reusing software components. A software development team often uses some components that are provided by other teams within the same organization or from other organizations, e.g. commercial software libraries. If the semantics of a reused component are changed, other components that depend on the stable semantics of this component may be affected. Although it is not desirable, it may be necessary to change the semantics of a reused component due to unanticipated changes. In such case, the problem is to identify the numerous other software components or applications both within the same organization or outside that use this changed component and to propagate appropriate changes to those other components or applications to make them compatible with the new version of the software component. In other words, the problem is both *when* and *how* to propagate these changes to update other software components that are not even visible to the maintainer of that reused component. We call this process *asynchronous software evolution* [Chow & Notkin 1996a], due to the asynchronous nature of the times when changes are applied to a reused component and then separately to the numerous other components that use it.

The problem of asynchronous software evolution is becoming ever more important due to the huge growth in the number of vendor libraries that support an even larger growth in the number of applications. Although it is hard to estimate precisely, it is easy to believe that almost every application imports from a number of libraries. Some of these libraries may even reuse routines from other libraries.

The asynchronous software evolution problem arises when the interface that an application depends on is changed in some way that requires a change to the application code. This problem manifests itself in two dimensions:

- the length of the chain of the use relations among the libraries and
- the multiplicity of the library users, which can be applications or other libraries.

If a library interface is stable, i.e., the changes in the future releases of the library do not need the application maintainers to change their code, then we have an ideal library interface that satisfies Parnas' criteria of "designing intermodule interfaces that are insensitive to the anticipated changes" [Parnas 1979, p.313]. But in real life, the future is hardly predictable and having a stable interface that can handle future changes is hard, if not impossible.

Library maintainers may decide to change its interface for a reason. An example of such a requirement is the standardization of the syntax of some related interfaces across the same library or, more often, to make them look more like another library which has an interface becoming popular in the programming world. Other examples include providing extensions to the current services and providing semantic compatibility with new devices or other libraries.

Library maintainers may be reluctant to change the interface of a library even with valid reasons for doing so. They fear that users of the current version of the library may reject the new version because of the additional requirement for them to change their user code. Some of the library users may not be able to change their code properly. Others may forget to change something that needs to be changed. The rest may find it too annoying, distracting or time consuming to deal with something they simply hate to do.

Library maintainers may be less reluctant to change the interface if they can somehow help the library users adapt their code to the new interface. When changing an interface, a library maintainer probably has a good idea of the effect it will have on the users of that interface. In fact, as we will see later, the library maintainer may know what changes are required in the users' code because they are the people who designed and implemented the interface that they should know a lot about.

When a library maintainer revises a library, there is a tension between improving the interface by changing it and leaving the interface the same as before. The library maintainer may decide against changing the interface if the cost to the application maintainers in terms of effort to upgrade outweighs the benefits of the improvement. Thus, reducing the cost to upgrade the applications would give library maintainers more leverage in improving a library interface.

In most cases, however, library maintainers do not know who their users are, much less are able to access and help them adapt their code. The library maintainers should still be able to specify the required adaptation changes in some way and let those changes propagate into the library users' code, at a different time and place from the library maintainers. Ideally, the integration of library interface changes and the application code changes in response to such interface changes should require little intervention from an application maintainer.

In a typical programming practice, a person is responsible for his or her own code. A library maintainer is only responsible for the library's code while application maintainers are responsible for the application code, including adapting the code to use a new library release. Nevertheless, when a library interface is changed, the library users need to be notified of those changes. The changes and suggested way to adapt code to those changes are usually written in a human language as part of a printed document, the README file associated with the new library or e-mail announcements. While there are also other methods of library change notification, a widely used method is to include a README file or a similar kind of document in a software distribution.

There are many weaknesses of the README file approach. The first problem with the use of README files is that it is not clear whether this kind of information can actually reach clients, e.g., the intended audience may not be aware of the existence of the README file. Furthermore, it is not clear that clients will be sufficiently motivated to read the README file. Even if they do read the README file, the updating activity may still be difficult to understand and to carry out. The README file may not provide complete or even sufficient information for all maintenance issues. There is no standard method or programming tool to verify that the README file is consistent and up to date.

The distribution of responsibility in the programming paradigm is not optimal either. When changing the library interface, the maintainer has information about potential effects of each change that is important for the application maintainer to know. The library maintainer may attempt to write the pertinent information down informally at the end of the project, prior to release. But this is far from reliable. The library maintainer should be required to think more carefully about the interface change since it may affect many applications. A formal or at least semi-formal mechanism should be available for the library maintainer to pass important change information to the application maintainers.

We identify a spectrum of programmer responsibility models with respect to handling interface changes in software evolution (see Figure 1). On one extreme is the absence of interface change information from library maintainers to application maintainers and no automation of the update of each application source in response to the interface changes. On the other extreme is the presence of interface change information from library maintainers to application maintainers and automation of the update of each application source in response to the interface changes. We regard the use of informal information, such as the content of a README, closer to the first extreme. We develop an approach that allows better collaboration between library maintainers and application maintainers. Thus, the approach helps move towards the other extreme.

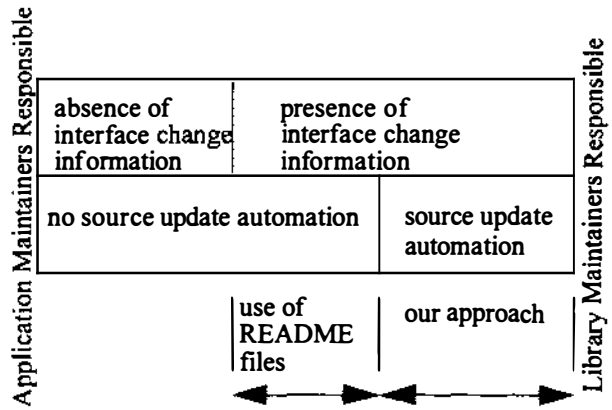


Figure 1. A spectrum of responsibility models.

3 Study of interface changes

Our approach to supporting interface changes is based on 7 case studies (Sections 3.1 - 3.7) of interface changes and the compilation of the results into a common pattern list (Section 3.8). In these studies, we identified a number of common changes. So, although the various kinds of interface changes are theoretically unbounded, there are actually common patterns of interface changes that show up again and again. The objective of our approach is to provide mechanisms so that the majority of the change patterns can be handled efficiently, both in terms of their specification and propagation. This chapter describes the common patterns of interface changes and the case study examples from which they are derived. We also discuss briefly the effects of those interface changes on application source. The case study examples were chosen from systems where there was interface change information as well as source code. The range of systems that are covered represents systems in various stages of development. Although the examples (Table 1) are taken from C and C++ programming languages, they can be generalized to other imperative programming languages that explicitly represent interface information. We do not imply that the patterns of the interface changes will be the same for other languages however. Language features certainly play an important role in what interface changes are feasible. But, as we will see shortly, most of the changes we look at are possible with other imperative languages, too.

We studied a wide variety of interface changes here. Our selection criteria are the availability of interface

Table 1 The list of case studies

System	Description
Borland C++	Borland C++ system and class library
libg++	The GNU C++ class library
nihcl	National Institute of Health Class Library
rdist	Remote file distribution program
gzip	Compress or expand files
make	Maintain, update, and regenerate groups of programs
diff	Differential file comparator

changes and the availability of source code. The interface changes in our studies are available in change log files (most of which are taken from a set of GNU programs), README files and release documents. Sometimes, we determined the interface changes by getting two versions of a program and finding the difference manually. One should note that some of the case studies (e.g. rdist, gzip, make and diff) are not taken from stand alone libraries but rather stand alone applications. However, the interface changes discovered from them still give invaluable insights to the problem of interface changes.

3.1 Case study: Borland C++

We first studied the interface changes in Borland C++ version 4.0. The interface change information was taken mostly from the README file [Borland 1993a] and chapter 1 of the library reference [Borland 1993c]. Some of the changes and how to handle them can also be found in the programmer's guide [Borland 1993b].

We hoped to get interface changes from the C++ library only but we got more than we hoped for. An example of a language change in their "Make" program was also observed. This clearly shows that changes can come in many unexpected places!

We focused on the changes in common language constructs in our studies. Thus, we focused on changes that are not tied to special features of C++, e.g., class inheritance.

The first change we noticed from this study is not an interface change in the C++ language or the library but a change in the Borland's MAKE language [Borland 1993a]. Unlike the pervious versions (i.e., version 3.1 or earlier) of MAKE, the new MAKE will not generate some space between two directly adjacent macros, i.e., two macros without any space in between them.

```
SRC1 = main.cpp
SRC2 = support.cpp

ALLSRC = $(SRC1)$(SRC2)
...
```

Figure 2. Illustration of the change in Borland's MAKE - Makefile

An example Makefile is shown in Figure 2. When ALLSRC is expanded it would become

```
main.cppsupport.cpp
```

instead of

```
main.cpp support.cpp
```

This change in MAKE can be considered as one kind of a library interface change because it affects the application's MAKE programs directly in a way that it requires corresponding responses in the application's MAKE programs to adapt to this change.

Sometimes, a certain protocol of function calls is required even though other functions are available. For example, when creating secondary threads in version 4.0, the _beginthread() and _endthread() functions must be used to ensure the proper initialization and cleanup. Although it has not been fully elaborated in the documentation, it appears that the function calls CreateThread() and ExitThread() from version 3.1 will need to be replaced by those two functions respectively.

Some functions were added to the new version. For example, there were two new functions that provide access to 32-bit operating system file handles:

```
_open_osfhandle()
_get_osfhandle()
```

Some functions were removed from the old version. For example, the following function was removed:

```
Type_info::fname()
```

Some functions were renamed to comply with ANSI naming requirements [Borland 1993c, Chapter 1]. For example, 7 global functions were renamed in 2 header files (see Table 2).

Some global variables were renamed to comply with ANSI naming requirements [Borland 1993c, Chapter 1]. For example, 7 global variables were renamed in 5 header files (see Table 3).

Some function parameters are removed in this release. For example, the following member functions for direct containers (except dictionaries) no longer take a delete parameter.

Table 2 Renamed functions in Borland C++

Old name	New name	Header file
_chmod	_rtl_chmod	io.h
_close	_rtl_close	io.h
_creat	_rtl_creat	io.h
_heapwalk	_rtl_heapwalk	malloc.h
_open	_rtl_open	io.h
_read	_rtl_read	io.h
_write	_rtl_write	io.h

Table 3 Renamed global variables in Borland C++

Old name	New name	Header file
daylight	_daylight	time.h
directvideo	_directvideo	conio.h
environ	_environ	stdlib.h
sys_errlist	_sys_errlist	errno.h
sys_nerr	_sys_nerr	errno.h
timezone	_timezone	time.h
tzname	_tzname	time.h

```
Flush
Delete
```

Thus, old function calls of these functions using an argument will not compile but syntax errors will be noted by the compiler.

Some functions were renamed but, in some cases, the reasons were not given. For example, bag and set container member function FindMember was renamed to Find but took the same parameters (see Table 4).

Table 4 Renamed functions in Borland C++ class library

Old name	New name	Class
FindMember	Find	bag container
FindMember	Find	set container

Some functions were added. For example, containers now have a member function called DeleteElements with this prototype:

```
void DeleteElements()
```

Also, functions might be added to make many classes to look consistent in their interfaces. For example, all list and double-list containers were given the DetachAtHead member function.

Sometimes, the function prototype changes were explained. For example, the Detach and Flush member functions were changed to make them similar to other classes. Sometimes, the reasons for the changes are obvious, for example, the assign member function of the string class was changed to include one more default argument.

Table 5 Functions added to Borland C++ class library

New function	Classes
DeleteElements	containers
DetachAtHead	list and double-list containers

The old prototype for this function was:

```
assign( const string&, size_t = NPOS );
```

It was changed to:

```
assign( const string&, size_t = 0, size_t =
NPOS );
```

The size_t parameter in the old version was the number of characters to copy. In the new version that is the second size_t parameter; the first one is the position in the passed string to start copying.

The same change was made in 11 other functions (see Table 6).

Table 6 Function prototype changes in Borland C++ class library

Function
string(const string &, size_t, size_t);
string(const char *, size_t, size_t);
string(const char *, size_t, size_t);
assign(const string &, size_t, size_t);
append(const string &, size_t, size_t);
append(const char *, size_t, size_t);
prepend(const string &, size_t, size_t);
prepend(const char *, size_t, size_t);
compare(const string &, size_t, size_t);
insert(size_t, const string &, size_t, size_t);
replace(size_t, size_t, const string &, size_t, size_t);

There is a rather complicated change when a particular group of macros was used. The old use required the use of both "declare" and "define" macros in the same file, for example:

```
DIAG_DECLARE_GROUP( Sample );
DIAG_DEFINE_GROUP( Sample, 1, 0 );
```

There was also a "create" macro that replaced the two macros above, for example:

```
DIAG_CREATE_GROUP( Sample, 1, 0 );
```

The new version still had both the "declare" and "define" macros. But the "define" macro now included the function of the "declare" macro too, i.e., making it similar to the "create" macro. Furthermore, the "declare" macro could not be used together with a "define" macro. This has two consequences:

- The "create" macro is removed.
- The "declare" macro is valid in files where the "define" macro is not used.

Thus, code that uses both the "declare" and "define" macros needs to remove the "declare" macro. Code that uses the "create" macro should be changed to use the "define" macro.

The meanings of these "find" functions (see Table Table 7) were all revised in a similar manner. The interpretation of the return value of this set of functions was changed. In the new version, a successful find returns the character location and a failed find returns NPOS. Since it is the interpretation of the integer values that changes, we classify this as a semantic change. The interpretation of the old values was not clear but most people would think it should return 0 for a failed search.

However, even with this newly revised set of find functions, there was some inconsistency. For example, from the set of "find_first_of", "find_first_not_of", "find_last_of" and "find_last_not_of" functions (two versions of each, one taking one argument and the other taking two arguments), all functions return NPOS when they fail except the "find_last_of" function that takes one argument. Most of us would strongly suspect that there was a misprint for that particular "find_last_of" function and indeed it should probably fail with a return value of NPOS. Here, we noticed not only the problem of semantic changes, but also the difficulty in checking the correctness or consistency of README files by the library maintainers, or the comments in the code, for that matter.

In Table 7, the function (marked "/*!*/") actually did return NPOS for a failed search in our experiment. Thus, the README document is incorrect.

3.2 Case study: libg++

We studied the GNU C++ library (commonly known as libg++) [Lea 1988]. We extracted the changes from the change log file for libg++ version 2.6.2. There had been a lot of changes for the past revisions of this library. While we did not attempt to verify that the change log is complete, we believe nevertheless it contains a good sample of the interface changes. We observed the following interface changes in the GNU C++ library. Most of the changes are

Table 7 Functions with semantic changes in Borland C++ class library

Function
`size_t find_first_of( const string _FAR &s ) const`
`size_t find_first_of( const string _FAR &s, size_t pos ) const`
`size_t find_first_not_of( const string _FAR &s) const`
`size_t find_first_not_of( const string _FAR &s, size_t pos ) const`
`/*!*/ size_t find_last_of( const string _FAR &s ) const`
`size_t find_last_of( const string _FAR &s, size_t pos ) const`
`size_t find_last_not_of( const string _FAR &s ) const`
`size_t find_last_not_of( const string _FAR &s, size_t pos ) const`

annotated with descriptions to provide more concrete examples instead of the generalization.

Some function parameter types were changed.

Some default parameters were changed to non-default. An example is given in Table 8. In this example, the fourth parameter, sk, had a default value but was changed to have no default value. The reason that was given is to prevent ambiguous matching.

Table 8 Functions with default parameters changed to non-default in libg++

Old function	New function	Header file
`istream(int filedesc, char* buf, int buflen, int sk = SOME_CONST, ostream* t = 0)`	`istream(int filedesc, char* buf, int buflen, int sk, ostream* t = 0)`	`istream.h`

Some functions were added. Some of these newly added functions were overloaded versions of an existing one (Table 9).

Some member functions were made public. In fact, making functions public is like adding functions, from the view point of a user using the function. A few examples are shown in Table 10.

Some function return types were changed. Two examples are shown in Table 11.

Table 9 Functions added to libg++

New function	Header file
`ostream << (const void * p)`	`ostream.h`
`re_comp()`	`std.h`
`re_exec()`	`std.h`
`more (overloaded) versions of abs()`	`builtin.h`
`more (overloaded) versions of even() and odd()`	`builtin.h`
`check_state() in File class`	`stream.h`
`rewind()`	`std.h`
`bsearch()`	`std.h`
`char* chr(ch)`	`builtin.h`
`virtual destructors`	`many classes`

Table 10 Functions made public in libg++

Function	Header file
`opterr (was private)`	`Getopt.h`
`(unspecified)`	`Poisson.h`
`(unspecified)`	`Lognormal.h`

Table 11 Functions with changed return types in libg++

Function	New return type	Header file
`puts`	`int`	`stdio.h`
`qsort`	`void`	`std.h`

The meaning of some function return values were changed. Table 12 shows two such examples.

Table 12 Functions with changed return values in libg++

Function	Header file	New meaning
`low()`	`MPlex.hP`	`returns lowest valid index`
`contains(Regex)`	`String.h`	`unspecified`

Some data types were renamed. One example is in math.h, where libm_exception is renamed to struct exception.

All header file names were shortened to make the system run on Unix System V. Some header files were added (e.g. new.h). A header file was split into two. For example, Regex.h was originally in String.h. Some type declarations were moved from one file to another. For example, bool enum was moved to bool.h from some unspecified source.

3.3 Case study: NIH class library

The National Institute of Health Class Library [Gorlen et al 1990] is available from the public domain. It was one of the first freely available class libraries and had gone through a few revisions. Unfortunately, due to the absence of a change log file, it is harder to determine interface changes in this library. We obtained versions 3.0 and 2.2 of this library. We inferred the interface changes from the README file and source code, when ambiguity arose.

The library name was changed from OOPS to its now popular name, NIHCL.

A number of global variables were moved to become static members of a new class.

Some macros were removed. For example, the macros `READ_OBJECT_AS_BINARY` and `STORE_OBJECT_AS_BINARY` were removed.

Some functions were added. An example is: dumpOn().

Several types were renamed. They are summarized in Table 13.

Table 13 Renamed types in NIHCL

Old name	New name
`Arrayobid`	`ArrayOb`
`Linkobid`	`LinkOb`

Some types were changed. For example, the type of bool was changed from char to int.

An extra parameter was added to one function. The extra parameter contains a default value. This function was:

```
String(char c, unsigned l=1);
```

It was later changed to:

```
String(char c, unsigned l=1, unsigned
extra=DEFAULT_STRING_EXTRA);
```

A function was added. For example,

```
void Process::select(FDSet& rdmask, FDSet&
wrmask, FDSet& exmask);
```

3.4 Case study: rdist

Here we present our case study of the rdist [Cooper 1994] program. The program we studied here is not really a library distribution. However, the change log file provided in the distribution contained interface changes. We observed the following interface changes from the change log file of the rdist program (version 6.1.0).

A variable, _PATH_OLDRDIST, was removed.

Some header files were renamed. Table 14 illustrates two such examples.

Table 14 Renaming header files in rdist

Old header name	New header name
configdata.h	config-data.h
patchlevel.h	version.h

A header file, config-os.h, was added.

A variable, VERSION, was moved from a file, config.h, to another, version.h.

3.5 Case study: gzip

We studied the interface changes in the gzip ("ftp.prep.ai.mit.edu:/pub/gnu") program also because of the availability of a good change log file. The following interface change was observed in the change log file of the gzip program (version 1.2.2).

Some global variables were added to a header file. For example, OF was added to lzw.h.

3.6 Case study: make

Interface changes were extracted from the change log file of the make program (version 3.70). Again, annotations with typical examples are provided to clarify the kinds of changes.

Some data types were changed by changing member types. For example, the type of member pid of struct child was changed from int to pid_t in job.h.

Some data types were changed by removing members. For example, a member, subdir, was removed from struct rule in rule.h.

Some data types were changed by changing member names. For example, a member of struct commands, lines_recurse, was renamed to lines_flags in commands.h.

Some data types were changed by additional members. Table 15 illustrates two changes of one such changed data type.

Table 15 Data types with new members in make

Data Type	Member	Header file
struct commands	command_lines	commands.h
struct commands	lines_recurse	commands.h

Some global variables were added. Table 16 illustrates two such examples.

Table 16 Global variables added in rdist

New global variable	Header file
starting_directory	make.h
unblock_sigs	job.h

Some global variables were removed. For example, files_remade was renamed to commands_started in make.h.

3.7 Case study: diff

We were fortunate to obtain two versions of the diff program and two change log files. The following interface changes were observed in the change log files of the diff program (version 2.7 and version 2.2).

Some functions were renamed. Table 17 illustrates a few of these examples.

Table 17 Functions renamed in diff

Old function name	New function name	Header file
bcmp	memcmp	system.h
bcpy	memcpy	system.h
index	strchr	system.h
rindex	strrchr	system.h

Some function parameters were changed from const to non-const. For example, the first argument of format_ifdef was no longer a const pointer.

Many functions and macros were added. Table 18 shows a list of those.

Table 18 Functions and macros introduced in diff

New function or macro	Header file
filename_cmp	system.h
filename_lastdirchar	system.h
initialize_main	system.h
same_file	system.h
index	diff.h
rindex	diff.h
line_cmp	diff.h
version_string	diff.h
change_letter	diff.h
print_number_range	diff.h
find_change	diff.h

Some functions and macros were removed. For example, S_IXGRP, S_IXOTH and S_IXUSR were removed from system.h.

Some header files were removed. For example, limits.h was removed.

Some data types were renamed. For example, struct direct was renamed to struct dirent in system.h.

Some data types were added. For example, enum line_class was added to diff.h.

Some global variables were renamed. For example, line_prefix was renamed to line_format in diff.h.

Some variables were added. Table 19 illustrates a list of the examples.

Table 19 New variables introduced in diff

New variable	Header file
ignore_some_changes	diff.h
horizon_lines	diff.h
default_line_format	diff.h
common_format	diff.h
line_prefix	diff.h
ifndef_format	diff.h
ifdef_format	diff.h
ifnelse_format	diff.h
unidirectional_new_file_flag	diff.h
file_label	diff.h
group_format	diff.h

Some global variables were removed. Table 20 illustrates some examples.

Table 20 Global variables removed in diff

Removed variable	Header file
PR_FILE_NAME	diff.h
Is_space	diff.h
textchar	diff.h

Some const global variables were changed to non-const. This change can be considered as a minor type change. This change probably has no effects on clients, however. Table 21 illustrates some examples.

Table 21 Global variables with minor type changes in diff

Variable	Header file
group_format	diff.h
line_format	diff.h

Some global variables were replaced.

3.8 Common patterns of library interface changes

Table 22 contains a union of all observed interface changes from the case studies. Each "Yes" entry corresponds to a kind of interface change for a software system that was studied. Our general classification of interface changes is shown in the first column. The classification contains the following major categories: changes in function prototypes, changes in functions, changes in function semantics, changes in data types, changes in global variables and changes in header files.

Our classification of interface changes enables us to study interface changes in some systematic manner. Thus, we introduce possible interface changes that were not observed in our case studies. So, some changes do not have associated check marks across the table.

Our classification of interface changes is only based on our empirical study of interface changes. It may not be complete because the case studies may not be complete. Furthermore, the study of interface changes was based on documents which themselves are not necessarily accurate (see Table 7 and the surrounding discussion).

The quality of individual software systems should not be inferred from Table 22. Some people may conclude that a software system that has fewer interface changes is better software. But Table 22 does not serve that purpose. The number of interface changes may be related to the length of history or revisions of a system. Also, a software system that discloses interface changes in an explicit manner may also have more interface changes. Further, the size of the system is clearly a factor.

The interface changes are classified based not only on their syntactic constructs but also their semantics. The consequences of these changes on the application maintainer are discussed below. In general, every application maintainer needs to spend a lot of effort to upgrade their source code for these changes. However, most of the upgrade process can be automated to reduce the effort required by each application maintainer.

3.8.1 Changing function prototypes

The most common interface changes probably fall within the category of function prototype changes. Some researchers interested in program restructuring [Griswold & Notkin 1993] have also identified some need for function prototype changes. In the rest of this section, we present the kinds of changes within this category.

Table 22 Patterns of interface changes

Change	Borland	libg++	nihcl	rdist	gzip	make	diff
Function prototypes							
Renaming functions	Yes						Yes
Adding parameters	Yes		Yes				
Removing parameters	Yes						
Reordering parameters	Yes						
Changing defaults to non default		Yes					
Changing non defaults to default							
Function semantics							
Changing parameter types		Yes	Yes				
Changing return types		Yes					
Changing return value meaning	Yes	Yes					
Changing parameter meaning							
Function replacements							
Adding functions	Yes	Yes	Yes			Yes	Yes
Replacing functions	Yes		Yes	Yes			Yes
Data types							
Renaming types		Yes	Yes				
Changing data type semantics							
Renaming fields in data types						Yes	
Adding fields to data types						Yes	
Replacing fields from data types						Yes	
Adding types							Yes
Replacing types				Yes			
Global variables							
Renaming global variables	Yes					Yes	Yes
Adding global variables					Yes	Yes	Yes
Replacing global variables			Yes	Yes			
Header files							
Renaming header files		Yes	Yes	Yes			
Adding header files		Yes		Yes			Yes
Replacing header files							Yes
Transfer between header files		Yes		Yes			Yes

One common function prototype change is renaming functions. Renaming a function means that only the name of the function is changed but all else remains the same. When a function is renamed, the application maintainer needs to determine if there is a function call using the old version and needs to rename that to a new name. A scan of all the function calls by the old name yields all possible sites of such calls. But some of these calls may not call the changed function. For example, if an application source does not include the header file containing the change, i.e., the application source does not use the changed function, it should not rename the function.

Another common interface change is adding parameters to a function. In some programming languages, a default value can be used for a parameter. If that is the case, the library maintainer may specify that default value and there should be no upgrade problem because no application code needs to be changed. If the language does not support default parameters, or if a default value is not used, then the application maintainer needs to search for the function calls which use the old version and change

them to use the new version by inserting an appropriate argument. The library maintainer should be able to pick an appropriate argument without no trouble.

There is actually a good reason why library maintainers are sometimes unable to use default parameters even when the language has that feature. Most programing languages (e.g. C++, Ada) only support default parameters as the right most parameters. If a parameter needs to be inserted at the front, no default values can be used.

In contrast to adding function parameters, removing a function parameter does not appear to be a common interface change. But it may happen, for example, when an argument can be ignored in the recent release. Even in this case, the application maintainer still needs to adjust the number of parameters in the function call to adapt to this new prototype.

Another function prototype change, reordering function parameters, may seem silly. But consistency among all function prototypes in a set of libraries may dictate this need. For example, the reverse order of the arguments in bcopy() and memcpy() is an obstacle for people learning the use of the C runtime library. In general, application programmers need to check for function calls that use the old interface and reorder the arguments accordingly.

One function prototype change that is similar to adding parameters is changing default parameters to non-default ones. However, it is easier because the default is known. Because of the known default value, application maintainers need to insert the default value only when that argument is not supplied. When that argument is supplied, the function call should not be changed.

The last function prototype change we studied is changing non-default argument to default ones. Since all existing calls should still work, this kind of interface change poses little problem for application maintainers.

3.8.2 Changing function semantics

Another category of interface change related to functions is changing function semantics. Function semantics are changed in two ways: changing types associated with a function (including return types and parameter types) and changing the interpretation of values (including return values and parameter values).

The first kind of function semantic change we studied is changing function return types. The return type of a function may be changed to a more general type, a more limited type, or even a different type.

When a function return type is changed and a function call to that function has been determined, the application maintainer may not need to upgrade the code if the return value is ignored by the application source. If the return value is used, the application maintainer may use a suitable type conversion function to convert the new value to the old type of the receiving variable, or change the type of the receiving variable to match the new return type. The former method requires the old return type is still available while the latter method may require multiple changes because the variable may be declared in another place and the variable may be used in other places, too.

Another function semantic change is changing parameter types. There are many possible reasons for this change. For example, the type of an argument may be changed due to other data type changes (Section 3.8.4). When a parameter type is changed, the application maintainer may need to apply a suitable type conversion function to the old function argument to convert it to conform to the new type, if implicit conversion is not available.

Function semantic changes include not only type changes but also interpretation of values even if there is no type change. The meaning of the value of a parameter may be changed for many reasons. For example, the meaning of some special integer may have a different meaning because of hardware changes. When the meaning of a function parameter is changed and a function call using that function is detected, the application maintainer needs to replace the old argument with an appropriate new expression.

Yet another function semantic change is changing the meaning of function return values. In fact, this appears to be more common in the case studies. For example, in the study of Borland C++ class library, many search functions return NPOS (i.e., -1) instead of 0 when the search fails. Perhaps the most typical problem is the interpretation of the error status of a function call, which is commonly returned as an integer value. In some systems, the return value is 0 when there are no errors. But in other systems, the return value is 1. Thus, changing the meaning of the return value is sometimes made to achieve consistency in the interpretation not only within a library but also across libraries. When the meaning of a function return value is changed and a function call using that function is detected, the application maintainer may apply a suitable value conversion function to the return value, if the return value is used.

3.8.3 Rearranging functions

A third category of interface changes is the rearrangement of functions, which include deleting, adding and moving functions.

The library maintainer may choose to delete functions for many reasons. Usually, another function will be provided to supplement the removed function. In this case, the application maintainer should locate the old function calls and replace them with appropriate calls.

A more common kind of rearranging functions is adding more functions. It directly supports new features added to a library. Unless some other functions are deleted, there is no reason to believe that the application code needs to be changed immediately in response to this kind of change.

Moving functions is, perhaps, a result of restructuring a library. For example, a group of functions may be split from one file to several files or joined from several files to one. Both can be carried out for reasons of maintaining cohesion within a file. When a function is moved from one header file to another and the usage is detected, the application maintainer should include the new header file, if it is not already included.

3.8.4 Changing data types

Another category of interface changes is changing data types. Data types are perhaps less susceptible to changes if information hiding is used. However, some language features (e.g. C struct) does not support this. In fact, more data type changes were observed in the C systems than in the C++ systems.

A common data type change is renaming data types. When a data type is renamed, the application maintainer just needs to rename old usage of the old name with new usage of the new name.

Another data type change is changing data type semantics. While this change was not observed in the case studies, it is conceivable that the interpretation of the values of data members may change in the course of time. For example, for the same data type,

```
struct T_pair { int x; int y; };
```

The old interpretation could be that (0, 0) is the center of the screen while the new interpretation could be that (0, 0) is the upper left corner.

When the data type has a new semantic meaning, the library maintainer does not have to fix the place of the declaration of the type but has to replace the usage of the members of the type in many other locations with appropriate expressions.

Another data type change is renaming data type members. When a member of a data type is changed, the application maintainer needs to replace usage of that member with a new name.

A easier data type change is perhaps adding data type members. This change usually does not require any upgrade effort by the application maintainer because this should have no effect on existing code at all.

A less easy data type change is replacing data type members. This change requires upgrade effort by the application maintainer if the removed member is used. There is perhaps no general solution to this problem. Fortunately, it is not expected that a member would be removed without adjustment by other suitable mechanisms.

Along with adding functions, adding data types provides a change that enhances the features of a system. In general, like adding functions, this change should require no extra effort from the application maintainer.

Replacing data types, like replacing functions, though undesirable, forces application maintainers to remove some old usage of obsolete data types. When a data type is removed, the application maintainer should study other new features that are presumably introduced to a library and find a suitable replacement. This manual process can be potentially much improved by using a tool that is aware of this change.

The last kind of data type changes we observe here is moving data types from one header file to another. When a data type is moved from one header file to another, the library maintainer needs to include the other header file for the application source to continue to work. This manual process can be potentially much improved by using a tool that is aware of this change.

3.8.5 Changing header files

One common header file change is renaming header files. Renaming header files is sometimes used to provide a uniform naming scheme for a library. Other times, it is used to port header files to another file system that have some file name restrictions. When a header file is renamed, the library maintainer needs to replace the old header file name with the new one. It is important to change only the #include or similar directives in some other systems that includes the file (not so straightforward because #include directives may depend on the user's prevailing search path which is defined by the user in many ways, e.g. specified in a Makefile or dependent on environmental variables).

Another change is adding header files. Adding a header file may provide additional features that can be

directly used by the application. However, since the application code did not use the header file already, there should be no effect.

Yet another change is removing header files. Removing a header file breaks all code that depends on the interface provided by that header file. The library maintainer probably provides other header files to provide similar kinds of services. The application maintainer just needs to find out what services are now available and apply them to the application code. This manual process can be improved by using a tool that recognizes the interface change.

3.8.6 Changing global variables

Another category of interface changes is changing global variables. It is commonly believed that global variables are not good interface choices. However, in some programming practices, they are still being used.

The first kind of interface change within the category of changing global variables is, again, renaming them. When a global variable is renamed, the application maintainer just needs to replace each use of the old name with a new name. The application maintainer cannot rely on a textual find and replace of the source because scoping, common in programming languages, allows a variable to hide another of the same name from an outer scope.

Another interface change is adding global variables. Adding global variables to a header file does not introduce upgrade problem to an application, however.

In contrast, another interface change, replacing global variables, introduces upgrade problem to an application. When a global variable is removed from a header file that is used by an application, the application maintainer needs to search for all usage of the removed global variable and replace each usage with a suitable one, which may not be easy to determine.

3.9 Remarks

In real life, as we observed in the case studies, interface changes do not come individually. The library maintainer introduces multiple but related interface changes in each revision. The combination of interface changes may have a different meaning from the sum of individual changes. For example, the result of removing a function and adding another function may be a replacement of a function.

The library maintainer should know the net result of a group of changes that he introduced to a library release. As a result, requiring the library maintainer to specify changes in a library release has an added advantage of resolving a group of interface changes into a few more meaningful ones.

The case studies presented in this chapter are empirical and they strongly support our taxonomy of interface changes. The patterns of changes are dependent on many factors (e.g. types of applications intended for, size of the users, the personality of the library maintainer and the culture of the organization). Nevertheless, we believe a general mechanism to handle interface changes across organizations can provide an effective solution to the task of upgrading application code in response to interface changes.

4 Our approach

The cost of a library interface change is the combined effort of introducing the interface change to a library release and upgrading all applications that may be affected due to this change. This is a generalization from the example shown earlier. While some application code may not need any changes, it still requires effort to verify that is indeed so.

We identified the cost of program source updates in response to library interface changes as the combined cost of application maintainers upgrading their code in response to a new library release. In addition to the high total cost in effort, there is the high risk of errors in current practices [Collofello & Buck 1987].

We require the library maintainer to annotate any changes that are made to indicate what updates must be made to the applications to accommodate the updated library. We then provide a set of tools that uses these annotations to update all applications that use the updated library. The idea is that the small amount of additional work done by the library maintainer can provide leverage for the many application maintainers.

Our model requires the library maintainers to specify interface changes and how existing users' code can be transformed to adapt to those changes. The specification of the library interface changes, as well as the specification for the transformations, is then extracted by a process on the library users' side. Another process on the user side constructs an internal representation of the library client program, checks for dependencies on the interface changes, and performs code transformation as stated in the transformation specification. A new client source program, which has been adapted to the new library, is produced at the end. The stages of this model of asynchronous software evolution is illustrated in Figure 3.

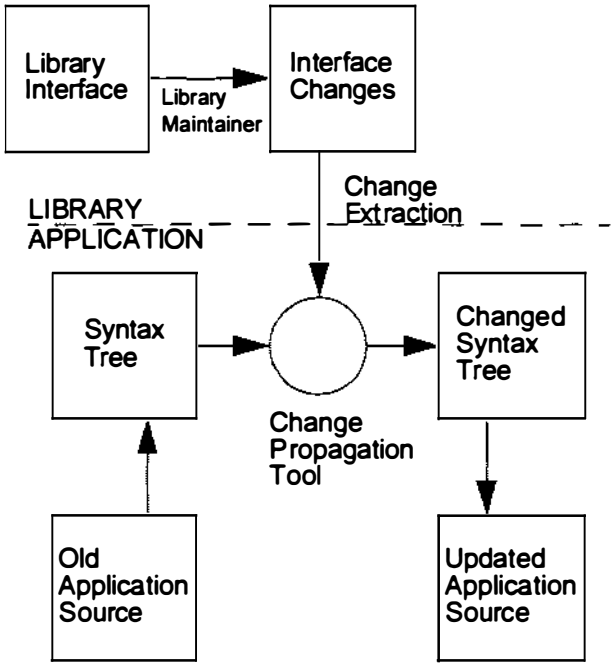


Figure 3. Overview of the change process

Figure 3 illustrates the change process. A library maintainer revises a library with interface changes. She records the changes in the interface as well as the transformation required for the users' code to adapt to the interface changes. When a client application sees these interface changes, its original source is extracted by a tool that constructs an internal representation (syntax tree) of the program. The internal representation is then modified by a change propagation tool into another internal representation, from which a new source code is generated.

The details of our design are beyond the scope of this paper (but are available in [Chow 1996, Chow & Notkin 1996b,c]). Here, we focus on whether our approach is feasible within the scope of software quality management. Thus, we focus on the validation of our ideas.

A first step in validating our ideas was developing a prototype system that supports common interface changes in open system software evolution. A second step was to test the basic concepts by applying the prototype system to examples derived from our case studies, as well as to some of our own examples. Overall, we hope to demonstrate the proof of concept of a working approach in a new research area in software evolution.

We have not produced a production quality tool set, however. One reason is expense. Another reason is the advantages of developing successive prototypes. In the absence of a production quality tool set, it is at best difficult to validate our tools with use real library maintainers and real application maintainers.

Using real library maintainers and application maintainers is an ideal way to validate our approach to supporting interface changes in open system software evolution. One of the advantages of utilizing real users would be to test our approach in real working environments with real interface changes. Instead, however, we use real interface change examples that are derived from existing software systems to simulate real software evolution without using real users [Chow 1996]. Thus, although our validation method is not ideal, it nevertheless supports the proof of a new concept in software evolution.

A set of change examples were designed to run through our tool set with but some minor adjustments. We targeted our tools to a common programming language - C. We added two extensions – function overloading and default arguments – to our targeted language to give some flavors of other languages, e.g. Ada [USDoD 1980] and C++ [Stroutstrup 1991].

We validated our approach using samples derived from interface changes categories. To run the update process, we assumed a Makefile [Feldman 1979] existed and all source files for an application were specified by $(SRC) in that Makefile. Further, we assumed that a new target, update, is created to call our tool set to update the application source code. Such a Makefile is easy to write and a skeleton of such a Makefile is provided in Figure 4.

```
1:  LIB_INC_PATH = ../include-new
2:
3:  ASE_PATH = ASE
4:  CFLAGS = -I$(LIB_INC_PATH)
5:  ASE_BIN = /homes/gws/kingsum/thesis/ase/
    bin
6:
7:  GRAMMAR_FILE = $(ASE_PATH)/grammar.g
8:  TRANSFORM_FILE = $(ASE_PATH)/
    transformer.sor
9:
10: SRC_DIFF = dir.C sdiff.C analyze.C
    diff3.C util.C
11:
12: update_diff:
13:       cp $(GRAMMAR_FILE).std
    $(GRAMMAR_FILE)
14:       cp $(TRANSFORM_FILE).std
    $(TRANSFORM_FILE)
15:       $(ASE_BIN)/upgrade_source
    $(GRAMMAR_FILE) $(TRANSFORM_FILE) CFLAGS=
    $(CFLAGS) SRC= $(SRC_ALL)
```

Figure 4. A Makefile template for our validation suites.

In the Makefile shown in Figure 4, a new include path for the new library release is defined in line 1 (LIB_INC_PATH). A local directory for working with the upgrade is defined in line 3 (ASE). The macro, CFLAGS (line 4), should be used for both the upgrade and the actual compilation of the programs. The tool directory is also defined in line 5 (ASE_BIN). The grammar specification file and the tree transformation specification file are defined in lines 7 and 8. We also need to know all the source files, which are conveniently defined in line 10 (SRC_DIFF). The update target (line 12) is processed by copying a standard grammar file and a standard tree transformation file (lines 13 and 14) and finally, our tool, upgrade_source, process the rest (line 15).

4.1 Validation using changes from diff-2.2

We validated our approach using a case study for the classification of interface changes. We picked the diff case study [Chow 1996] in version 2.2 because it had interface changes which showed up in multiple header files and affected multiple source files. Furthermore, a header file often included another header file in this study. Thus, a chain of include directives across multiple header files needs to be resolved to obtain the correct change specifications. Those features from this case study would be a good challenge to our tools. Also, source code for two versions of this system were also available. So we could examine the interface changes to the code level to apply our tools to the interface changes.

We examined the changes that were made between versions 2.2 and 2.7. We looked at the following header files: diff.h, getopt.h, regex.h and system.h. Among these header files, we examined four interface changes in system.h (see Figure 5). One of the interface changes was to rename a data type from "direct" to "dirent" (lines 2-5). Another interface change was to replace "bcopy" with "memcpy" and reordering the first two arguments (lines 7-12). The other two interface changes were to rename functions, from "index" to "strchr" (lines 21-26) and from "rindex" to "strrchr" (lines 28-33). We also examined one interface change in diff.h (see Figure 6), which was to rename a function from "Is_space" to "ISSPACE" (lines 2-7).

We examined 5 source files: dir.C, sdiff.C, analyze.C, diff3.C and util.C, which could be viewed as application sources.

The source file, dir.C, was determined by our tool to depend on diff.h, regex.h and system.h. Thus, our tool extracted the interface changes from both header files that contained interface changes and applied the all 5 interface changes to dir.C. Only 2 uses from dir.C were found by our tool that needed to be changed: "struct direct" was replaced by "struct dirent" and "bcopy" was replaced by "memcpy" with reordering of the first 2 parameters.

The source file, sdiff.C, was determined by our tool to depend on getopt.h and system.h. Since there were no

```
1:  //
2:  // _ASE_SN_BEGIN_
3:  // SN_OLDNAME = direct
4:  // SN_NEWNAME = dirent
5:  // _ASE_SN_END_
6:
7:  // _ASE_FCALL_BEGIN_
8:  // FNAME = bcopy
9:  // ARGS = 3
10: // PATTERN = #(fc:FCALL_bcopy_3 id:ID
    lp:L_PAREN ar1:gen_expr co1:COMMA
    ar2:gen_expr co2:COMMA ar3:gen_expr
    rp:R_PAREN)
11: // ACTION = id->SetText("memcpy");
    #gen_logical_expr = (SORASTBase*) #(fc,
    id, lp, ar2, co1, ar1, co2, ar3, rp);
12: // _ASE_FCALL_END_
13:
14: // _ASE_FCALL_BEGIN_
15: // FNAME = bcmp
16: // ARGS = 3
17: // PATTERN = #(fc:FCALL_bcmp_3 id:ID
    lp:L_PAREN ar1:gen_expr co1:COMMA
    ar2:gen_expr co2:COMMA ar3:gen_expr
    rp:R_PAREN)
18: // ACTION = id->SetText("memcmp");
    #gen_logical_expr = (SORASTBase*) #(fc,
    id, lp, ar1, co1, ar2, co2, ar3, rp);
19: // _ASE_FCALL_END_
20:
21: // _ASE_FCALL_BEGIN_
22: // FNAME = index
23: // ARGS = 2
24: // PATTERN = #(fc:FCALL_index_2 id:ID
    lp:L_PAREN ar1:gen_expr co1:COMMA
    ar2:gen_expr rp:R_PAREN)
25: // ACTION = id->SetText("strstr");
    #gen_logical_expr = (SORASTBase*) #(fc,
    id, lp, ar1, co1, ar2, rp);
26: // _ASE_FCALL_END_
27:
28: // _ASE_FCALL_BEGIN_
29: // FNAME = rindex
30: // ARGS = 2
31: // PATTERN = #(fc:FCALL_rindex_2 id:ID
    lp:L_PAREN ar1:gen_expr co1:COMMA
    ar2:gen_expr rp:R_PAREN)
32: // ACTION = id->SetText("strrchr");
    #gen_logical_expr = (SORASTBase*) #(fc,
    id, lp, ar1, co1, ar2, rp);
33: // _ASE_FCALL_END_
34: //
```

Figure 5. Interface change specifications in system.h.

```
1:  //
2:  // _ASE_FCALL_BEGIN_
3:  // FNAME = Is_space
4:  // ARGS = 1
5:  // PATTERN = #(fc:FCALL_Is_space_1 id:ID
    lp:L_PAREN ar1:gen_expr rp:R_PAREN)
6:  // ACTION = id->SetText("ISSPACE");
    #gen_logical_expr = (SORASTBase*) #(fc,
    id, lp, ar1, rp);
7:  // _ASE_FCALL_END_
8:  //
```

Figure 6. Interface change specifications in diff.h.

interface changes in getopt.h, only interface changes from system.h might need to be applied to this source file.

The source file, analyze.C, was determined by our tool to depend on diff.h, regex.h and system.h. However, it did not use anything that was affected by the interface changes. Thus, although interface changes were extracted from diff.h and system.h, no interface changes could be applied to analyze.C because no interface change pattern match was found from the application source.

The source file, diff3.C, was determined by our tool to depend on getopt.h and system.h. Two incidents of dependence on interface changes were detected and applied to this file.

The source file, util.C, was determined by our tool to depend on diff.h, regex.h and system.h. Three usages that were affected by the interface changes were detected and changed by our tool.

4.2 Status

We provided a mechanism for library maintainers to pass interface change specifications to application maintainers. Our current system can handle a great number of common interface changes such as function changes, data type changes, global variable changes and header file changes. Table 23 summarizes the various classes of changes that can be handled using our design and tool set.

Function prototype changes are handled well by our system. One particular function prototype change – changing non default arguments to default – does not require our system to handle it at all because existing function calls can be used without any adjustment for that change. For the rest of the function prototype changes, it is quite straightforward to write the interface change specification.

Function semantic changes are slightly more difficult to handle. This should be regarded as a property of the nature of semantic changes, however. When an argument type is changed, a type conversion function may be provided by a library to convert an existing argument type to the required type. Similarly, when an argument value is changed, a value conversion function may be provided by a library to provide the necessary conversions. These conversion functions need to be applied to existing function calls or arguments of existing calls. Thus, the specification of interface changes is not so straightforward but still can be written in our system.

Table 23 Status of our implementation

Change	Handled?
Function prototypes	
Renaming functions	Yes
Adding parameters	Yes
Removing parameters	Yes
Reordering parameters	Yes
Changing default arguments to non default	Yes
Function semantics	
Changing parameter types	Yes
Changing return types	Yes
Changing return value meaning	Yes
Changing parameter meaning	Yes
Function replacements	
Replacing functions	Yes
Data types	
Renaming types	Yes
Changing data type semantics	No
Renaming fields in data types	Yes
Replacing fields from data types	Yes
Replacing types	Yes
Global variables	
Renaming global variables	Yes
Replacing global variables	Yes
Header files	
Renaming header files	Yes
Replacing header files	Yes
Transfer between header files	No

In addition to parts of a function, an entire function can also be replaced by another one. This is usually accompanied by adding new functions to a library. We did not find it difficult to write this specification change. In fact, such an example has already been shown earlier in our validation of one of the case studies (see Figure 5).

Like function changes, some data type changes (e.g. adding a filed to a data type or adding a new data type) do not require any change specifications at all. Changes that

require adaptation of application code are renaming data types and fields in data types or replacing them. Common data type changes can be specified based on syntactical structures and handled by our system that way.

Global variables change specifications do not appear to be difficult to write and are adequately handled by our system. Our system performs scoping analysis to determine whether a variable is global or not.

Specifying and handling header file changes is similar to global variable changes in general. However, because our system is based on extracting interface changes through include directives, removing a header file causes a fundamental problem. This problem can be easily solved by a two step process. In the first step, the header file to be removed should be distributed along with another header file to be used. In this step, the change specification specifies that the other header file should be used. In the second step of the distribution, a change specification that removes the include directive for itself will be sufficient.

Our system adequately handled many common interface changes. However, we do not claim to solve all conceptually possible interface change problems.

5 Limitations

Our approach is based on interface changes that are stored in the interface specification files. There are, however, some classes of library updates that our current implementation cannot handle.

For example, we cannot not handle changes that are dependent on simultaneously changing two non-local places in the source code. A change dependency on multiple locations of the source code is not easy to handle but fortunately, this does not happen often. When this happens, a conservative approach is to alert the application maintainers whenever one of the two functions is used and let the application maintainers decide what to do.

A specific example of this problem occurs when a change is made to an output parameter of a function. In this case, the declaration of the variable to which the output parameter will be assigned generally has to be changed. But we cannot yet handle a change which is in a different location than the affected use.

In addition, there are a number of language specific features that our implementation has not been able to handle. A common problem among these language features is aliasing. For example, the alias of a variable to another variable, or when change is based on a function pointer that may point to any function in the program, only some of which may in fact be affected by a semantic change specification. Also, the introduction of inheritance class hierarchy and virtual member functions make some C++ changes hard to handle. These are interesting issues for future work. However, even in these cases, the application maintainers are still no worse than, if not better than before.

We also assume that results of function calls are independent of the order of execution of their parameters. In other words, no undesirable side effects will hinder reordering parameters of function calls which may yield a different result. This assumption falls in line with the common programming practice that a programmer should not write the code in such a way that the result of a function call depends on the actual order of parameter evaluation. Further, common programming languages (e.g., C, Ada, C++) do not specify the order of parameter evaluations, either.

In our implementation of the system, our grammar is written for a ANSI C-like grammar that requires parameter types for function prototypes. In contrast to K&R C, this grammar provides sufficient information about function prototypes so that our approach can pick a function call that matches one of a set of overloaded functions.

There is a bigger context of interface changes. Since libraries often have multiple releases, the true dependency on a header file is not only the use of a header file from a library, but also the header file of a particular release. Our approach can easily be adapted to libraries with multiple releases. In this case, on the library side, each header file of a library release will be identified by a unique version number. It is also identified by the older version number from which it is changed. The requirement for a version number in a header file can be easily met by using some revision configuration system, e.g. RCS [Tichy 1985]. On the application side, for each header file included by a program source, we can record the version number of the header file in the beginning of the source. When a header file is included, and when the previous version number of a header file is matched with the version number that is being used by a source, a change specification can then be propagated to the source, and the version number that is currently used can be changed to the newer version number.

This simple solution works well when an application upgrades with each library release, which is, perhaps, uncommon. In order to allow an application to upgrade from some older versions of a header file, we can introduce an additional keyword to the change specification language. On the library side, a new keyword, appropriately named VERSION, can be used to associate a change specification with a particular version of a header file from

a library release. On the application side, only change specifications with versions that match what is currently being used by an application source will be propagated to it. One drawback of this approach is that library maintainers may need to write multiple change specifications for multiple previous versions. We may relax the matching criteria from matching a unique version to a set of versions. Still, deducing the effect of a chain of change specifications across multiple versions lies in the hand of the library maintainer. The need to develop a tool to combine the effects of chaining multiple change specifications for different versions of the same change into one may be justified when that is found to be common.

6 Conclusion

Our approach relieves application maintainers from having to identify whether or not the changes to the library require changes to the application. Our tool set is responsible for extracting and propagation changes, if there are any, and if the changes affect the application code, while application maintainers only need to add a few rules in the Makefile.

Our approach also relieves application maintainers from having to identify and manually update each location in the application that must change in response to library changes. When a change may be required, our tool set will determine if a match of a code fragment is found by syntactic tree pattern matching as well as semantic condition matching. When a match is found, our tool set proceeds to change the code fragment as instructed by the change specification.

Our approach reduces productivity loss due to adaptation of application source in response to library interface changes.

Possible change conflicts may occur when an application uses two libraries which are changed in some conflicting manner. For example, both libraries may introduce the same function prototype to replace some existing ones. This is a subset of the problem of library interface conflicts (including change conflicts) and should be addressed separately. If there are no library interface conflicts, there should be no change conflicts because each change is based on a pattern that uses a feature that comes from one unique header file.

Our approach is not only useful for library interface changes, it can also be used whenever the detection of usage needs to be done at the client site and suggestions or corrections will be made at that time. Thus, naturally, it can be used to adapt a program to use a competitor's library interface to make the conversion of programs easier to use a newer and, perhaps, more efficient library.

Some organizations advocate the maintenance of all interfaces. They may introduce new interfaces that are more general but they will never throw away old interfaces. However, the cost of keeping the old interface, which may be discouraging, is the size of the header files which, at a minimum, is definitely a distraction to new users. It also subjects the users to misuse of the old interface. Keeping the old interface also means keeping larger object code for the old interface and new interface together. This results in longer link time. Also, it is more likely to have link time conflicts with other libraries.

Even in the case that old interfaces are kept, and more generic interfaces are introduced, our prototype system can still be used to give warnings to the client users when the use of the old interface is detected.

Although our discussion has focused on interface changes across organizations, our approach can also be useful within the same organization. In some programming teams, even though all source code is potentially visible to a library maintainer, a library maintainer may not be able to write to it. On the other hand, even if some directories from users seem to indicate the use of a library interface, a library maintainer cannot be sure of that because the directory may be not being used at all. Thus, our approach is applicable even to a small programming team where the actual update should be applied only by the application maintainer.

6.1 Contributions

The primary contributions of this paper are:

- The identification of the problem of handling interface changes as a common software evolution problem across organizations.
- The classification of interface changes.
- A method to support shifting the responsibilities of handling interface changes from application developers to library maintainers.

6.2 Future work

Although the primary goal of our approach is to handle program interface changes, its use is not limited to that domain. We believe requirement specification and design changes may also be propagated to the program. However, more understanding in the structures of requirement specification and design may be needed.

Acknowledgments

Part of this work is derived from my thesis work at the University of Washington. Comments from the referees help shape the final version of this paper.

References

[Belady & Lehman 1985] L. A. Belady and M. M. Lehman, "Programming system dynamics or the metadynamics of systems in maintenance and growth", Res. Rep. RC3546, IBM, 1971. Page citations from reprint in M. M. Lehman, L. A. Belady, Editors, *Program Evolution: Processes of Software Change*, Ch. 5, APIC Studies in Data Processing, No. 27, Academic Press, London, 1985.

[Borland 1993a] Borland's README file for version 4.0.

[Borland 1993b] "ObjectWindows for C++ Programmer's Guide", version 2.0, Borland, (1993).

[Borland 1993c] "Borland C++ for Windows Library Reference", version 4.0, Borland, (1993).

[Chow 1995] Kingsum Chow, "Program Transformation for Asynchronous Software Maintenance", Proceedings of ICSE-17 Workshop on Program Transformation for Software Evolution, William Griswold, editor, The 17th International Conference on Software Engineering, April 24-28, 1995, Seattle, Washington, USA. The proceedings are also published as Technical Report Number CS95-418, Computer Science and Engineering, University of California, San Diego, pp. 29-34.

[Chow 1996] Kingsum Chow, "Supporting Interface Changes in Open System Software Evolution", Ph.D. dissertation, Department of Computer Science and Engineering, University of Washington, Seattle, Washington. (1996)

[Chow & Notkin 1996a] Kingsum Chow and David Notkin, "Asynchronous Software Evolution", Asia-Pacific Workshop on Software Engineering Research, March 21, 1996, Hong Kong.

[Chow & Notkin 1996b] Kingsum Chow and David Notkin, "Semi-automatic Update of Applications in Response to Library Changes", Technical Report UW-CSE 96-03-01, Department of Computer Science & Engineering, University of Washington, Seattle, Washington, USA.

[Chow & Notkin 1996c] Kingsum Chow and David Notkin, "Semi-automatic Update of Applications in Response to Library Changes", (to appear) International Conference on Software Maintenance, Monterey, California, USA, Nov 4-8, 1996.

[Collofello & Buck 1987] Collofello, J. S., and Buck, J. J, "Software quality assurance for maintenance" IEEE Computer. (Sept. 1987), 46-51.

[Cooper 1994] Michael Cooper, "Rdist Version 6.1", "ftp:// ftp.usc.edu/pub/rdist/", May 1994.

[Feldman 1979] Stuart I. Feldman, "Make - a Program for Maintaining Computer Programs", Software - Practice & Experience, 9(4) pp. 255-265 (April 1979).

[Gorlen et al 1990] Keith E. Gorlen, Sanford M. Orlow and Perry S. Plexico, "NIH Class Library Revision 3.0 Release Notes" (1990). Available from "ftp://alw.nih.gov/pub/nihcl.tar.Z".

[Lampson 1984] Butler W. Lampson, "Hints for Computer System Design", IEEE Software pp. 11-28 (January 1984).

[Lea 1988] Doug Lea, "The GNU C++ Library" (libg++), USENIX C++ Conference, 1988.

[Lehman 1980] M. M. Lehman, "On understanding laws, evolution and conservation in the large-program life cycle", J. Syst. Softw. 1, 3 (1980).

[Pancake 1995] Cherri M. Pancake, "The Promise and the Cost of Object Technology: A Five-Year Forecast", Communications of the ACM 38, 10 pp. 33-49 (October 1995).

[Parnas 1972] David L. Parnas, "On the Criteria To Be Used in Decomposing System into Modules", Communications of the ACM pp. 1053-1058 (December 1972).

[Parnas 1979] David L. Parnas, "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering. 5(2) pp. 128-138 (March 1979).

[Parr 1994] Terence J. Parr, "An Overview of Sorcerer: A Simple Tree-Parser Generator", International Conference on Compiler Construction (April 1994).

[Parr 1995] Terence J. Parr. "Language Translation Using PCCTS and C++ (A Reference Guide)".

[Parr et al 1994b] Terence John Parr, Aaron Sawdey and Gary Funck, "The Sorcerer Reference Manual", Version 1.00B13. (Nov 12, 1994)

[Stroustrup 1991] Bjarne Stroustrup, "The C++ Programming Language", 2/ed. Addison-Wesley Publishing Company. (1991).

[Tichy 1985] Walter F. Tichy, "RCS - A System for Version Control" Software - Practice & Experience", 15(7) pp. 637-654 (July 1985).

[USDoD 1980] United States Department of Defense, "Ada Programming Language" MIL-STD-1815.

Justifying Testing Resources

Karen King
15450 SW Koll Parkway
MS Des1-146
Beaverton, Or 97006-6063
503-578-4284

Keywords:
Testing Resources, Budget, ROI (Return on Investment), Testing Ratio, Risk Management, Defect Analysis, Cost estimation

Karen King is the software supplier engineer at Sequent Computer Systems in Beaverton, Oregon. She received her BS in Electrical Engineering from Rice University, and has over ten years experience in software testing and other aspects of software quality.

Justifying Testing Resources

Introduction

One of the most commonly asked questions asked among testing professionals is:"What's your ratio of testing people to engineers?". When asked what they hope to gain from this question, the standard reply is that the test person wants to justify to management that the testing department is understaffed. In reality, management is seldom convinced by metrics of other organizations' testing ratios. Many testing professionals feel frustration at a gut level that not enough resources are being allocated to the testing process, but don't know how to convert that gut feel to data which is convincing to management. This paper describes problems with comparing testing ratios, and a method for making concrete proposals for additional testing resources in managerial terms.

Why is testing ratio not a good comparison?

There are many factors which affect the optimal testing ratio. Some examples follow. Note that within the examples, there is an implicit assumption that all other factors are held constant, and that this is the single factor affecting the testing ratio:

Quality expectations of your customer

Depending on who your customers are, and how they are using your software, your customers will have different expectations and acceptance levels of the quality of your software. For example, if your customers are using your software in life critical or mission critical applications, then their tolerance of defects within the software will be very low. If the customer's tolerance for defects is lower, then more effort will be put into testing in order to achieve lower defect rates, and the testing ratio would be higher.

Testability of design

If the development team is skilled in designing easily testable software, the amount of effort required to test to a designated confidence level will be lower, and the number of testing resources required will be lower, so the testing ratio would be lower.

Capability of the software development team

If the development team is more productive, then the amount of development resources required to develop a given amount of software is lower, and the testing ratio would be higher.

Capability of the software test team

If the test team is more productive, then the amount of testing resources required to develop a given amount of tests is lower, and the testing ratio would be lower.

Division of labor between development and test

The division of tasks between testing and design (for example, who writes unit tests, who duplicates the defects, etc), will greatly affect the resources required by both test and

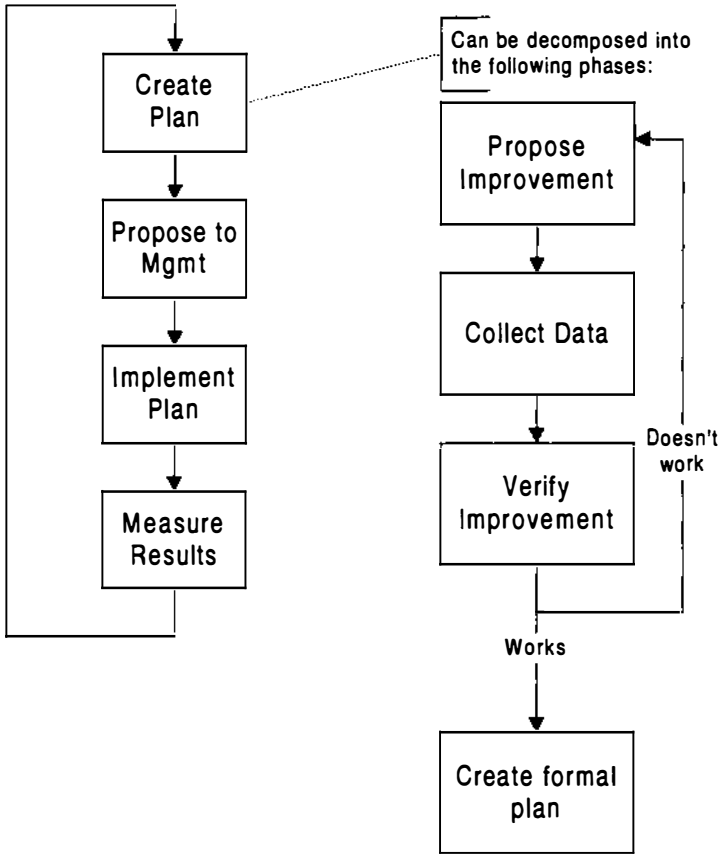
development, which will have a major effect on the testing ratio. If design is more involved in testing, then the testing ratio would be lower.

Many factors affect an organization's testing ratio. All of these factors, plus others which may be specific to your organization or the other organization, should be taken into consideration when comparing the testing ratios of different organizations.

Another Method of Justifying Testing Resources

In order to convince management to spend additional resources on anything, then you have to have a justification that the money being spent will generate an appropriate return on investment. From the perspective of testing, this means that you must convince management that increasing testing will either increase sales, or decrease product-related costs.

We'll use a standard Deming plan-do-check-act model in order to implement the testing process improvement. In this case, since we are mainly focusing on the planning and justification of the improvement, we'll also model the "plan" portion of the improvement as a separate plan-do-check act model. A graphical representation of the process being used is shown below.



Each of the steps outlined in the model is covered in detail below.

Propose Improvement

Assuming that you had the resources, figure out how you would use them. What additional testing would you do with the additional resources?

If you have data from the field, do a pareto analysis on those past defects. Think about what types of testing could have been done to catch those defects before they were released to the field. Note that this does NOT mean figuring out what specific test could have been written to catch each particular defect. Instead, this is an analysis of what class of tests could have been written to detect the problems.

If you don't have field data, work with other testers in your organization to brainstorm what types of testing they believe is missing. Work as a team to analyze the ideas, to figure out what would yield the highest returns within your organization. Beware of the "test methodology of the month" fad, where people believe that if you'd only implement the latest new methodology, then all the defects would be detected.

Before going farther, consider whether the solution(s) being proposed are the most cost effective solutions. For example, is there a simpler or cheaper test methodology that would work to prevent the same problems? Is there anything that could be done within design to prevent these problems before they reached test?

Collect Data (A)

Figure out how much this enhanced testing will cost.

Work within your group to do a work breakdown of the tasks required to upgrade your testing. Do estimates to figure out how much time each task will take. Don't forget to include factors like learning curves, and resource loading when doing calculations. Also, if your proposal involves a new process, don't forget to include time to define the process before the person begins work using the process.

Be sure and consider all costs associated with getting the new person. Include things like benefits in the calculation, the cost of equipment, new software which might be necessary, etc. Note that there are many possible ways to get additional resources, hiring new people is not the only solution. Talk to someone in your organization who does budgeting, to ensure that you're calculating costs in the same terms that your management does.

Calculate costs for any new tools which may be necessary. New tools may be ones which need to be custom-developed, or ones which can be purchased off-the-shelf.

Collect Data (B)

Calculate an estimated value of the changes. The change's value can come from two places -- value of increased sales, and cost savings which would be incurred by the organization if the defects had occurred.

This is the most difficult part of justifying additional test resources, because most organizations do not have explicit data on "value of defect prevention". Much of the data gathered here is subjective. When reporting the data, it is important to report conservative numbers, and be prepared to explain where the data came from.

Figure out which defects from your customer defect tracking system which could have been prevented by the proposed testing. Talk to the following groups to understand potential cost savings or additional sales would/could result from the reduction of defects. If people become enthused about your project, remember to harness their energy and excitement about the project when presenting to upper management.

- Marketing and sales, to see what value customers would perceive in the proposed defect reduction. Those organizations might be aware of any sales opportunities which failed because of quality concerns.
- Customer service, to understand how much they could save by catching these defects before they reach the field. Include the actual historical costs of things like:
    - time spent by customer service to explain work-arounds/fixes for these defects
    - cost of doing a quick turnaround patch for these defects
- Talk to engineering to understand how much they could save by not having to fix the defect with a special release. This also includes the cost of testing/verifying the special release.

In mission-critical or life-critical software situations, there may not be a history of defects found in the field, because a statistically significant history of major defects would put the organization out of business. Instead, another approach could be taken to justify the cost for additional testing : risk of defects.

In order to collect data on risk and the associated affects on products, the following method could be used.

- For the class of testing being considered, think about what types of errors could cause the types of defects which would be caught by this type of test.
- For each type of defect that would be caught, what is the probability that the type of error could occur? (Note, this is not a number which can be quantified -- this will be an estimate. Think about it, and be conservative with your estimate)
- Figure out the effectiveness of the current processes which would catch those types of errors. Figure the effectivenes from data from past experience. For example, if the current process for catching defects is inspections, look at data on how many inspection-preventable defects were released to test and also to the field)

- Based on the probabilities, calculate the probability of this defect reaching the testing phase. Since we assumed that these defects would not be caught by current testing, this calculated probability is the same as the probability that the defect will reach customers. By explaining the probabilities, potential consequences of these defects, and cost of additional testing, this is one method to justify spending more resources on test.

Verify Improvement

Verify that the Return on Investment on the extra testing resources is justified. If it costs more to fix the problem than is gained by diminished risk and direct cost savings, then the proposal will not be accepted.

Look at the numbers from your research. Compare your costs to your expected savings (if your savings include risk savings, then this will be difficult). If it costs more to fix the problem than is gained by diminished risk and direct cost savings, then the proposal will not be accepted. In that situation, begin again with a different testing improvement.

Do a "sanity check" by considering other methods of solving this problem. Is there an even more cost effective method? If so, then implement the alternate method, and continue thinking about other testing improvements.

Propose to Management

Once all data is collected, prepare a report for management presenting all data collected. Here are some tips on preparing to make your proposal to management

Be prepared! Have all the background information on all the data you've collected. If in the course of collecting data, you found some people who are enthusiastic about your project, be sure and include them. They might have some insight into specific management sensitivities which you may be able to capitalize on.

Prepare a collection plan, and inform management of how you will collect information on defects found in test which otherwise would have been released to the field. Your collection plan should outline your strategy for recording which defects are found by which test suite, and analyzing which defects were only found by the new test suites. These defects would have been found in the field, if this new testing hadn't been done, and the value of these defects is the most measurable component of the savings from the new testing.

Include a timeframe for your improvements. Let management know when they can expect change to start becoming measurable.

Set up a method of tracking progress on your project. If your management has periodic project reviews, it may be possible to track your improvement project along with other development projects.

If there is resistance, then try to understand where the problem occured. If there's a concern about money being requested, then investigate cutting down the proposed new testing. The analysis would have to be redone, because fewer defects will be found.

If management is not able to commit any resources, then ask them what it will take in the future. Learn from this experience, so that you can be more prepared with your next proposal.

Long term plans

No matter if you received additional resources or not, there is something you can do to improve your chances of getting funding next time. Analyze your current practices to be certain that you are using your current resources efficiently.

Long term, the goal is to prevent defects, and not to just test them out. Testing engineers should work with development engineers so that development understands what types of defects are being released to testing. Testers can help develop plans to prevent the high-impact defect causes, which will result in better quality at the time that testing receives the software.

To increase testing effectiveness, look at the testing which is done currently. Is there testing that could be cut? Do an analysis to understand which test suites are catching defects. If there are suites which appear to never catch any defects, then analyze why that might be the case. Is the things tested by that suite verified elsewhere? Is there some reason why those defects are never being created? If you find some reason why that type of test suite can never detect errors, then don't generate that type of test suite in the future.

Look at the cost of tests being developed today. For expensive tests, look at the design process to see whether there's anything that can be done at that level to prevent that class of defects. Defect prevention may not be completely effective immediately, so it may not be possible to remove the test immediately. In the long term, however, once it can be proven that the prevention is effective, then the test can be removed.

Examine training for people within test. If testers are not familiar enough with testing and programming techniques, testing may take longer than necessary. Be aware though, that this is more of a long-term solution. It will take time for testing personnel to be more efficient.

New testing tools could be another way to better leverage the current testing staff. Again, this is more of a long-term solution. It takes time for testing personnel to learn the new tool, and start using it effectively.

Conclusions

In general, comparing testing ratios is not the most effective way to convince management to add new testing resources.

If you must compare testing ratios, be aware of how different factors compare to those within your organization.

If you have data, you can make a strong case for new resources. However, collecting all the data necessary is not an easy process.

Even once you have agreement, collect data to correlate with your savings expectations. Having actual savings data can greatly help in doing future justifications.

Think about how you might detecting this class of errors earlier in the process, or prevent the errors completely by making changes in the design process. If changes are made to prevent the defect, maintain a check in the process until you are certain that the change has been effective.

Where Did the System Test Department Go?
(Or, Through the Hourglass and What Novell Found There)

Charles D. Knutson
ComSoft Consulting
962 NW Cypress Ave.
Corvallis, Oregon 97330
cknutson@csconsult.com

Abstract:
Over a period of nine years, the organization of system testing at Novell, Inc. has shifted along a spectrum from dependent to independent testing and back again. This paper chronicles the rise, demise, and rebirth of system testing at Novell during that time period. As a case study, this history of a shifting system test organization within a single company provides an excellent opportunity to present and discuss lessons learned, including advantages and disadvantages of the various organizational approaches, and to share perspectives from individuals who were there through the changes.

Bio:
Charles D. Knutson joined Novell in Provo, Utah as a System Test Supervisor in early 1989, and became System Test Manager for client products shortly thereafter. Mr. Knutson left Novell in September 1994. He founded ComSoft Consulting in Corvallis, Oregon, and is a doctoral candidate in Computer Science at Oregon State University. Mr. Knutson holds B.S. and M.S. degrees in Computer Science from Brigham Young University. Prior to joining Novell, Mr. Knutson was a software engineer at Hewlett-Packard in Sunnyvale, California.

Where Did the System Test Department Go? (Or, Through the Hourglass and What Novell Found There)

Charles D. Knutson
ComSoft Consulting
Corvallis, Oregon 97330
cknutson@csconsult.com

Abstract

Over a period of nine years, the organization of system testing at Novell, Inc. has shifted along a spectrum from dependent to independent testing and back again. This paper chronicles the rise, demise, and rebirth of system testing at Novell during that time period. As a case study, this history of a shifting system test organization within a single company provides an excellent opportunity to present and discuss lessons learned, including advantages and disadvantages of the various organizational approaches, and to share perspectives from individuals who were there through the changes.

1. Introduction

There are as many ways to organize a system test effort as there are companies trying to produce quality software. Some companies do very little structured testing, others place testing functions within individual development teams, others maintain independent test organizations, and still others export much of their testing out of house. The motivations behind these approaches to testing are as varied as the companies that implement them, the rationales as troublesome to debate as religion and politics.

However, companies occasionally go through transitions, reorganizations, and internal restructuring, and in the wake of these changes, testing functions can shift along the spectrum. In 1987, Novell, Inc. first began to do structured system testing. Over the following nine years, through massive growth, then restructuring and downsizing, system testing at Novell has traveled along this spectrum from one end to the other. This paper explores lessons that have been learned along the way.

This kind of case study is valuable because of the perspectives of many individuals who witnessed the transformation of testing at Novell from the early days to the present. Rather than trying to compare the experiences of Company A with Company B, these individuals instead can offer insight into their experiences with system test within the same company over a period of years. The author is indebted to those individuals, current and former employees of Novell, who have shared their perspectives, experiences, and above all, their commitment to quality.

2. The Organizational Spectrum

Not all companies are committed to quality, and not all companies understand the need for system testing. But those that do understand it also appreciate one of the great religious questions of software quality assurance, "Who owns system test?" Is it development's responsibility? Does

it belong to a separate independent organization within the company? Or does it belong outside the company entirely?

These perspectives can be viewed as lying along a spectrum (see Figure 1). At the extreme left edge of the spectrum is the default--no testing at all. In reality every engineer does at least a little testing whether he realizes it or not, but at this point there is no formal focus on quality. At the right edge of the spectrum is another extreme--the company cares about quality, but no one inside knows what to do about it, so it's exported out of house to a third-party test organization.

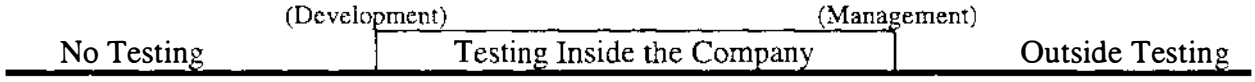


Figure 1. Spectrum of testability.

We can view this spectrum of testing as being more dependent on development toward the left, and more independent from development toward the right. At the far left, the only testing that is done (if any) is by the development engineer. At the far right, someone (probably a manager) owns a relationship with an external test organization, and the burden of quality assurance is so far from the engineers that it is not even within the walls of the same company. Between these two extremes lies the most common scenario--system testing that takes place inside a company.

Testing within a company lies along a similar spectrum. At the left edge of the spectrum, testing is imposed on development engineers. There is no system or integration testing per se, and engineers are responsible for the quality of their product with no independent verification before the customer sees it. At the right edge of the spectrum, test functions are exclusively the domain of an independent system test organization. Engineers code products and throw them over the wall to the quality assurance folks who either accept or reject, and throw bad ones back over the wall for development to repair.

3. From Little or No Testing to an Independent Organization

In 1987 an embryonic system test department was formed at Novell, consisting primarily of individuals from service and support, and involving few if any trained software engineers. At that time, products were beginning to be more complex, the test configurations a bit more unwieldy, and there was a strong sense that someone should be looking at the product before beta customers did. As one of the original members of that test organization put it:

> *"In the growing stages, I see that it [system test] had a lot of credibility. Not only did people buy into a much needed function, but they [testers] had a lot of credibility and objectivity."*

There was an appreciation of the objective view that another set of eyes could lend to the product. There was also value seen in the ability of this new group to maintain a single lab in which the various components could be brought together for testing. But the actual efforts of that

test organization were little more than taking the car around the block and kicking the tires. Still, it represented a step up at the time, and was generally appreciated by development.

Within two years after this humble beginning, the system test department had grown in size, function, and talent. Part of this was due to efforts of key players in management to assure for system test engineers the same salary and career path available to development engineers. This meant that a new engineer with a bachelor's degree in Computer Science could start in either a development or testing spot, and make the same money. This provided a dramatic boost in the caliber of the engineers joining the department, but led to animosity on the part of some development engineers. The dynamics between the two groups were beginning to change.

From 1989 to 1991 the system department began a push to automate testing, and continued to hire educated and experienced software engineers who saw a career path in software testing. The system test department grew to over one hundred engineers, and included "the super lab," a facility with more than one thousand workstations that could be configured in myriad ways for testing. This organization was independent from development, with a director reporting to the vice president of development. There was a strong sense of identity within the system test department, and some engineers actually transferred from development to system test. One senior Novell development engineer moved into the test organization because he believed that the system test represented the most challenging engineering task to which one could aspire. As he put it, a development engineer can always finish a product and be done, but a system test engineer's test suite can never be done. The system test engineer has a fundamentally more frustrating and challenging task than the development engineer.

3.1 The Independent Approach, Pros and Cons

In the minds of some system test purists, Novell's system test department in 1991 represented a pinnacle of sorts. There was much to recommend it as an example of how to organize and implement an independent system test organization. The following are some of the positives associated with this independent test organization. The issues in this list are loosely organized from most beneficial to least.

Shared focus and culture. Development and testing are by nature somewhat inherently at odds. Development is a constructive activity, while testing is destructive. Development engineers do what's best for the customer by building a product. Test engineers do what's best for the customer by trying to break the product. When test engineers are gathered in a single organization, they can share a common focus. They gain praise and rewards from their peers for being an effective obstacle through which the development engineers must pass. As a group, they understand that the customer gets a better product when they are tough to please, so there is a supportive organization in which they can give way to the dark side and dance on the graves of the engineers whose lives they have made miserable, all in the name of quality.

Career path for test engineers. In most companies, testing is viewed as an ancillary function, while development is viewed as a "real" engineering job. By gathering test resources under a single umbrella, a career path is established for test engineers. By being good at testing, they can

be rewarded and promoted along a path that parallels development. In addition, there are supervisor and manager spots available in this organization which allow individuals to gain promotions and experience without having to move to development to do it. One of the most significant advantages to this approach is that excellent engineers can be recruited into this kind of organization, because they see a career path. Otherwise, if they view testing strictly as secondary in the eyes of upper management, they will never seriously consider taking a position as a tester.

Independent quality control. This is both a blessing and a curse. It means that there are separate management chains reporting to upper management (some recommend that testing and development should join no lower than at a vice-president level). The blessing is that if development is not getting the job done, someone is chartered to blow the whistle. The curse is that whenever someone blows the whistle, there is resentment. But at least upper management can get an accurate picture of product readiness when it really needs one.

> *"The objective was to ship a good product. That was my end goal. . . . I could count on the difference of objectives. Although that put testing people in a difficult position, it just made it a lot easier to ensure what was happening. As a product manager, or marketing manager, I went to test and said, 'OK, what to you think about this product?' And I didn't have to worry about them being managed by a development manager that wanted to ship without going through a rigorous test."* (Former marketing manager)

Cohesive product vision. The nature of development work often involves a very narrow focus on a particular piece of a product. This leads to both technical and product myopia on the part of many development engineers. Testing, on the other hand, requires a broader view, at a technical as well as a product level. Just managing the versions of all the pieces that compose a large product, and knowing which ones are supposed to work together, is a large undertaking, and one typically easier for testing to do.[1]

Shared testing resources. There is a certain economy of scale when test resources are in a single department. This is especially true in the area of shared automation technology. If we accept the fact that test automation can lead to faster and more effective tests, we are immediately faced with the cost of developing such tools. With a distributed test effort, there is rarely enough critical mass to develop effective tools, because the efforts are splintered. But when a single organization includes test teams for various products, it is easier to pool resources to create automation that can be shared across the department. In addition, lab resources can often be effectively shared.

Imposed configuration management. This is subtle, but important. Many companies recognize the importance of configuration management, but there is often a phase of painful transition once it begins to be mandated by management. The test organization needs to know that what it is

[1] As a personal example, I wrote OS/2 device drivers at Hewlett-Packard before becoming OS/2 system test supervisor at Novell in 1989. I learned low-level details from my device driver experience, but I gained vastly more knowledge about OS/2 while functioning in a system test capacity, and the knowledge gained from testing was more balanced and broad-based.

testing is the latest and greatest stuff. To test against non-current stuff is to waste all the time and resources involved in testing the wrong product. Configuration management is the natural solution to this, and system test can often be a positive catalyst in solidifying its importance in the minds of development engineers.[2]

Automated bug tracking. It doesn't take a dedicated department to do automated error tracking, but it sure helps to have the weight of a hundred or so test engineers, and a plea from a single source (the director of the test department) behind the effort. Since such a system invariably involves some kind of database management, the resources for maintaining the system obviously lies in the test department.[3]

If the history of the Novell system test department could be simply summarized by these advantages, it would undoubtedly still exist today in the same form. However, there were notable disadvantages to the organization. This list is loosely organized from most problematic to least.

Adversarial relationships. This is probably the single biggest problem with independent quality control--there is a natural conflict of priorities, with developers building, and testers trying to tear down. In a healthy environment, development sees testing as a necessary obstacle, which, if overcome, means the customer gets a great product. But too often, testing is just seen as the enemy, or as a non-team player.[4] There were situations in which relations between testing and development were excellent, but these were the exception, not the norm.

> *"I think some test engineers did it better, where there weren't those problems. They'd go up and sit down with the engineers, with the developers, and be involved, and that worked a whole lot better. But that wasn't the attitude of the testing department as a whole. . . . [For good relationships to happen], both sides ignored the general push by both of their respective departments, and they had the personality match to do it. . . . Pretty rare."* (Former Novell engineer who worked in testing and then development)

End of the process mentality. There is debate within the system test community concerning how far back in the process testing should get involved. One view has them doing essentially unit, integration *and* system test functions in tight coordination with development. The other view has testing in more of a QA function, where system test sits at the end of the process with a big piece of chalk. They test the product, and if it fails, they put a big X on it and throw it over the wall.

---

[2] Some of my earliest memories as a system test supervisor was of a conflict with development when I wanted a product delivered as if I were the first customer. The development engineers indignantly announced that it was on their server and I could go get it, and that this is the way it had always been done. My pleas about wasting resources by testing the wrong thing fell on deaf ears, since our resources (in their minds) weren't that valuable anyway. It was literally years before the corporate culture had evolved to the point that configuration management was accepted by the old guard without visible hostility.

[3] At Novell, our system was called NETS (Novell Error Tracking System), and was built and maintained by the system test department. We produced buttons that said, "Catch your bugs in NETS," and we managed to get the company to chant the mantra, "If it's not in NETS, it isn't a bug."

[4] One of my most memorable experiences as a system test manager was being indignantly asked in a meeting by a development manager, "Don't you trust me?!" My response was that "trust" was not in my job description, that Novell paid me to test, not trust. I said that, like Reagan to the Soviets, I would "trust but verify."

Novell's system test department always leaned toward the QA function, and this was probably part of the source of conflict between development and testing. There was less a feel of helping to build a good product, and more a sense of just being a roadblock.

Conflict of societal status. Despite equivalent pay scales and career opportunities, there is still a deep-seated sense among many development engineers that system testing is a second class function, and that testers are second class citizens. The following excerpt from a conversation with a former Novell engineer (who worked in both testing and development) summarizes the feeling:

> *"I'll give you a strong opinion here, and it may be offensive to some. I think test engineers are only necessary because of bad development engineers. . . . I think if development engineers were the quality they should be, they would know how to write decent code. They would be good enough engineers that they could develop code that would have relatively few bugs. I'm not saying they would never have bugs, cause that's impossible, but their design architecture would be such that it would tend to reduce bugs."*
> *"So you're saying that your best engineers shouldn't be your test engineers."*
> *"Right!"*

Encouragement of sloth by development engineers. When a test department begins to effectively test product, many development engineers begin to lose a sense of responsibility for the quality of their own product. They stop doing unit testing and other kinds of testing, because they know testing will take care of it. When that attitude becomes institutionalized, it leads to incredible situations in which a problem is found in a product, and testing takes more of the blame for not finding it than development takes for having created the bug in the first place.

Delays and inefficiencies. If a system test department does not develop effective automated tests, they can introduce a tremendous (albeit arguably necessary) overhead on the time to market for products. This can also be a source of conflict with development when there is a perception that testing is adding overhead and delays without returning much quality for the cost.

Lack of understanding by testing of core technology. This also contributes to the general tendency toward ill-will between development and testing. The black-box purists argue that testing shouldn't have to know much about the underlying technology, just the input and expected outputs. But when this mentality is followed strictly, it leads to a technology gap between development and testing, which lessens testing's credibility, and reaffirms them as second class citizens in the minds of development engineers.

4. Through the Hourglass

In its heyday, Novell's system test department had individuals on both sides of the religious debate holding different opinions, and forces on both sides exerting influence. One of the key lessons learned is that it is up to the leadership of an independent system test organization to make peace with development. We could argue idealistically that the onus should be on both

sides equally, but in reality, the weight of history sides with development, and independent testing will always struggle against that. As Boris Beizer [1] prophetically stated:

> *"Every time the beneficial conflict between project and QA erupts into a confrontation, both are damaged. If QA attempts to impose unrealistic goals, to make the software development effort noncompetitive, to impose a burdensome overload of non-productive paperwork more concerned with form than function, then . . . software quality will suffer and, eventually, independent QA will disappear."*

Unfortunately, despite the tremendous advantages provided by the independent system test organization, the disadvantages proved difficult to manage, and the conflicts often erupted into confrontation. Beizer predicted that in such situations independent QA would disappear, but with an organization of 120 engineers at its peak, it was hard to imagine the department simply going away. The wall between development and testing seemed as firm as the one between East and West Berlin.

Still you don't just kill a healthy system test department. First you have to make it worth killing. An understanding of the economics of testability will help to make it clear how the system test department was weakened to a point that made it easy prey for more dramatic decisions.

4.1 The Economics of Testability

We live daily with a working understanding of principal and interest. It's been said of interest, "Them that understands it, earns it. Them that don't, pays it." In general there are positive and negative spirals associated with principal and interest. Understanding these will help us understand the economics of testability.

When an individual takes home more money than he earns, he has extra money to invest. Suppose that he has a farm that is being paid off. This extra money each month can be applied to the principal on the loan. As a result, the following month the principal remaining on the loan is smaller than it would have been without the extra payment, so the interest owed is also smaller, therefore a normal payment will naturally result in more money toward principal than it would have otherwise been. If the individual continues to earn more than is required, and applies the extra income to the debt, the farm will eventually be paid off on an accelerated schedule. At that point, the farm can be maintained for a fraction of the cost of originally buying it.

The opposite spiral is probably the one more familiar to most of us. We begin with some debt, but then we spend each month more than we earn, and put the difference on a credit card. After one month, we have increased the principal on the loan, which increases the interest owed, so that the same payment will have less impact on reducing the principal. If we compound this with continuing to add to the principal each month by spending more than we earn, we place ourselves on a steady spiral leading toward bankruptcy.

The economics of testability operates on nearly identical principles. The positive spiral begins with a debt consisting of the gap between the current set of products that need to be adequately

tested, and a set of automated test suites. The gap between what we can test and what needs to be tested is the principal on our debt. Each month we apply resources to the task. Some testing needs to be performed just to maintain the status quo and ship product. This is the interest on the loan. It must be paid regularly, but does not help us to close the gap. To do that we must apply extra resources. In other words, an effective test organization must have more resources available to it than it needs to minimally test products. These resources can be applied to the gap between products and test suites. As the gap begins to close, it takes fewer resources to test products, because more of the testing is effectively automated. In other words, the interest is shrinking because the debt principal is shrinking. This frees up even more resources, which can in turn be applied to the gap. At a certain point the farm can be paid off, which occurs when automated tests exist to sufficiently test the entire product. At this point a test team can actually shrink in size and still maintain an adequate test. Or it can handle an increasingly complex product line without having to grow.[5]

Similarly, the negative spiral begins with the same gap between product and test suite, but with only enough resources to manually run the few tests that exist, but no resources to catch up and close the gap. As a result, the team maintains an ineffective system test until the product grows in complexity, at which point the gap widens, and ever more features fall through the cracks and remain untested. At a certain point this organization, doomed to failure by inadequate resources will collapse under its own inadequacy, and someone else will pick up the slack--either another independent organization, or the development organization responsible for the product.

4.2 The Downward Spiral

By late 1992 there were forces within Novell that believed independent testing should not exist, and that the resources could be better used if distributed throughout the development organization and merged with development teams. But a large organization has a tremendous amount of inertia, and suggestions that the department should be dissolved were rejected as impractical by the powers that be.

But an argument was made by some development managers that if development teams built better products, we would need less system testing. In other words, we can save money by trimming the system test department if we can beef up the unit testing functions on individual development teams. This argument was not completely out of line with those who held that the wall between testing and development was too severe, and that there needed to be tighter relationships between testing and development.

---

[5] As a system test supervisor at Novell, I was approached by the director of system test during a budget crunch because I had five engineers on my team, which was considered fat. I was being asked to lose one engineer. We were also one of the few teams actively automating our entire test suite. I argued to the director that if he would let me keep my five engineers for one year, he could then take two instead of one, because we'd need fewer engineers to get the job done if we were permitted to get it done right the first time. To his credit he left us as we were, and the suite we created formed the core of system test for that product for the next five years, allowing us to completely regress our product in less than 24 hours, even after the team had shrunk to three members.

Once upper management accepted this argument, there was discussion concerning how best to facilitate development teams doing more unit testing, and then how best to realize the savings in a smaller system test organization. These two objectives were combined into a single proposal. Since the test engineers knew testing better than anyone, and we were going to be able to reduce the system test department eventually anyway, and we needed better test resources on development teams, why not just move resources directly from testing to development? This was agreed to, and approval was given for immediately moving between 10% and 20% of the system test department personnel directly to individual development teams. As soon as this happened, the economics of testability changed from a positive spiral to a negative one.

The argument that better unit test implies fewer resources needed for system test is questionable. Certainly better products from development imply shorter test cycles because there are fewer bug fixes, and hence, fewer regression test cycles. This does provide some savings in resource expenditures. But when the bulk of the test suites are fully automated by a test department, the cost of a regression is negligible in terms of manpower. Instead, I believe that the cost reduction in personnel happens when the farm is paid off, and when the bulk of existing products have adequate automated tests available. Even if it could be argued that better unit testing would provide cost savings through less system testing, there would still be a lag time for this to be realized. By immediately moving testers to development teams, the system test department experienced an immediate shortfall in the resources required to stay even with the products. In other words, it embarked on a negative cycle, with no extra resources to invest in further automation, and barely enough resources to adequately perform system tests.

Several other factors contributed at this time to a weakening of the system test department. At the time that resources were suddenly reduced, several major product releases were scheduled, effectively doubling the amount of actual system testing that needed to be done, further crippling the ability to get ahead through automation. The second factor was the first engineering layoff in the history of the company, which decimated the NetWare Products Division. In this layoff, the engineers most severely hit were those that had recently been transferred to development from testing. Development managers, when faced with a choice of who to layoff, in general turned toward their testers who were considered second class citizens and who were also the newest members with the fewest attachments. A third factor was that many of the surviving "unit testers" (especially the best engineers) were converted into development engineers to compensate for the loss of manpower. These conversions, coupled with the layoffs, left the majority of development teams in the same situation they had been in before the transfer. They still had no real test resources, only now system test was even less capable of picking up the slack.

The result at this point was a slow and inevitable spiral toward death. With each new product feature, testing fell further and further behind in its ability to adequately handle the job. Testing began to take longer and longer for fewer results. It takes only a little wind to blow over a house of cards, and this was the condition of the system test department at Novell in 1993. When the suggestion was made by the same development managers to the same upper managers that the system test functions be merged completely into development teams, there was much debate, but approval was eventually given, and overnight the system test department ceased to exist as an independent organization. A small group of a dozen test engineers was left behind to try and

manage integration issues and to see that the various products would actually load on the same file servers together, but the efforts of one dozen engineers to perform quality control on the work of several hundred is obviously relatively minuscule.

5. Back to Dependence

At this point, system test at Novell had come full circle, from dependence to independence and back to dependence again. Certainly this new state of affairs was far better than the original state of little or no testing at all. Because of the discussions and debates, development managers were far more aware of testing issues than they had previously been. More importantly, unlike the unit test transfer, in which a system test department still existed, this time there was no safety net separate from a manager's own team, so development managers began to take testing more seriously.

As an interesting side note, a disproportionately high number of the development managers at Novell are former managers, supervisors, and engineers from the system test department. I believe that there are several reasons for this. One is simply that the energy, vision and focus of the test department during its heyday put them in the position of hiring extremely qualified people with a lot of potential. But another factor is that once testing became a critical function of development teams, individuals with test experience were more of a premium, and were looked upon by directors as having the ability to manage both development and testing functions. I also believe that one of the most critical reasons for this is that testers, living as long as they did at the end of the food chain, could see much more of the product and its place in the product line. They were also in a better position to understand the entire product life cycle.

5.1 The Dependent Approach, Pros and Cons

Those who had pushed for development to own testing functions were motivated in part by a desire for higher quality products. They saw dependent testing as the way to achieve it. And indeed, there are significant advantages to this approach. The following are some of the positives associated with this approach to testing. The issues in this list are loosely organized from most beneficial to least.

Tighter involvement throughout the life cycle. Testing no longer happens just at the end of the product development cycle--testers are involved from the very beginning. By and large testers and developers now share the same vision, focus, and motivations. When these are unified, better products can be delivered by a development team. Testers can also help with specification, design, and code reviews and create tests at all levels from unit to system.

Teamwork relationship. Testers are no longer the enemy. The same individual manages both testers and developers, so conflict is easier to handle. In ideal situations, the test engineers on a development team can actually train the development engineers in the art of unit testing and assuring the quality of their own products. The test engineers can act as a kind of leaven in the team to raise the issues of testing to a team that might otherwise be oblivious.

Development engineers more responsible. Even though there are testers on the same team, the practical experience is that development engineers know testers can't handle all the code being developed by the team. So engineers must take more responsibility for the quality of what they build.

Fewer delays and inefficiencies. There is tighter involvement during testing and bug fix cycles, so there are far fewer institutionalized delays and inefficiencies.

Greater understanding of core technology. Test engineers are more exposed to the core technology. Since they are not enemies, they have greater potential access to necessary information.

There are notable disadvantages to dependent system testing as well, otherwise Beizer [1] would not have been inspired to write, "Independence of QA from the project hierarchy is probably the most important doctrine that can be established." The following are some of the disadvantages that occur with dependent testing. This list is loosely organized from most problematic to least and mirrors the list of advantages to an independent test organization.

No independent quality control. The fox guards the hen house, and there are opportunities for abuse. However, if development managers are empowered and held responsible, this factor can be minimized. Still, upper management now has only one source of information to try and gauge the quality of a product. This would never be a problem if all development managers were always impeccably honest and selfless.

No career path for test engineers. The single largest complaint I have heard from development managers functioning under the new system is that they can't hire good engineers to be testers on a development team. The best engineers see the second-class citizen label, and will typically lean strongly toward development positions. As one development manager said,

> *"I have the general philosophy for test engineers that if you don't provide a career for them, you're not going to be able to keep good people in that discipline. . . . The good part in the old days was there was a career path for [test] engineers. And you had some really talented people doing that kind of work."*

No shared focus and culture. It could be argued that the negative, destructive focus that is sometimes shared by a system test organization is a necessary evil that gives test engineers their edge. Put them with development engineers and they just can't get too excited about finding bugs anymore, because their immediate society doesn't accept that. It is possible that they will lose their testing aggressiveness because of this, and be less effective as testers.

Conflict of society status. This was one of the negatives listed under independent testing, and it's still a disadvantage here. I'm not sure we'll ever overcome the stigma that sometimes is attached to system test departments and personnel. Perhaps when test departments begin to develop and

ship automated test tools, bug tracking systems, and similar technology, they gain credibility as development engineers of test products, and hence, gain greater credibility from development.[6]

No cohesive product vision. When test engineers merge into development teams, they naturally are more distanced from other test engineers. So the inter-product focus that exists in a test department is lost. Test engineers on development teams can still own the overall vision for that team, but lose the connection to other products.

No shared test resources. This can be overcome by making test resources available to test engineers from different teams.

The need for some form of system test still exists. The arguments in favor of dissolving system test notwithstanding, there still is a need for system testing, and a handful of individuals parked on development teams are not in a great position to do it.

I should point out that configuration management and automated bug tracking survived as benefits at Novell, and were used even after the demise of the test department. But I am unsure if they could have risen as effectively as they did had the system test department never existed in the first place.

6. Conclusion

Novell system test passed through the narrow neck of an hourglass and dramatically transformed the way its testing functions are organized. The natural spasms that happened after the test department was dissolved have calmed, and the new system seems to be working. But the need for independent system testing hasn't gone away completely, and a new system test department has grown from the ashes of the former, spawned by the original dozen engineers left behind. It now has more than 60 engineers, and includes all of the corporate integration testing functions, coordinating products from several divisions. But it doesn't look after the quality of every component the way the former system test department did. Rather it plays more of a global role. The quality of each piece is now exclusively the responsibility of each team.

So what does the future hold? That's impossible to predict, but one thing is for sure. There is now a more pervasive awareness of quality issues at Novell because testers live on every team than there was in the past when there was one large policeman with a big stick. Some believe testing has never been better, while others long for the glory days of the system test department.

7. References

[1] Beizer, Boris. *Software System Testing and Quality Assurance.* New York: Van Nostrand Reinhold Company, 1984.

[6] As a system test manager, I always stressed that we were a development team whose product was a system test, and that our product was more challenging to create then the one we were going to test. We typically wrote as much code (if not more) then the development engineers, and my teams were typically higher educated and at least as well paid as our counterparts in product development.

Decomposition of Multiple Inheritance DAGS for Testing

Chi-Ming Chung, Timothy K. Shih, Ying-Feng Kuo, and Chun-Chia Wang
Department of Computer Science and Information Engineering
Tamkang University
Tamsui, Taipei Hsien
Taiwan 251, R.O.C.
Fax: Intl. (02) 623 8212
email: cmchung@cs.tku.edu.tw

Abstract

Software testing is a key issue to assure software quality. It is an important area in the research of Software Engineering. In line with the methodologies of object-oriented analysis and design widely developed, many testing techniques have been proposed. However, not many focus on the testing criteria of an inheritance hierarchy. In this paper, we introduce a concept named URI (Unit Repeated Inheritance) in Z to realize an object-oriented testing based on inheritance relation. The approach describes an ILT (Inheritance Level Technique) method based on URIs as a guide to test an inheritance hierarchy. Also, two testing criteria algorithms, intra level first and inter level first, can be formed based on the proposed mechanism.

Key words: Software Testing, Inheritance Hierarchy, Unit Repeated Inheritance, Inheritance Level Technique, Z Notation

1 Introduction

Software testing is the process of finding programmer errors or program faults in the behavior or in the code of a piece of software. Many testing methodologies for procedure–oriented languages on detecting control flow and data flow errors have been proposed[5, 7, 8]. However, these traditional testing methods are not sufficient for software development in the object–oriented paradigm due to the new data abstractions and control abstractions such as inheritance, instantiation, polymorphism, and message passing. The importance of an efficient testing criterion for automatic testing brings the attentions to many researchers[1, 10, 11]. Therefore, object-oriented systems still need testing.

Inheritance is a type of hierarchical relationship among objects in a class structure and depicts a hierarchy that makes a design object–oriented. It is also a mechanism that allows for a class, the child class, to inherit properties, features, and methods, from one (single inheritance) or many classes (multiple or repeated inheritance), its parent classes. The child class can be refined by modifying or removing the inherited methods, or adding new properties. Although multiple (or repeated) inheritance relates more closely to the real world situation, the usage of multiple (or repeated) inheritance could easily introduce more software errors in the analysis, design, and coding of object–oriented software. Thus, multiple (or repeated) inheritances should be avoided as much as possible. In the case when it is necessary to use repeated inheritances, the testing of an inheritance hierarchy seems to be important.

To overcome the problem of inheritance relationship, several methods for the testing of inheritance have appeared in the literature[12, 13]. However, none of these proposed techniques explicitly address the testing issues in an inheritance hierarchy. In[13], the research on *method sequence specification*, proposes a specification technique that specifies the causal relationship between methods of a class, but the article does

not consider the relation between classes of an inheritance hierarchy. Meanwhile, although the child class is obtained by refinement of its parent class, it seems natural to assume that a parent class that has been tested can be reused without any further retesting of the inherited properties. However, this intuition is proved false in[10]. Harrold et al.[12] propose an algorithm that determines the minimal subset of retesting for the C^{++} programming language, but there are still few encouraging results in the software testing of an inheritance hierarchy. For example, overuse of multiple (or repeated) inheritances may result in name conflicts. Our objective of this research is to provide a good testing strategy of an inheritance hierarchy that presents a resonable approach to the use of multiple (or repeated) inheritance.

Z is one of the more popular formal specification languages. It makes use of *schemas* to structure specifications[14, 15, 16]. In this paper, we use *Z* notation to describe a model of repeated inheritance and introduces a novel concept named URI as the testing unit of our object-oriented testing. The following subsection summarizes the *Z* notation briefly. Section 2 discusses a mathematical graph model of inheritance graphs with *Z*. Section 3 proposes a mechanism and an algorithm of decomposing an inheritance hierarchy into a number of URIs. We also prove the soundness and completeness of the proposed algorithm. Section 4 illustrates the ILT method with a word processor application program. Section 5 presents two testing criteria algorithms for conducting ILT testing. The last section addresses our conclusions and contributions.

1.1 A Brief Review of Z

Z notation is a language for expressing formal specifications of systems. It is based on typed set theory, coupled with a structuring mechanism, i.e., the schema calculus is one of its key features. A schema introduces a named collection of variables and relationships among variables that are specified by axiom definitions. Schemas are used to describe both the static aspects of a system (e.g., the structure of a program) and the dynamic aspects (e.g., the execution). Schemas can be generic; thus polymorphic functions can be defined. Every variable introduced in a *Z* specification is given a type. These types can be given set names or can be constructed by type constructors (e.g., tuple, schema product, or the power set constructors). Free type definitions[1] add nothing to the power of the *Z* language but ease the definition of recursive objects. Free type definitions can be translated into other *Z* constructs. An abbreviation definition using symbol "==" introduces a new global constant. The identifier on the left becomes a global constant and its value is given by the expression on the right. An axiomatic description introduces one or more global variables and optionally specifies a constraint on their values. A short summary of *Z* notation operators is given in the appendix. In the discussion follows, we will introduce concepts and syntax of *Z* notation when necessary.

2 Inheritance Graph

One of the main benefits of object-oriented programming languages is that it facilitates the reuse of classes. A class is a template that defines the *attributes* that an object of that class will possess. A class's attributes consist of *data members* (*instance variables*) and *member functions* (*methods*). Classes are used to define new classes or subclasses through a relation known as *inheritance*. Inheritance facilitates an object-oriented design that classifies objects into various levels of *classes* in an *object taxonomy* (i.e., object hierarchy) and permits a subclass to inherit attributes from its parent classes and either extend, restrict or redefine them. Some systems allow a subclass to have only one parent class from which the subclass inherits attributes of the parent class (i.e., single inheritance). Other systems allow a subclass to inherit attributes from its multiple parent classes, called a multiple inheritance. The latter, which relates more closely to the real world situation, could be even more complicated in that the same attribute of a parent class can be inherited by a subclass more than once via multiple paths in the inheritance hierarchy. One of the delicate problems raised by the presence of multiple inheritance is what happens when a class is an ancestor of another in more than one way. If you allow multiple inheritance into a language, then sooner or later someone is going to write a class D with two parent classes B and C, each of which has a parent class A. This situation is called a *repeated inheritance* and must be dealt with properly[6]. Researchers suggest that the usage of repeated

[1] Free type definitions are discussed in[14, 15] as a short mechanism to introduce new types in Z.

(multiple) inheritance could easily introduce more errors in the design and implementation phases and thus a repeated (multiple) inheritance should be avoided as much as possible. In the case when it is necessary to use repeated (multiple) inheritances, the software testing of an inheritance hierarchy seems to be important. We believe that the most suitable mathematical model for describing an object taxonomy with repeated inheritances is a DAG (Directed Acyclic Graph) with no loops (edges start and end at the same node). In this section, we use Z notation as a tool for our presentation.

A *basic type* defines a kind of objects without specifying the detailed contents of the objects. Classes in an object taxonomy are vertices in a DAG. The following free type definition defines a vertex type:

$$[V]$$

Note that at the specification level, we do not care about the detailed representation of a vertex. However, we need at least some objects to start with, in order to construct other objects in our system. An *abbreviation definition* (i.e., using a $==$ sign), similar to the macro definition in a procedural language, is used to represent inheritance relations as edges in the taxonomy. A directed edge, represented by an ordered pair of vertices (e.g., (v_1, v_2) in the domain of $V \times V$), starts from the first vertex and ends at the second. A path in the graph is a consecutive nonempty list of vertices. Note that $\text{seq}_1\ X$ is a nonempty list of objects of type X in Z. The following are abbreviation definitions for edges and paths:

$$E == V \times V$$
$$P == \text{seq}_1\ V$$

In Z notation, a specification can be decomposed into a number of *schemas*. Specifications are specified in mathematical operations known as the *schema calculus*. Schema calculus is a mechanism of composing a schema from other predefined schemas by using logic operators or schema operators defined in Z. A schema is similar to a class in C^{++}, or a package in *Ada*. However, a schema is relatively declarative. That is, one specifies only *what* the specification is instead of *how* a procedure computes. A schema consists of a declaration part (above a horizontal line) and a predicate part (below the line). The declaration part specifies variables used in the schema and *schema references* which are similar to property inheritance. Using a schema reference, a child schema can access variables defined in its parent schema. The predicate part specifies *preconditions* the schema must hold and a number of functions restricting values of variables in the schema. The following schema, *InheritanceGraph*, introduces a DAG mathematical model that denotes an object taxonomy. Note that $\mathbf{P}\ T$ denotes the power set of a type T.

InheritanceGraph

$$vertices : \mathbf{P}\ V$$
$$edges : \mathbf{P}\ E$$
$$paths : \mathbf{P}\ P$$
$$graph : \mathbf{P}\ V \times \mathbf{P}\ E$$
$$roots : \mathbf{P}\ V$$
$$leaves : \mathbf{P}\ V$$

$$graph = (vertices, edges)$$
$$\exists i : \mathbf{N}_1 \bullet \exists p : P \bullet p \in paths \Rightarrow p(i) \in vertices$$
$$\exists i : \mathbf{N}_1 \bullet \exists p : P \bullet p \in paths \wedge i \geq 1 \wedge i < \#p \Rightarrow (p(i), p(i+1)) \in edges$$
$$\exists p : P \bullet p \in paths \Rightarrow (\forall i, j : \mathbf{N}_1 \bullet i \neq j \wedge i \leq \#p \wedge j \leq \#p \Rightarrow p(i) \neq p(j))$$
$$roots = \{\ v : V \mid \forall v' : V \bullet (v', v) \notin edges\ \}$$
$$leaves = \{\ v : V \mid \forall v' : V \bullet (v, v') \notin edges\ \}$$
$$roots \subseteq vertices$$
$$leaves \subseteq vertices$$

The taxonomy is represented by a *graph*, which consists of a set of *vertices* and *edges*. The prefix operator "#" denotes the number of elements in a set. A list (i.e., a sequence) is treated as a set of maplets from

positive integers (i.e., $\mathbb{N}_1$) to elements of the sequence. For instance, $\{ (1, v_1), (2, v_2), (3, v_3) \}$ is a list of three vertices. Since lists are represented as sets in Z, we use $\#p$ to denote the length of list p. *paths* consists of nodes obtained from *vertices*, which makes ordered pairs (i.e., *edges*) as segments of a path. The taxonomy is a DAG, which has a number of root nodes (i.e., vertices without any incoming edge) specified in *roots* and a number of leave nodes (i.e., vertices without any outgoing edge) specified in *leaves*. The graph is acyclic. That is, there exists no path containing the same node twice or more. Note that $p(i)$ is a function application, which takes i as input and returns the ith element (a vertex) of path p.

Repeated inheritance occurs while a node in the graph is reachable from one of its ancestors via more than one path. The *RepeatedInheritanceGraph* schema contains a schema reference, denoted by Ξ*InheritanceGraph*, that inherits variables and properties from *InheritanceGraph*. The Ξ sign also means that there is no *state change* (i.e., changing the content of schema *InheritanceGraph*) while using variables of *InheritanceGraph*. A *complete inheritance path* is a path that starts from a node in *roots* and ends at a node in *leaves*. Complete inheritance paths are used in the software testing, which will be discussed later in the paper. In a *RepeatedInheritanceGraph*, there must exist at least two paths that share at least two common nodes in the graph.

RepeatedInheritanceGraph

$\Xi InheritanceGraph$

$cipaths : \mathbb{P}\ P$

$\exists p_1, p_2 : P \bullet \exists i_1, i_2, j_1, j_2 : \mathbb{N}_1 \bullet$
$\quad p_1 \in paths \land p_2 \in paths \land$
$\quad i_1 < j_1 \land i_2 < j_2 \land$
$\quad p_1(i_1) = p_2(i_2) \land p_1(j_1) = p_2(j_2)$

$cipaths \subseteq paths$

$cipaths = \{ p : P \mid p(1) \in roots \land p(\#p) \in leaves \}$

So far, we have presented a mathematical graph model for object hierarchy containing repeated inheritances. In the next section, we propose a mechanism that decomposes an inheritance graph.

3 Graph Decomposition

Before we discuss a technique for decomposing a repeated inheritance graph, a number of functions to be used globally in the discussion are given below. An *axiomatic description* in Z contains two parts, the declaration and the predicates. Variables introduced in the declaration must be unique within the specification and have their scope extended toward the end of the specification. Note that the specification we mention here is the one for object inheritance decomposition presented in this paper. The predicate part is optional. When used, restrictions are applied to the variables. Usually, the signatures of function variables are given in the declaration part and the values of those functions are specified in the predicate part. We use a number of axiomatic descriptions to specify some global functions in our specification.

$fcn : \mathbb{P}\ P \rightarrow \mathbb{P}\ V$
$fce : \mathbb{P}\ P \rightarrow \mathbb{P}\ E$

$\forall ps : \mathbb{P}\ P \bullet fcn(ps) = \{ v : V \mid \exists p_1, p_2 : P \bullet p_1 \in ps \land p_2 \in ps \bullet \exists i, j : \mathbb{N}_1 \bullet$
$\quad p_1(i) = v \land p_2(j) = v \bullet v \}$

$\forall ps : \mathbb{P}\ P \bullet fce(ps) = \{ e : E \mid \exists p_1, p_2 : P \bullet p_1 \in ps \land p_2 \in ps \bullet \exists i, j : \mathbb{N}_1 \bullet$
$\quad (p_1(i), p_1(i+1)) = e \land (p_2(j), p_2(j+1)) = e \bullet e \}$

$$
\begin{array}{l}
gredges : (\mathbf{P}\ V \times \mathbf{P}\ E) \rightarrow \mathbf{P}\ E \\
grvertices : (\mathbf{P}\ V \times \mathbf{P}\ E) \rightarrow \mathbf{P}\ V \\
grpaths : (\mathbf{P}\ V \times \mathbf{P}\ E) \rightarrow \mathbf{P}\ P \\
\hline
\forall g : \mathbf{P}\ V \times \mathbf{P}\ E \bullet \forall e : \mathbf{P}\ E \bullet gredges(g) = e \wedge second\ g = e \\
\forall g : \mathbf{P}\ V \times \mathbf{P}\ E \bullet \forall v : \mathbf{P}\ V \bullet grvertices(g) = v \wedge first\ g = v \\
\forall g : \mathbf{P}\ V \times \mathbf{P}\ E \bullet \forall p : \mathbf{P}\ P \bullet \\
\quad grpaths(g) = \{\ p : P \mid \exists i : \mathbb{N}_1 \bullet p(i) \in grvertices(g) \wedge (p(i), p(i+1)) \in gredges(g) \bullet p\ \}
\end{array}
$$

$$
\begin{array}{l}
makegraph : \mathbf{P}\ P \rightarrow (\mathbf{P}\ V \times \mathbf{P}\ E) \\
extendgraph : \mathbf{P}\ P \times (\mathbf{P}\ V \times \mathbf{P}\ E) \rightarrow (\mathbf{P}\ V \times \mathbf{P}\ E) \\
\hline
\forall p : \mathbf{P}\ P \bullet \forall g : \mathbf{P}\ V \times \mathbf{P}\ E \bullet \\
\quad makegraph(p) = g \Leftrightarrow \\
\qquad (\forall i : \mathbb{N}_1 \bullet i \geq 1 \wedge i < \#p \bullet (p(i), p(i+1)) \in gredges(g)) \wedge \\
\qquad (\forall i : \mathbb{N}_1 \bullet i \geq 1 \wedge i \leq \#p \bullet p(i) \in grvertices(g)) \\
\forall p : \mathbf{P}\ P \bullet \forall g_1, g_2 : \mathbf{P}\ V \times \mathbf{P}\ E \bullet \\
\quad extendgraph(p, g_1) = g_2 \Leftrightarrow \\
\qquad gredges(g_2) = gredges(g_1) \cup \{\ (p(i), p(i+1)) \mid i \geq 1 \wedge i < \#p\ \} \wedge \\
\qquad grvertices(g_2) = grvertices(g_1) \cup \{\ p(i) \mid i \geq 1 \wedge i \leq \#p\ \}
\end{array}
$$

A set of paths may contains a number of common vertices (and edges) if these paths intersect each other in the graph. In the graph decomposition algorithm to be discussed, we need to count the number of common vertices and edges. We will use two functions, *fcn* and *fce* (find common nodes and find common edges), for these purposes. Function *fcn* takes as input a set of paths and returns a set of common vertices. Function *fce* takes the same input and returns a set of common edges. For example, let $ps = \{\ (1,3,4,6,8), (2,3,4,6,9)\ \}$, we have $fcn(ps) = \{\ 3,4,6\ \}$ and $fce(ps) = \{\ (3,4), (4,6)\ \}$.

For the sake of convenience, we have three selection functions, *gredges*, *grvertices*, and *grpaths*, which take as input a graph and select the edges, vertices and paths of the graph, respectively. Since a graph is made up from the cartesian product of its vertex set and edge set, we use the standard functions, *first* and *second* in Z, for extracting the first and the second parts of graph g in *grvertices* and *gredges*.

makegraph takes as input a path and constructs a graph from that path. Note that a path is also a graph. *extendgraph* adds a path to an existing graph yielding a new graph. Some standard set notations are used in these functions too. The set $\{$ *Declaration* | *Expressions* $\}$ is a set of values taken from variables in the *Declaration*, with those values satisfy constraints in *Expressions*.

Before the graph decomposition algorithm is presented, we firstly introduce an atomic object of repeated inheritance and discuss its property, followed by a proof of the property.

Definition 1 *A unit repeated inheritance (or URI) is a DAG with its number of edges equal to its number of nodes, and contains exactly two inheritance paths.*

Lemma 1 *For each URI, let CN denote the number of common nodes and CE denote the number of common edges, there must exist a condition $CN - CE = 2$ for the URI.*

Proof: Assuming that there are two inheritance paths, $p1$ and $p2$, construct a URI. Each of $p1$ and $p2$ has m and n nodes and thus $m-1$ and $n-1$ edges, respectively. According to the definition of URI, the number of nodes and the number of edges must be equal, that is,

$m + n - CN = (m-1) + (n-1) - CE$, *hence*

$CN - CE = 2$. □

In the software testing of an inheritance hierarchy, we use URI as an atomic element. In the following, we will prove that a repeated inheritance graph can be decomposed into one or more URIs. Afterwards, we can use the concept of URIs in the development of testing criteria because there may be software errors hidden in the back of a URI, such as *name conflict* errors. For now, we discuss an important property of a URI with respect to an inheritance graph.

Definition 2 *Let $G = (V, E)$, $G_1 = (V_1, E_1)$, and $G_2 = (V_2, E_2)$ be graphs, $\oplus$ is a binary operator on the domain of graphs defined by the following logical equation:*

$$G_1 \oplus G_2 = G \Longleftrightarrow (V_1 \cup V_2 = V) \wedge (E_1 \cup E_2 = E)$$

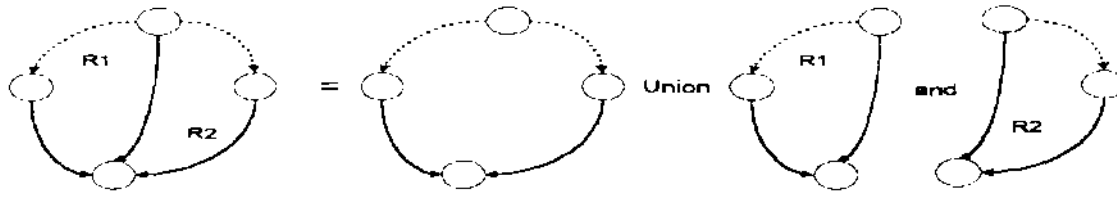
The association of the introduced binary operator is from left to right.

Theorem 1 *Let $G = (V, E)$ be an inheritance graph. If it contains repeated inheritances, then the graph G could be decomposed into a set of unit repeated inheritances (URIs).*

Proof: The proof is by induction on the number of *euler regions* of an inheritance graph[17]. Let $G_r = (V_r, E_r)$ be an inheritance graph with r euler regions, suppose that we remove a common edge, $\overline{e}$, which is one of the edges between any two consecutive euler regions, then the number of euler regions will be decreased by one and two URIs will disappear. We assume that the remaining graph is $G_{r-1} = (V_{r-1}, E_{r-1})$, where the number of euler regions are *r-1*, $V_r = V_{r-1}$, and $E_r = E_{r-1} \cup \{\overline{e}\}$. And then the removed regions are the two URIs, R_1 and R_2, which must contain the removed common edge.

Induction hypothesis: $G_r = G_{r-1} \oplus R_1 \oplus R_2$.

- If $r = 2$, then the claim is true because we observe that an inheritance graph with 2 euler regions satisfy the base case as follows:



Thus, $G_2 = G_1 \oplus R_1 \oplus R_2$

- We now assume that the induction hypothesis is true for an inheritance graph with r euler regions, and prove that this assumption implies that an inheritance graph with $r + 1$ euler regions holds the induction hypothesis. Let $G_{r+1} = (V_{r+1}, E_{r+1})$ and $G_r = (V_r, E_r)$, we know from the property of a graph that $V_{r+1} = V_r$ and $E_{r+1} = E_r \cup \{\overline{bc}\}$, suppose that $\overline{bc}$ is a common edge between the two euler regions existing on G_{r+1}, not on G_r. By the induction hypothesis, $G_{r+1} = (V_{r+1}, E_r \cup \{\overline{bc}\}) = (V_r, E_r \cup \{\overline{bc}\}) = ((G_{r-1} \oplus R_1 \oplus R_2) \oplus (V_r, \{\overline{bc}\})) = G_r \oplus R_1 \oplus R_2$, which is exactly what we wanted to prove. So the theorem is complete. □

Based on the above definitions and theorem, we define a *URI* schema that contains a variable *uriset* which represents a set of URIs with respect to the graph.

URI

$\Xi RepeatedInheritanceGraph$
$uriset : \mathbb{P}(\mathbb{P}\ V \times \mathbb{P}\ E)$

$\forall uri : \mathbb{P}\ V \times \mathbb{P}\ E \bullet uri \in uriset \Rightarrow \#gredges(uri) = \#grvertices(uri)$

$\forall uri : \mathbb{P}\ V \times \mathbb{P}\ E \bullet uri \in uriset \Rightarrow$
$(\forall ps : \mathbb{P}\ P \bullet$
$(ps = grpaths(uri)) \wedge (\#fcn(ps) - \#fce(ps) = \#ps))$

We use an adjacent matrix representation of the graph in our software testing. The two dimentional matrix is denoted as a function maps from a cartesian product of two vertices to the range of a non-negative integer. An non-zero element of the matrix represents there exists at least an edge (or a path) from the first

vertex to the second vertex. Two useful functions, *adjmtx* and *grrep*, are to convert a graph to its matrix representation and vice versa.

$$
\begin{array}{|l}
\Omega : V \times V \rightarrow \mathbb{N} \\
adjmtx : (\mathbb{P}\ V \times \mathbb{P}\ E) \rightarrow (V \times V \rightarrow \mathbb{N}) \\
grrep : (V \times V \rightarrow \mathbb{N}) \rightarrow (\mathbb{P}\ V \times \mathbb{P}\ E) \\
\hline
\forall\, v_i, v_j : V \bullet \forall\, g : \mathbb{P}\ V \times \mathbb{P}\ E \bullet \\
\quad (v_i \in grvertices(g) \land v_j \in grvertices(g) \land \\
\qquad \Omega(v_i, v_j) = 1 \Leftrightarrow (v_i, v_j) \in gredges(g) \land \\
\qquad \Omega(v_i, v_j) = 0 \Leftrightarrow (v_i, v_j) \notin gredges(g)) \Rightarrow \\
\qquad\quad (adjmtx(g) = \Omega \land grrep(\Omega) = g)
\end{array}
$$

$$
\begin{array}{|l}
\_\ mult\ \_ : (V \times V \rightarrow \mathbb{N}) \times (V \times V \rightarrow \mathbb{N}) \rightarrow (V \times V \rightarrow \mathbb{N}) \\
\hline
\forall\, a, b, c : V \times V \rightarrow \mathbb{N} \bullet \\
\quad a\ mult\ b = c \Leftrightarrow \\
\qquad (\forall\, v_i, v_j : V \bullet c(v_i, v_j) = (\forall\, v_k : V \bullet \sum a(v_i, v_k) * b(v_k, v_j)))
\end{array}
$$

A matrix multiplication operator *mult* is also defined. We use Ω^r to represent a matrix, where each element (i.e., $\Omega^r(v_i, v_j)$) of the matrix represents the number of different paths of length r from vertex v_i to v_j. Ω^r is obtained from the matrix multiplication of r matrices.

$$
\begin{array}{l}
\Omega^1 == \Omega \\
\Omega^r == \Omega\ mult\ \Omega^{r-1}
\end{array}
$$

$$
\begin{array}{l}
\forall\, v_i, v_j : V \bullet \Omega^r(v_i, v_j) = \\
\quad \#\{\ p : P \mid v_i \in grvertices(grrep(\Omega^r)) \land \\
\qquad v_j \in grvertices(grrep(\Omega^r)) \land \\
\qquad p \in grpaths(grrep(\Omega^r)) \land \\
\qquad p(1) = v_i \land p(\#p) = v_j\ \}
\end{array}
$$

In the following, there are two global functions, *rlength* and *rrepeat*, that compute *rregion* of schema *LevelRepeatedInheritance*. Function *rlength* is a set of inheritance paths of length r from vertex v_i to vertex v_j. Function *rrepeat* is a set of repeated inheritances obtained by the union of different paths in *rlength*. Using functins, *rlength* and *rrepeat*, we define schema *LevelRepeatedInheritance*, which find out all repeated inheritances at the r level.

$$
\begin{array}{|l}
rlength : \mathbb{N}_1 \times V \times V \rightarrow \mathbb{P}\ P \\
\hline
\forall\, r : \mathbb{N}_1 \bullet \forall\, v_i, v_j : V \bullet \\
rlength(r, v_i, v_j) = \{\ s : P \mid s(1) = v_i \land s(\#s) = v_j \land r = \#s - 1\ \}
\end{array}
$$

The following is an abbreviation definition for repeated inheritnaces:

$$
\begin{array}{l}
RI == \mathbb{P}\ P \\
\\
\forall\, r_i : RI \bullet \exists\, s_1, s_2 : P \bullet s_1 \in r_i \land s_2 \in r_i \Rightarrow \\
\qquad s_1(1) = s_2(1) \land s_1(\#s_1) = s_2(\#s_2)
\end{array}
$$

$$
\begin{array}{|l}
rrepeat : \mathbb{N}_1 \times V \times V \rightarrow RI \\
ri : RI \\
\hline
\forall\, r : \mathbb{N}_1 \bullet \forall\, v_i, v_j : V \bullet \\
rrepeat(r, v_i, v_j) = \{\ s : P \mid \exists\, r_i : RI \bullet s \in rlength(r, v_i, v_j) \land s \in r_i\ \}
\end{array}
$$

LevelRepeatedInheritnace

$\Xi InheritanceGraph$
$r : \mathbf{N}_1$
$n : \mathbf{N}_1$
$v_i, v_j : V$
$rregion : \mathbf{N}_1 \times V \times V \rightarrow RI$

$\forall v_i, v_j : V \bullet \forall r : \mathbf{N}_1 \bullet r = 2 \bullet$
$\quad rregion(r, v_i, v_j) = rrepeat(r, v_i, v_j) \vee$
$\quad rregion(r, v_i, v_j) = (rrepeat(r, v_i, v_j) \cup rlength(r-1, v_i, v_j)) \vee$
$\quad rregion(r, v_i, v_j) = (rlength(r, v_i, v_j) \cup rlength(r-1, v_i, v_j))$

$\forall v_i, v_j : V \bullet \forall r : \mathbf{N}_1 \bullet 2 < r \leq n \bullet$
$\quad rregion(r, v_i, v_j) = rrepeat(r, v_i, v_j) \vee$
$\quad rregion(r, v_i, v_j) = \bigcup_{p=2}^{r-1}(rrepeat(r, v_i, v_j) \cup rregion(p, v_i, v_j)) \vee$
$\quad rregion(r, v_i, v_j) = \bigcup_{p=1}^{r-1}(rrepeat(r, v_i, v_j) \cup rlength(p, v_i, v_j)) \vee$
$\quad rregion(r, v_i, v_j) = \bigcup_{p=2}^{r-1}(rlength(r, v_i, v_j) \cup rregion(p, v_i, v_j)) \vee$
$\quad rregion(r, v_i, v_j) = \bigcup_{p=1}^{r-1}(rlength(r, v_i, v_j) \cup rlength(p, v_i, v_j))$

A global function, *finduriset*, computes *uriset* of schema *URI*. If there exists two different inheritance paths *set*1 and *set*2, function *finduriset* calls *makegraph* and *extendgraph* to construct a URI that must satisfy the definitions of a URI. A function *nofuri* is to count the number of elements in the set of URIs computed by *finduriset*.

$finduriset : \mathbf{P}\ P \rightarrow \mathbf{P}(\mathbf{P}\ V \times \mathbf{P}\ E)$
$ps : \mathbf{P}\ P$

$\forall ri : \mathbf{P}\ V \times \mathbf{P}\ E \bullet \exists set1, set2 : P \bullet$
$\quad finduriset(ps) = \{\ uri : \mathbf{P}\ V \times \mathbf{P}\ E \mid (uri = extendgraph(set1, makegraph(set2))) \wedge$
$\qquad (ps = grpaths(ri)) \wedge$
$\qquad (set1, set2 \in ps) \wedge$
$\qquad (set1(\#set1) = set2(\#set2)) \wedge$
$\qquad (\#fcn(ps) - \#fce(ps) = 2)\ \}$

$nofuri : \mathbf{P}\ P \rightarrow \mathbf{N}$

$\forall ps : \mathbf{P}\ P \bullet nofuri(ps) = \#finduriset(ps)$

Using function *finduriset*, we define schema *Decompose*, which inherits variables from schema *URI* (and thus schema *RepeatedInheritanceGraph*) and constructs a set of URIs from an inheritance graph.

Decompose

ΞURI

$\forall i : \mathbf{N}_1 \bullet \forall \Omega^i : V \times V \rightarrow \mathbf{N} \bullet$
$\quad \forall ps : \mathbf{P}\ P \bullet \forall p : P \bullet$
$\qquad p \in ps \Rightarrow p \in cipaths \wedge$
$\qquad \Omega^i = adjmtx(graph) \wedge$
$\qquad uriset = finduriset(ps)$

The following is an algorithm, called *Finding_URIs_Algorithm*, to decompose an inheritance graph into a set of URIs. Besides, the algorithm is proved to be *sound* and *complete*. The proof is described as follows.

Definition 3 *Let G=(V,E) be an inheritance graph.*

1. *A root class is a class node with no in-edges in G.*
2. *A terminal class is a class node with no out-edges in G.*

3. *Ancestor(v) is a set which records all of the different inheritance paths from vertices in roots set to v, where v is a vertex in V.*

Algorithm:

Finding *Unit Repeated Inheritances* (URIs);
Let CN denote the number of common nodes and CE be the number of common edges;
Input: A directed graph $G = (V, E)$;
Output: A set of Unit Repeated Inheritances;
Step 1: Build a directed graph consisting of classes and inheritance edges
and initialize Ancestor(v)={ v }, for all vertices in V;
Step 2: Going *breadth-first*, traverse for all root classes;
in the process of traversal, every element of Ancestor(v_i) is added into
Ancestor(v_j), v_i is a parent class of v_j;
Step 3: For all terminal classes do
if the number of the ancestor set ≥ 2 **then**
begin
Union ($set1, set2$) $\{set1 \neq set2\}$;
if there exist common parents, record the union set **then**
if $CN - CE = 2$ **then**
A *unit repeated inheritance* is found;
else no URI exists;
endif;
else no repeated inheritances exist, return an empty set;
endif;

In the algorithm, we utilize the *breadth-first search* to traverse all classes in order to obtain the *Ancestor* set of each class. In the process of traversal, all ancestor classes of each class is added into the *Ancestor* set of the class. Then, we check whether each terminal class has an *Ancestor* set of cardinality greater than 2. If the *Ancestor* set of a terminal class contains two or more elements, we unite any two elements in the *Ancestor* set. Next, if every union set has common parents and the number of nodes of the union set is equal to the number of edges, a URI is found; otherwise, we discard the union set because it cannot satisfy the property of a URI. And, if the number of elements of an *Ancestor* set of the terminal class is less than 2, there are no repeated inheritances, and the algorithm returns an empty set. In the following subsection, we prove that the algorithm is *sound* and *complete*.

3.1 Soundness and Completeness of the Algorithm

Lemma 2 (**Soundness of** the Finding_URIs_Algorithm). *The repeated inheritances generated from the inheritance graph G by the Finding_URIs_Algorithm must be URIs.*

Proof: By *breadth-first traversal search* (BFS), we can associate BFS numbers with nodes. That is, a vertex has a BFS number l if it was the l*th* vertex to be visited by the BFS.

Assuming that there are two arbitrary complete inheritance paths, $set1$ and $set2$, in *Ancestor*(v), where v is one of the terminal classes in the inheritance graph, each has m and n nodes and thus $m-1$ and $n-1$ edges, respectively. We unite the two paths. If the union set satisfies $CN - CE = 2$, then,

Total number of nodes $= m + n - CN$.

Total number of edges $= (m-1) + (n-1) - CE = m + n - 2 - (CN - 2) = m + n - CN$.

We find that the number of nodes is equal to the number of edges, thus the repeated inheritance generated from the algorithm is a URI. Thus, the algorithm is sound. □

Lemma 3 (**Completeness of** the Finding_URIs_Algorithm). *If there exists a URI in the inheritance graph G, the URI must be in the set of URIs obtained by the Finding_URIs_Algorithm.*

Proof:

1. Due to the property of repeated inheritance, there must exist two or more different inheritance paths in the repeated inheritance. Therefore, if there is only one element in the *Ancestor* set of a terminal class, we cannot get any repeated inheritances. So, the "**if** the number of the ancestor set ≥ 2" is a necessary condition.

2. If there exist no commom parents of any two paths, that is, the only common node in the union set is the terminal class, we have $CN = 1$ and $CE = 0$. Thus,
   Total number of nodes $= m + n - 1$.
   Total number of edges $= (m-1) + (n-1) - 0 = m + n - 2$.
   This does not satisfy the property of URIs. We can conclude that if there exists no common parent of two paths, there is no URI.
   Therefore, the "**if** there exist common parents" in the algorithm is a necessary condition.

3. According to the property of URIs, it must be true that $CN - CE = 2$ for each repeated inheritance. That is, we can deduce from $CN - CE = 2$ that the number of common nodes is equal to the number of common edges within the repeated inheritance. Therefore, the "**if** $CN - CE = 2$" is a necessary condition.

As the above discussion, the three "**if**" expressions in the algorithm are all necessary conditions. If there exists a URI in the repeated inheritance, it must be selected by the algorithm. Therefore, the algorithm is complete. □

4 Inheritance Level Technique

The Inheritance Level Technique method is developed based on URIs. In spite of the completeness of URI testing, we cannot ensure the correctness of an inheritance graph because many errors exist in different inheritance paths instead of in unit repeated inheritance. When we test these classes in the inheritance paths, the topological order should be kept. Since inheritance satisfies the transitive property, the errors in ancestor classes would propagate to descendant classes naturally. In other words, we have to test those superclasses before their subclasses. This testing hierarchy is divided into n levels, denoted by ILT(0), ILT(1), $\cdots$, ILT(n), where n is the length of the longest inheritance path in an inheritance graph plus one. The higher the value of n is, the more complex and error prone the inheritance hierarchy will be. As a consequence, the n level of testing hierarchies can be defined as the following:

- ILT(0): Every class in an inheritance graph needs to be tested at once.
- ILT(1): Every set of two related classes (that is, inheritance path of length = 1) needs to be tested at least once.
- ILT(2): Every set of three related classes (that is, inheritance path of length 2) and all the repeated inheritances at this level need to be tested at least once.

  ⋮

- ILT(n): This testing level represents that the ILT hierarchy is complete.

Now, we explain how to use these inheritance levels to test all inheritance paths in an inheritance hierarchy. First, we use the breadth-first traversal algorithm to traverse all root classes (without in-edges, denoted by a set **Root**). Then we construct an adjacent matrix corresponding to the inheritance hierarchy

and check all nonzero entries to assure the correctness of relationships between classes. ILT(i) is based on the idea of *rregion*(i) which is derived from the result of $\bigcup_{p=2}^{i-1} ILT(p)$. The testing hierarchy prototypes represented by ILT(n) are shown as the following:

- ILT(0): This is the object testing of object-oriented software. The sequences of objects to be tested should satisfy some specific order. Since inheritance possesses the property of transitivity, the errors of parent objects should be tested in advance. Interestingly, the sequence of objects to be tested is in a topological order. Using a *topological ordering* algorithm, we get the following precedence ordering sequence (see Figure 1). According to this sequence, ILT(0) would be exercised correctly.

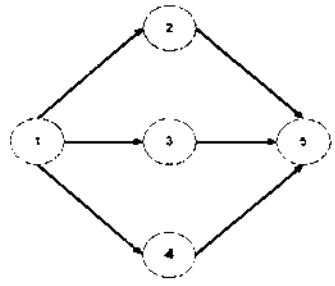


Figure 1: The ILT(0) Testing Level

precedence ordering sequences= { (1,2,3,4,5), (1,2,4,3,5), (1,3,2,4,5), (1,3,4,2,5), (1,4,2,3,5), (1,4,3,2,5) }

- ILT(1): Given an inheritance hierarchy Ω with n classes $c_1, c_2, \cdots, c_n$, it corresponds to an adjacent matrix which represents the inheritance relations among the n classes $c_1, c_2, \cdots, c_n$ as the following:

$$\Omega = [m_{ij}]_{n \times n} \begin{cases} m_{ij} = 1, & \text{if there exists a directed edge from } v_i \text{ to } v_j \text{ and } i \neq j. \\ m_{ij} = 0, & \text{otherwise.} \end{cases}$$

Every $a_{i,j}$=1 needs to be tested at least once since correctness of object testing could not assure the "is-a" relationship is correct.

- ILT(2): All the inheritance paths of length 2 should be tested at least once. Ω^2 denotes $\Omega \times \Omega$. The entry of Ω^2 is defined as the following: $a_{ij} = k$ if and only if there are k different paths of length 2 from class a_i to class a_j, otherwise $a_{ij} = 0$.

$$\Omega^2 = [a_{ij}]_{n \times n} \begin{cases} a_{ij} = k, & \text{if there are k different paths of length 2.} \\ a_{ij} = 0, & \text{otherwise.} \end{cases}$$

Every a_{ij} needs to be tested at least once such that we can test all relationships between three classes. If $a_{ij} = k > 1$, we unite two, three, four, $\cdots$, and k from these k different paths to form some repeated inheritances, where $i \in Root$. Besides, if $a_{ij} \neq 0$ and corresponding entry $a_{ij} \neq 0$ in ILT(1), we also unite corresponding paths in ILT(1) and ILT(2) to get another repeated inheritances. To decompose these repeated inheritances into URIs helps us detect all name conflicts at this level.

$\vdots$

- ILT(i): All the inheritance paths of length i should be tested at least once. These paths with length i are kept in Ω^i and defined as the following:

$$\Omega^i = [a_{ij}]_{n \times n} \begin{cases} a_{ij} = k, & \text{if there are k different paths of length i.} \\ a_{ij} = 0, & \text{otherwise.} \end{cases}$$

Every $a_{ij} \neq 0$ needs to be tested at least once such that we can test all relationships between i classes. If $rregion(i,i,j)$ is not empty, then we can test all repeated inheritances in $rregion(i,i,j)$ to help us detect all name conflicts at this level.

$\vdots$

- ILT(n): Given a matrix $\Omega^n(\Omega^{n-1} * \Omega)$, if all elements are zeros in $\Omega^n[a_{i,j}]_{n*n}$, the ILT testing method is complete.

4.1 An Example

It is easier to see a good example to understand how an inheritance graph is decomposed into a set of repeated inheritances and URIs. Consider a word processor application program using a number of abstract data types implemented in different classes. The program needs to maintain a list of words as user documentation. Also, a dictionary is used for spelling checking. The dictionary contains a user dictionary implemented in a binary tree and manipulated by the user for inserting / deleting special words. The word processor application program is also implemented in a complicated data type such as an AVL-tree. The fundamental data types used are static arrays and dynamic memory allocation. Figure 2 shows an inheritance graph facilitating the design and implementation of this word processor application program.

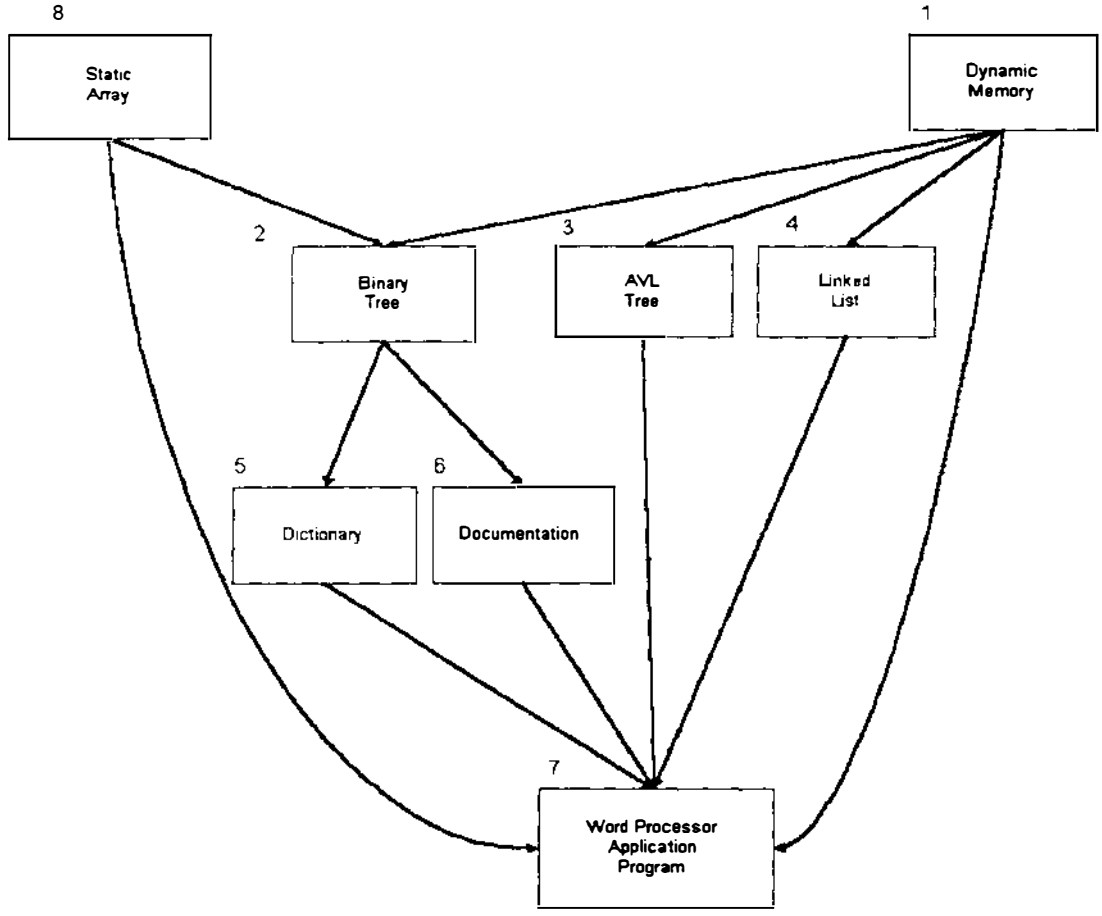


Figure 2: The inheritance graph of abstract data types used in a word processor application program.

To ease our discussion, each data type is attached to a node number ranges from 1 to 8.

After the *breadth-first* traversal, we find that there are two root classes 1 and 8.

1. **ILT(1)** encounters an adjacent matrix with respect to Figure 2 as the following:

$$\Omega^1 = \begin{array}{c} \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \end{array} \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}_{8\times 8}$$

Since there doesn't exist any path with a length greater than 2, this level cannot contain any repeated inheritances. Therefore, we only get $rlength(1, i, j)$ below, where i and j are vertices:-
$rlength(1,1,2) = \{(1,2)\}$, $rlength(1,1,3) = \{(1,3)\}$, $rlength(1,1,4) = \{(1,4)\}$, $rlength(1,1,7) = \{(1,7)\}$, $rlength(1,8,2) = \{(8,2)\}$, and $rlength(1,8,7) = \{(8,7)\}$.

According to the non-zero entries, existing inheritance relationships between two classes are exercised at least once.

2. **ILT(2)** encounters a matrix obtained by the *1st* transitive closure as the following:

$$\Omega^2 = \begin{array}{c} \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}_{8\times 8}$$

We find that the $a_{1,7}$ entry is 2. According to ILT(2), we have $rrepeat(2,1,7) = \{(1,3,4,7)\}$ and unite $(rlength(2,1,7), rlength(1,1,7))$. Therefore, $rregion(2,1,7)$ contains three repeated inheritances as shown in Figure 3.

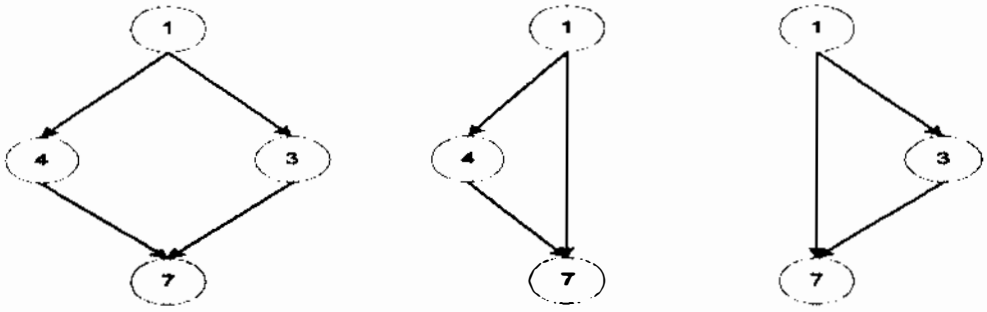


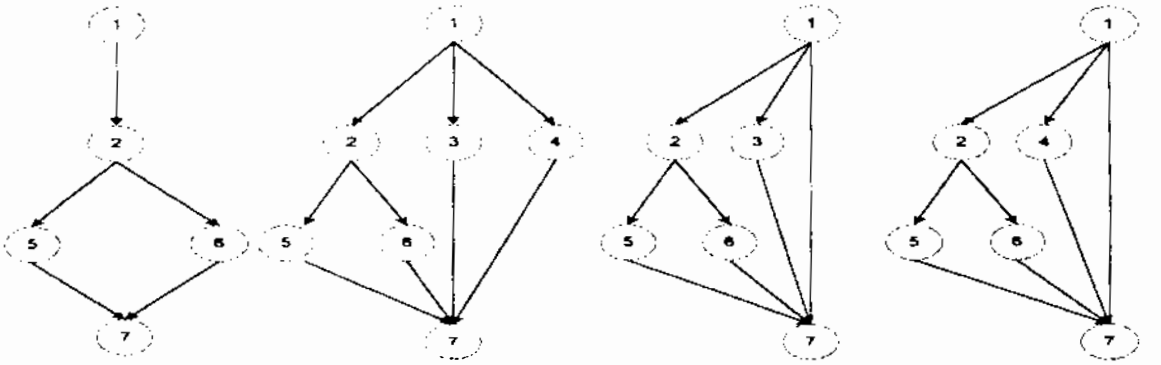
Figure 3. The Repeated Inheritances of ILT(2).

By definition 1 and lemma 1, the three repeated inheritances are also URIs. These three URIs at this level need to be tested at least once.

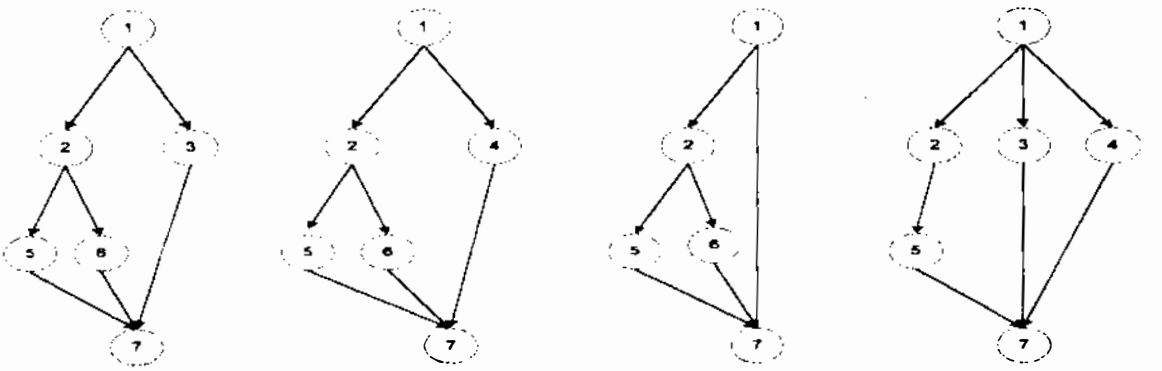
3. **ILT(3)** encounters a matrix obtained by the *2nd* transitive closure as the following:

$$\Omega^3 = \begin{array}{c} \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \end{bmatrix}_{8\times 8}$$

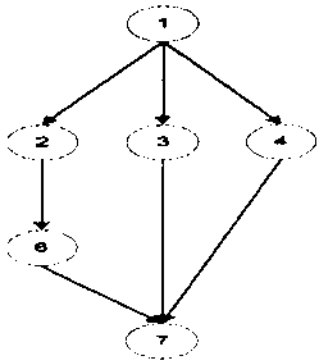
ILT(3) includes the following elements and the repeated inheritances as Figure 4 and 5.
$rrepeat(3,1,7) = \{(1,2,5,6,7)\}$, $rlength(3,1,7) = \{(1,2,5,7),(1,2,6,7)\}$.
$rregion(3,1,7) = \{(1,2,5,6,7),(1,2,3,4,5,6,7),(1,2,3,5,6,7),(1,2,4,5,6,7),(1,2,3,5,6,7),$
$(1,2,4,5,6,7),(1,2,5,6,7),(1,2,3,4,5,7),(1,2,3,4,6,7),(1,2,3,5,7),$
$(1,2,4,5,7),(1,2,3,6,7),(1,2,4,6,7),(1,2,3,5,7),(1,2,4,5,7),(1,2,5,7),$
$(1,2,3,6,7),(1,2,4,6,7),(1,2,6,7)\}$.



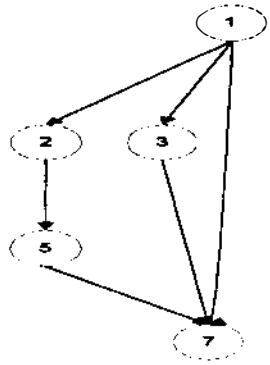
1. $rrepeat(3,1,7)$ 2. $rrepeat(3,1,7) \cup rregion(2,1,7)$ 3. $rrepeat(3,1,7) \cup rregion(2,1,7)$ 4. $rrepeat(3,1,7) \cup rregion(2,1,7)$



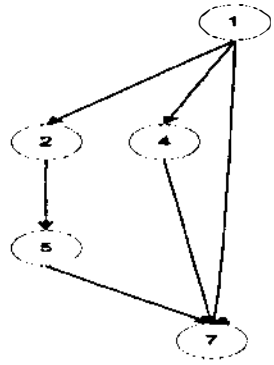
5. $rrepeat(3,1,7) \cup rlength(2,1,7)$ 6. $rrepeat(3,1,7) \cup rlength(2,1,7)$ 7. $rrepeat(3,1,7) \cup rlength(1,1,7)$ 8. $rlength(3,1,7) \cup rregion(2,1,7)$



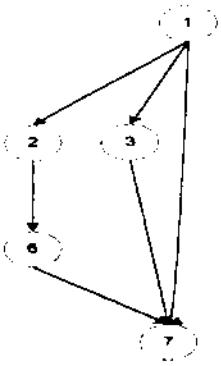
9. $rlength(3,1,7) \cup rregion(2,1,7)$



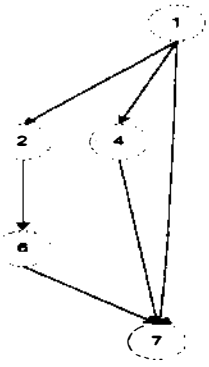
10. $rlength(3,1,7) \cup rregion(2,1,7)$



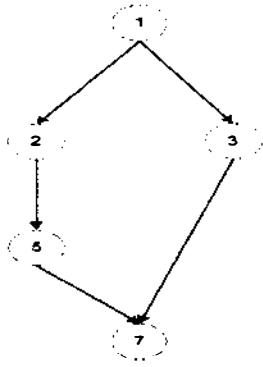
11. $rlength(3,1,7) \cup rregion(2,1,7)$



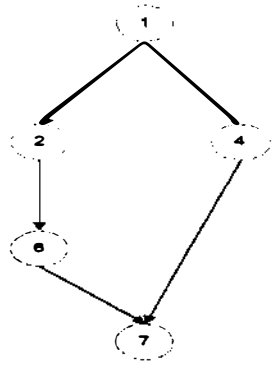
12. $rlength(3,1,7) \cup rregion(2,1,7)$



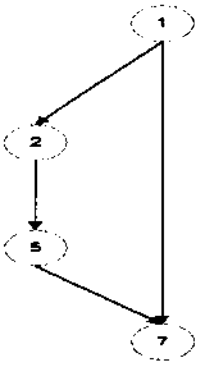
13. $rlength(3,1,7) \cup rregion(2,1,7)$



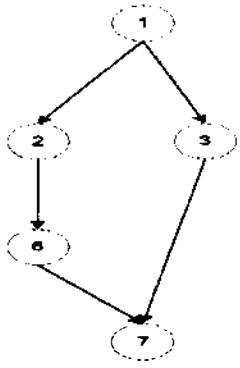
14. $rlength(3,1,7) \cup rlength(2,1,7)$



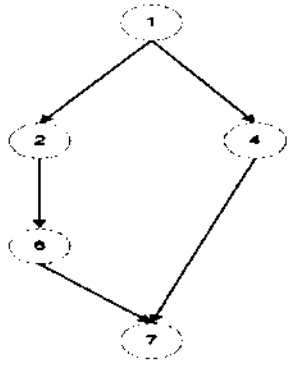
15. $rlength(3,1,7) \cup rlength(2,1,7)$



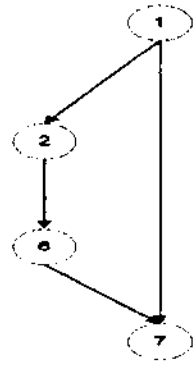
16. $rlength(3,1,7) \cup rlength(1,1,7)$



17. $rlength(3,1,7) \cup rlength(2,1,7)$



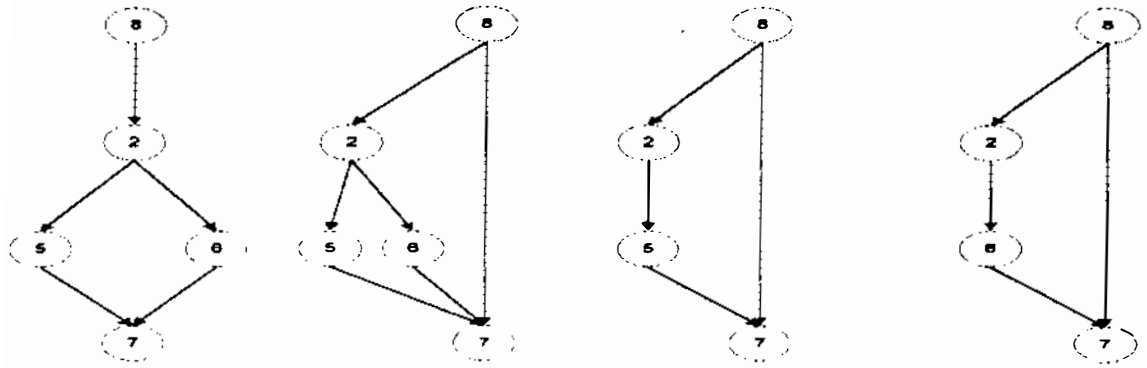
18. $rlength(3,1,7) \cup rlength(2,1,7)$



19. $rlength(3,1,7) \cup rlength(1,1,7)$

Figure 4. The Repeated Inheritances of ILT(3) with respect to **root** 1.

$rrepeat(3,8,7) = \{(8,2,5,6,7)\}$, $rlength(3,8,7) = \{(8,2,5,7),(8,2,6,7)\}$.
$rregion(3,8,7) = \{(8,2,5,6,7),(8,2,5,6,7),(8,2,5,7),(8,2,6,7)\}$.



1. $rrepeat(3,8,7)$ 2. $rrepeat(3,8,7) \cup rlength(1,8,7)$ 3. $rlength(3,8,7) \cup rlength(1,8,7)$ 4. $rlength(3,8,7) \cup rlength(1,8,7)$

Figure 5. The Repeated Inheritances of ILT(3) with respect to **root** 8.

$rregion(3,1,7)$ and $rregion(3,8,7)$ contain nineteen and four repeated inheritances, respectively. By theorem 1, those repeated inheritances can be decomposed into a set of URIs. Each URI at this level needs to be tested at least once.

4. **ILT(4)** encounters a matrix obtained by the *3rd* transitive closure as the following:

$$\Omega^4 = \begin{array}{c} \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{8\times 8}$$

Now, we find that Ω^4 is a zero matrix, so the testing hierarchy is complete.

5 ILT Testing Criteria

A model of inheritance graph was discussed in schema *InheritanceGraph*. Based on this schema, we then derived schema *RepeatedInheritanceGraph* by restricting the graph to have common nodes and using the notion of complete inheritance paths. An important property of a URI is proved and a function based on the property is given to decompose a repeated inheritance graph into a set of URIs. This decomposition mechanism can be used in software metrics and testing. For instance, one can count the order of a URI set from an inheritance graph to measure the complexity of the graph. Nevertheless, based on this decomposition mechanism, we have developed some testing criteria. The first criterion is to iterate the decomposition by the number of paths in each repeated inheritance graph. The second criterion is to iterate the decomposition by the length of complete inheritance paths. The following pseudo code can be easily used in building a software testing tool:

- *Intra level first criterion:* This testing criterion is based on the number of URIs. For a class hierarchy with euler's closed region equal to n, the testing steps could be divided into n levels vary from 2 to

n, i.e., ILT(2), ILT(3), ···, ILT(n). In other words, URIs in ILT(i) are exercised according to the descending complexity order of URIs, and URIs in ILT(i+1) cannot be exercised until all URIs in ILT(i) are tested. For example, for some fixed *i*, we choose the most complex URI in ILT(i) to be tested. As a fact, complexity is proportional to the amount of hidden errors.

- *Inter level first criterion:* This testing criterion is based on the lengths of inheritance path in ILT matrix. For a class hierarchy with the length of inheritance paths equal to n, then the testing steps could be divided into *n* levels vary from 2 to n, i.e., ILT(2), ILT(3), ···, ILT(n). In other words, URIs in ILT(i) are exercised according to the descending complexity order of URIs. For all groups from ILT(2) to ILT(n), we select one URI contained in ILT(i) to be exercised during the *i*th iteration until the ILT(i) is empty.

Two algorithms supporting the two testing criteria are presented as follows:

Intra Level First Algorithm:

```
Do i:=2 to n
  begin
    repeat
      in ILT(i)
      select the repeated inheritance R with highest URIs complexity;
      conduct test;
      remove R from ILT(i);
    until ILT(i) is empty;
  end;
```

Inter Level First Algorithm:

```
repeat
  Do i:= 2 to n
    begin
      in ILT(i)
      select the repeated inheritance R with highest URIs complexity;
      conduct test;
      remove R from ILT(i);
    end;
until ILT(2), ···, ILT(n) is empty;
```

We define that a repeated inheritance with the highest URI complexity is the one which contains the largest number of URIs. Note that the more URIs in an inheritance graph, the more complex and error prone it will be. That is, by testing a higher number of URIs first, we might be able to detect inheritance hierarchy errors earlier. In addition, based on the tested URIs and the tested results, a tester is in a better situation of choosing the next URIs to be tested than one based on URIs being selected randomly. This testing hierarchy provides users a good framework for selecting a proper testing correctness level in an inheritance hierarchy.

6 Conclusions

Our contributions are, first, the issues of repeated inheritance object hierarchy were carefully studied and a model used the *Z* notation was proposed. Based on this model, we introduced an important method named URI as a basic unit of object-oriented testing and then developed an algorithm to decompose a repeated inheritance graph into a number of unit repeated inheritances (URIs). We also proposed the ILT method

to illustrate the processes of the testing of an inheritance hierarchy. This method can be easily used in software metrics and testing of an object-oriented analysis/design paradigm. And the mechanism can be easily implemented following our pseudocode.

References

[1] Chung, C. M., and Lee, M. C., "Object-Oriented Programming Testing Methodology," *International Journal of Mini & MicroComputers,* Vol. 16, No. 2, 1994, pp. 73–81.

[2] Jacobson, I., "Object-Oriented Software Engineering: A Use Case Driven Approach," Addison-Wesley, 1992.

[3] Laski, J. W. and Korel, B.,"A data flow oriented programming testing strategy," *IEEE Trans. on Software Eng.,* Vol. 9, No. 3, May 1983.

[4] Lorenz, M., "Object-Oriented Software Development: A Practical Guide," Prentice Hall, 1993.

[5] McCabe, T. J., "Design Complexity Measurement and Testing," *CACM,* Vol. 32, December 1989, pp. 1415-1425.

[6] Meyer, B., "Object-Oriented Software Construction," Prentice Hall, 1988.

[7] Ntafos, S. C., "On Required Element Testing," *IEEE Trans. on Software Eng.,* Vol. SE-10, No. 6, November 1984, pp. 795-803.

[8] Rapps, S. and Weyuker, E. J., "Selection software test data using data flow information," *IEEE Trans. on Software Eng.,* Vol. SE-11, No. 4, April 1985, pp. 367-375.

[9] Seidewitz, E. and Stark, M., "Towards a general object-oriented software development methodology," *Ada Letters,* July/August 1987, pp. 54-67.

[10] Perry, D. E. and Kaiser, E. G., "Adequate Testing and Object-Oriented Programming," *JOOP,* 2(5), 1990, pp.13-19.

[11] Smith, M.D. and Robson, D.J., "A framework for testing object-oriented programs," *JOOP,* June 1992, pp. 45-63.

[12] Harrold M., McGregor John D., and Fitzpatrick Kevin, "Incremental testing of object-oriented class structure," *Proceedings of the 14th International Conference on Software Engineering,* 1992, pp. 68-80.

[13] Kirani S. and Tsai W.T., "Method sequence specification and verification of class," *JOOP,* 1994, pp. 28-38.

[14] Spivey, J. Michael. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

[15] Spivey, J. Michael. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, 1989.

[16] Spivey, J. Michael. "A Guide to the **Zed** Style Option". Technical report, Oxford University Computing Laboratory, 1990.

[17] Berge C., "Graph and Phypergraphs," (Amsterdam: North-Holland), 1973.

Using A

Stream-Based Object Oriented Interpreter

To Manage Complex Automated Tasks

Keywords : Automation, Stream-Based Engine

Address : 2410 137th PL SE Mill Creek WA 98012

Email Address : a-jimhe@microsoft.com

Brief Biography Of Author(s) :

- **James Thomas Heuring** : Began programming as a software engineer in 1985 and worked up to a lead engineering position on a blood chemistry analyzer. Later went on to write software for avionic subsystems (engine controller, air inlet control system, central air data computer). In both of these environments the software must work 100% of the time. Came to Microsoft as a contractor over 2 years ago to help test new mail client (Exchange). Developed the stream-based engine used to test Microsoft Exchange. Will soon be absorbed into a full time automation team. Has been responsible for design and development of the system as well as integrating the engine with other Microsoft tools.
- **Robbie Paplin** : Robbie came to Microsoft in 1992 to work on Microsoft Project as a Software Test Engineer. He is currently working in the Exchange Product unit's client test team. During the last product cycle he was one of the main advocates of stream based testing tools and automation. He has developed several tools to help the test team take advantage of these technologies and eliminate some of the more tedious aspects of testing.

Technical Overview

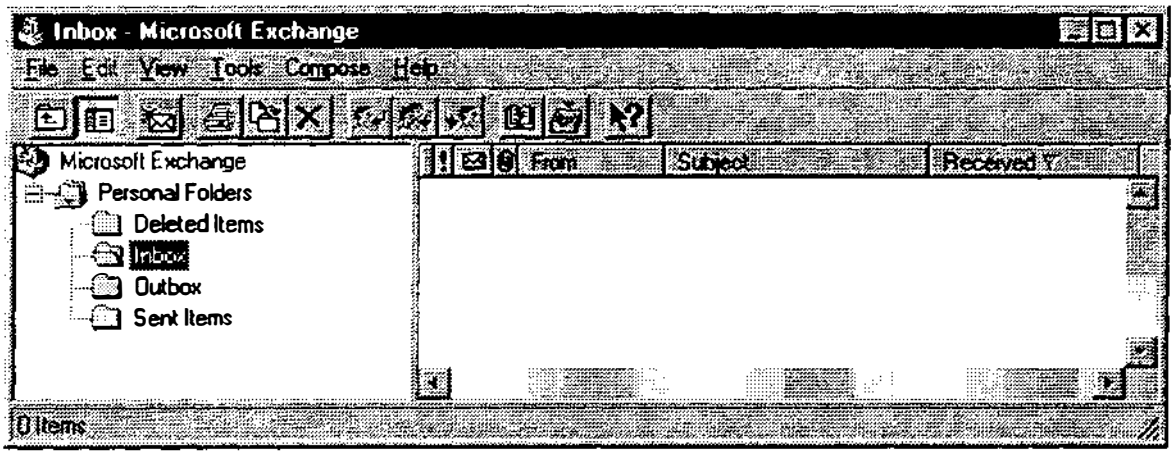
Purpose : To develop the simplest and most efficient method of creating automated tasks. Tasks can be used as building blocks for more complex automation.

How this is achieved :

- Simple pre-defined object definitions allow users to quickly create and perform complex tests. Each object definition consists of data elements and functions.
- Data for how tests are performed is read in from streams, usually text files. Objects and Directives are a fed into an interpreter which performs the operations based on the properties of an object and the functions performed against the object.

Motivation : In the following example we are going to establish need to test a high number of possible scenarios and give an examples of how an stream-based object-oriented interpreter is used to achieved this objective. The Microsoft Exchange client is used as the example of the application being testing.

Pictured below is the Microsoft Exchange client.



There are 2 panes for the client : the left-hand pane has the folder tree and the right hand pane contains the folder contents. Below the root (Microsoft Exchange) is the store (Personal Folders). Below the store are the folders. At the store level we can create any number of nested folders. To manipulate (add, delete, rename...) the folders we have implemented the **Folder** object. Consider the following object definition, **Folder** and its data elements used in the following example.

```
Folder::CreateObject
Folder::StoreName= Personal Folders
Folder::FolderName=Folder1
Folder::Create
```

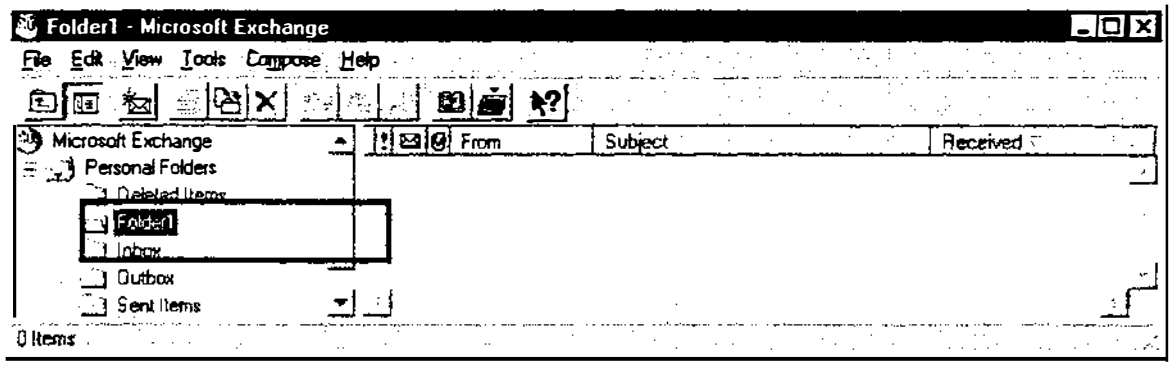
Objects have three components :

- **Constructor and destructors** for the Object. In this example we have **Folder::CreateObject**. This statement is used to create the object in memory against which we will perform operations.
- **Data members :** in this example we have data members **StoreName** and **FolderName**.
- **Function members** which are used to perform operations against the object. In this example we have **Folder::Create**.

The interpreter reads in a line at a time. Therefore as the previous snippet is interpreted the following actions are performed.

Line Interpreted	Action
Folder::CreateObject	Create **Folder** object in memory
Folder::StoreName= Personal Folders	Set the **StoreName** to "Personal Folders"
Folder::FolderName=Folder1	Set the **FolderName** to "Folder1"
Folder::Create	Create the folder

Shown below you can see that **Folder1** has been created in the **Personal Folders** store.

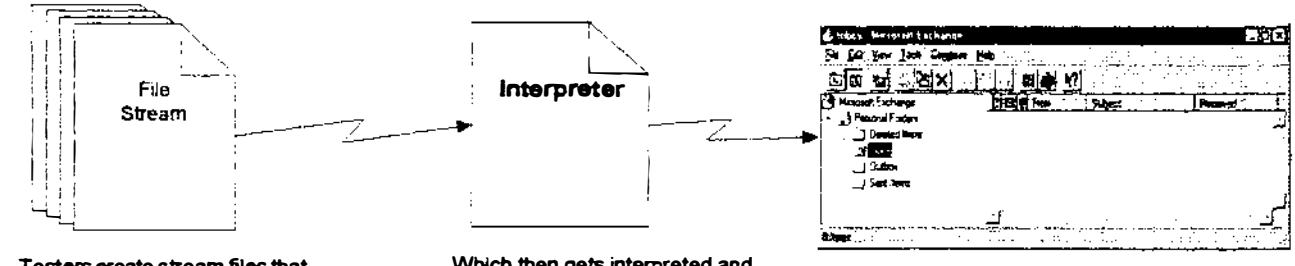


Advantages of Objects :

- Because all necessary foldering functionally is the **Folder** object there is only one piece of code which needs to be updated as the product and testing requirements change.
- By using objects to perform operations you provide layer of abstraction between your test cases and the code that implements them. By using this approach the person using the interpreter does not have to write the mass of code to perform the operations. This shortens the development cycle for the automation scenarios.

How the stream data is read :

The interpreter reads one line at a time. This line of data can be read from a text file or any source which you implement it to read from and this is why this is a stream-based interpreter. Currently, the interpreter is written to interpret text file streams. File streams are simply text files which contain the stream.



Testers create stream files that test areas of the product or set-up time consuming scenarios

Which then gets interpreted and executed by the interpreter

Changes are made to the target system

Streams Allow Easy Access To Functionality
The interpreter(Engine) reads one line at a time. This line of data can be read from a text file or any source which you implement it to read from. Because each line of the stream is text it is easy to create a stream which is simply a human readable text file. Therefore, there is no compilation necessary after the engine has been created.

Definition of the "Engine" : The Engine is your implementation of the interpreter plus the code which the interpreter executes. The distinction here is that the interpreter is simply the code which parses the object definitions. The interpreter itself is a fairly small piece of code which is easy to implement. The current interpreter for Exchange version of the Engine is less than 1000 lines of code. However, the code which supports the engine is over 70 thousand lines including white space and comments.

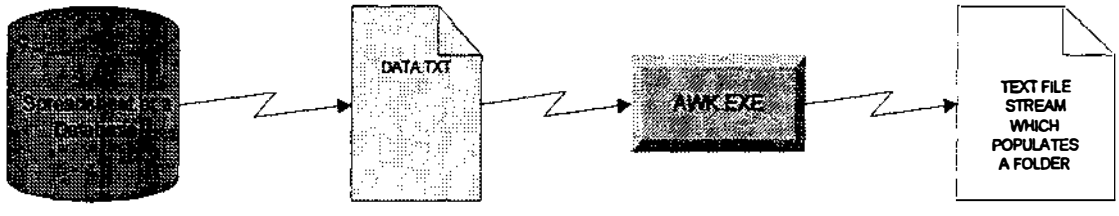
The remainder of the document will deal exclusively with file streams and the support mechanisms which make them effective.

Feeding streams to the Engine :
Because streams are composed of lines of text it is simple to create them using programs and other mechanisms. These mechanisms include :

- Databases and Spread Sheets
- Programs called Permuters which created highly permuted file streams

Databases and Spread Sheets :
In the following diagram we demonstrate the process of populating a folder with messages. The contents of the messages are the movie reviews obtained from a Cinemania CD. As the resultant file stream is read, the contents of the movie reviews are filled into an electronic form which is posted to a folder of movie reviews. This methodology is useful in creating large beds of data to test against. In addition, because of the sheer number of movie reviews, over 22 thousand, we were able to test the long term use of electronic forms.

Populating a Folder



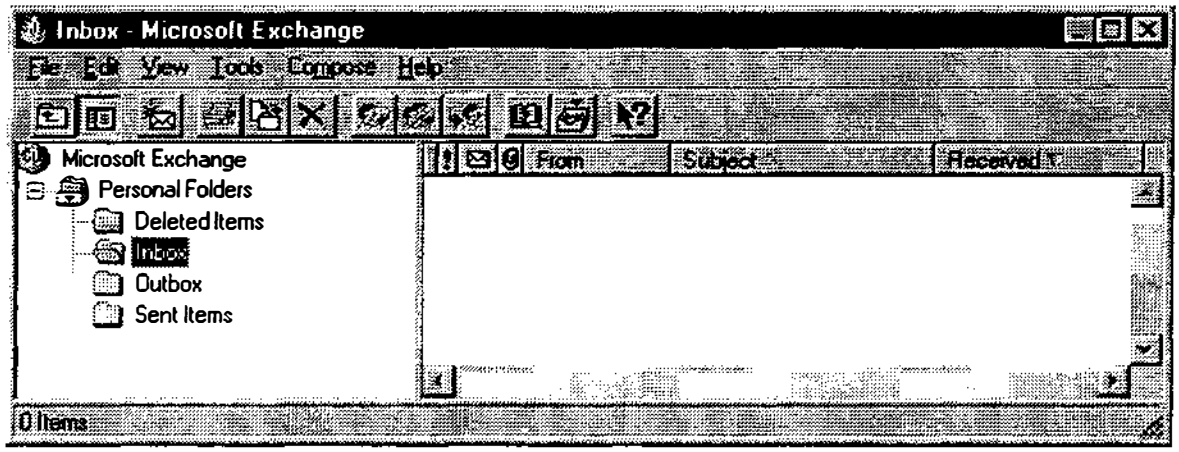
To send custom electronic forms, we start wit a spreadsheet of sample data.

The contents of spreadsheet or database are exported to a text file.

Which gets transformed into a file stream via an AWK script

Programs which created highly permuted file streams ("Permuters"):
Since a stream is composed of objects with member data and functions it is fairly simple to create programs that make iterative changes to the object properties and perform operations against the changed object.

Let us return to our example of using the **Folder** object. In the right-hand pane you can see the column headers.



In Microsoft Exchange you can change the view associated with a folder. Some of the view properties you can change are Columns, Group By, Sort Order and Filter. Consider that there are well over 20 different common data fields which can be used in the Columns, Group By and Sort fields we can see that the number of possible views is actually infinite. However, we can write a permuter which creates a large number folders each with a slightly different views.

To manipulate (add, create, delete, clear...) the views we have implemented the **View** object. Consider the following object definition, **View** and its data elements used in the following example.

```
View::CreateObject
  View::ViewName=Sensitivity: 1
  View::Columns=Sensitivity;Item Type;Attachment;From
  View::Widths=40;40;40;40
  View::GroupBy1=Sensitivity
  View::GroupBySortOrder1=Ascending
  View::GroupBySortOrder4=Ascending
  View::List=Folder Views
  View::GroupSortBy=Conversation Thread
  ' Add the View
  View::Add
```

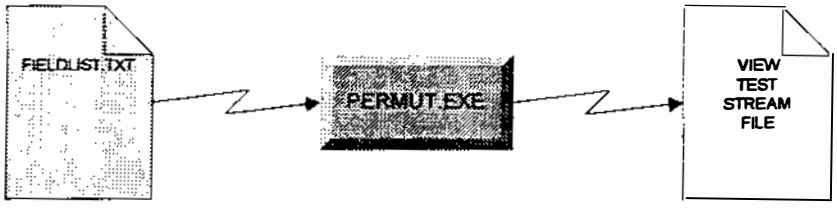
When an object is created the object persists in memory until it is destroyed. The object maintains the properties it has been assigned. Therefore we can add a few extra lines to the previous example and create a view which is slightly different as shown below.

```
  View::ViewName=Sensitivity:2
  View::GroupBySortOrder1=Descending
  View::Add
```

In the 3 lines above we have added another view called **Sensitivity:2** which has a descending not ascending group by sort order.
Note : in the example snippet above we have only added 3 lines to the stream because the other properties have been set by the previous stream.

In the following diagram we show the process of creating a view permuter which created custom views. This permuter takes as its input a list of field names to be permuted. The output is a file stream which creates the custom views.

Testing Views



To test views & custom properties, we make a list of custom properties

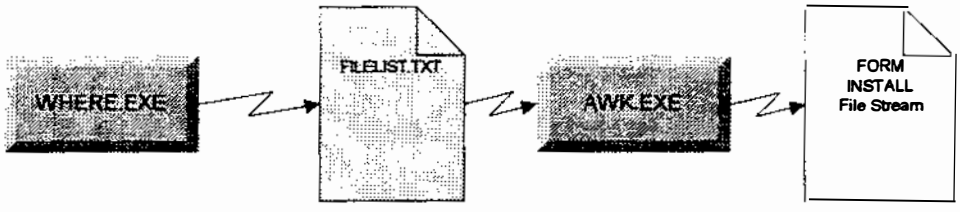
The permuter creates a listing of all possible combinations of those custom properties and creates a file stream.

Using the Engine as a tool in a larger process :

- **Bulk forms installer -** Whenever a new build of the server comes out , usually the stores get erased, so we need a quick way of re-installing all the forms. Also, since the Engine uses the user -interface the install process is also tested.

Shown below is the process of generating the file stream which in turn is used by the engine to install the forms on the server.

Installing forms



To install e-forms, we start w/ a list of all the eforms on the drive.

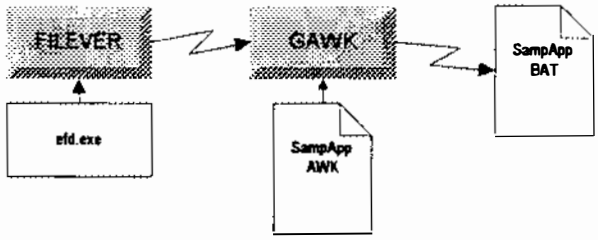
Which gets transformed into a File Stream file via an AWK script

Shown below is the whole process used to regenerate all the sample electronic forms and install them on a server. The last step in this process is the installation of the forms on the server. The engine is used to install the forms on the server and has as its input the file generated earlier in the process.

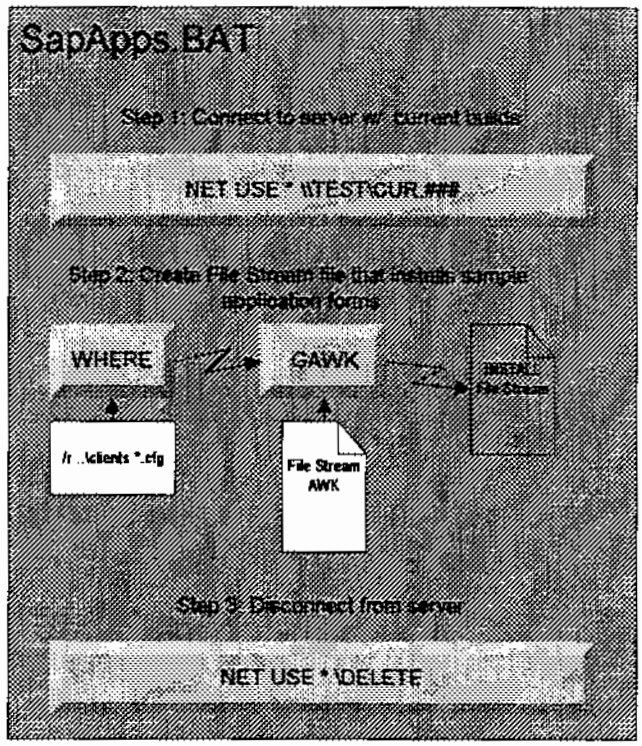
How the new Automated Form Builder works

CODEGEN.BAT

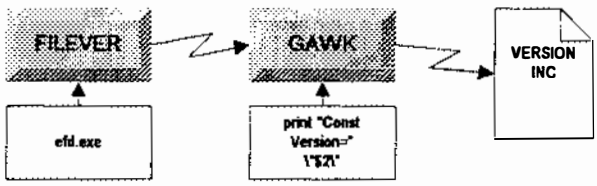
Step 1: Create BAT file that creates sample application install file stream



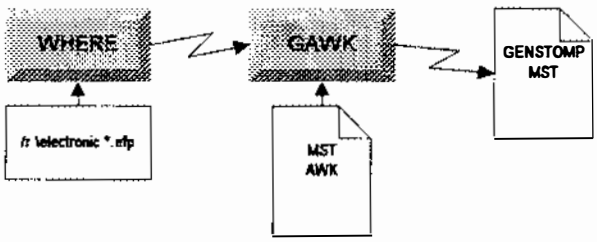
Step 2: Run BAT file that creates sample Application install file stream



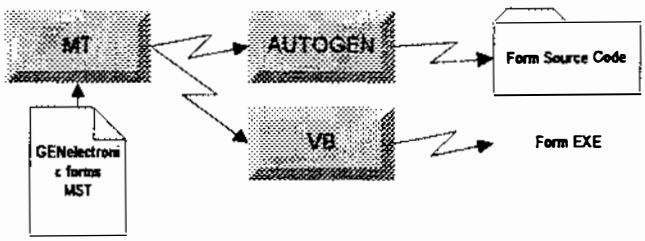
Step 3: Create INC file that contains version #



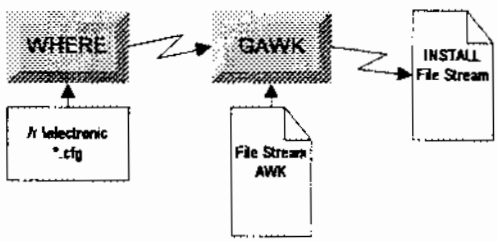
Step 4: Create MST file that codegens & compiles EFP files



Step 5: Run MST file that codegens & compiles EFP files



Step 6: Create file stream file that installs electronic forms



Obviously, the engine plays a part in many of our testing processes. This is because of its ease of use and well publicized set of objects.

Features of the interpreter which are not object-based :
There are two kinds of input the parser reads

- **Formal objects** - these were described above using the **Folder** and **View** object.
- **Directives** - directives could be loosely viewed as any input to the interpreter which is not an object

Currently Supported Directives :

- INCLUDE - You can recursively include file streams.
- CONSTANT - You can define a constant just like other languages.
- NumberOfRepetitions - You can specify the number of Number Of Repetitions for a file stream.
- SharePointDirectory - Share point macro statement where file steams are located.
- DataFileDirectory - Macro statement used to indicate where the data used in the stream is.

An engine is generated for each of the target systems :

The current implementation of the engine supports the following :

- Language Independence - the current implementation of the engine works against both DBCS and non-DBCS languages. These include English, Japanese, French, German.
- Platform independence - currently supported platforms are Win16, Win95, NT, NT-SUR
- Processor independence - currently supported processors are Intel, MIPS, ALPHA

What this means is that when a tester generates a stream the stream will run against all the supported languages, platforms and processors. This is achieved by generating an engine for each of the languages, platforms and processors. Because the engine can be run in so many environments we can ultimately test the majority of target systems for a product using only one stream for a given test scenario. Of course this approach puts pressure on the engine developers to develop very stable and flexible code but the result of this effort is reduced time to develop data streams and maximum reuse. Also, it is very important to document your objects well so that they can be used easily.

Currently supported objects of the Exchange engine :
Bolded items are objects which are of general interest

- CopyDesign
- LogOnOff
- **Logging** - used to log test results
- **Resource** - used to monitor system resources
- View
- Folder
- DesignOverView
- Form
- Permission
- Rules
- AdvancedCriteria
- User
- **Test** - used to dispatch tests remotely
- Message
- File
- Install

Amazon : This project is called Amazon because the large number of objects implemented by the stream-based interpreter.

How to utilize already existing automation :
As mentioned earlier, the engine itself is quite large. Much of the source code existed before the interpreter. Wrappers were written to map the objects to the pre-existing source code.Objects are defined as types composed of strings as show below. Here we return to the example of the folder object.

```
TYPE FolderStruct
  m_StoreName AS STRING
  m_FolderName AS STRING
END TYPE ' FolderStruct
```

Functions are passed a pointer to this structure as the one and only passed parameter. This makes for maximum flexibility because calls to functions don't need to be changed as parameters are added. Within your wrapper function you can call your pre-existing automation code as shown in following example :

```
FUNCTION Folder_Create (ptrFolderObject AS POINTER TO FolderStruct) AS INTEGER

    ' Call your folder create routine here

END FUNCTION 'end Folder_Create()
```

More advanced use of the engine :
- Running tests remotely - the engine is currently being used to test remotely on other machines. A copy of the engine is run on remote machines. The engine waits for streams to execute.
- Multi-user testing - file streams are sent to remote machines. The streams instruct the clients to perform operations against commonly defined objects.

Conclusion :

Benefits
- Allows anyone to quickly define tests. Tests can be designed without writing code scripts. Because the engine is a stand alone application there is no need to recompile. Currently, there are several groups running the ENGINE automation and have found it very easy to create large numbers of exhaustive suites.
- Because file streams are text files which are read by the engine it is easy to automatically generate them with code, awk scripts, etc. Currently, the Electronic Forms Designer test team has created over 12 MEG ofDAT files for testing. This work was done by only 2 people working part-time on this task.
- Testing many permutations is very easy. This is facilitated by recursive inclusion and execution of the file streams.
- All the complexity is in the engine. Therefore, you have 100% reuse of automation code. Changes to the target project can be responded to quickly and efficiently since all object support code is within the engine and not scattered about test code scripts for each scenario.
- Object definitions provide a standard interface to the underlying functionality. Since there is, for example, only one folder object, quality and consistency can be maintained because there is only one way to access the functionality.
- Finds a stress related / resource related / permutation bugs.

Drawbacks
- You have to implement the engine. Each object has data elements and functions. You must define the object structure and write the source code to perform the function(s).

Experiences using controlled iterative development on an object oriented development effort.

Abstract:
The advantages of iterative development have been discussed and described in numerous publications and through reported project experiences. Equally praised has been the use of object oriented development. While we agree that there are significant benefits to both these approaches, we found barriers to their effective use when we embarked on a project to develop a customer contact system for a major state university.

Without procedures and practices in place to control and manage the unique nature of iteration, development can spiral out of control, resulting in poor quality, slipped deadlines and unmet user expectations. This paper will describe the challenges we faced and our efforts to "get control" of the development process by defining and implementing practices that focus on managing object oriented, iterative development. This approach we refer to as "controlled iteration".

Biographical:

Frank Armour
American Management Systems
4050 Legato Rd
Fairfax, Va 22033
Phone (703) 267-2206
FAX (703) 267-2222
frank_armour@mail.amsinc.com

Frank Armour is currently AMS's object technology methodologist and is a member of the AMS Corporate Technology Group, where he is responsible for the development of AMS's Object Technology Methodology. 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.

Monica Sood
American Management Systems
4050 Legato Rd
Fairfax, Va 22033
Phone (703) 267- 5982
FAX (703) 267-2222
Monica_Sood@mail.amsinc.com

Monica Sood is an AMS object technologist and a member of the AMS Corporate Technology Group. Monica's responsibilities include development of AMS's Object Technology Methodology. In her role as an Object Technologist, Monica offers expertise in development of object-oriented client-server systems and application of object methodology and technology to AMS software projects. Monica is an associate of AMS Center for Advanced Technologies and an associate of AMS System Development and Information Technology Management Knowledge Center.

Marilyn Kraus
American Management Systems
4050 Legato Rd
Fairfax, Va 22033
Phone (703) 267- 3310
FAX (703) 267-2196
mfkraus@msn.com

Marilyn Kraus is an AMS project manager and technologist. Marilyn is a member of the AMS Higher Education Consulting and Technology group helping colleges and universities achieve breakthrough performance through the intelligent use of technology. Marilyn is an associate of the AMS Center for Advanced Technologies and an associate of the AMS System Development and Information Technology Management Knowledge Center.

Copyright:

Experiences using controlled iterative development on an object oriented development effort.

Frank Armour, Marilyn Kraus and Monica Sood
American Management Systems

Introduction

The advantages of iterative development have been discussed and described in numerous publications and through reported project experiences. Equally praised has been the use of object oriented development. While we agree that there are significant benefits to both these approaches, we found barriers to their effective use when we embarked on a project to develop a customer contact system for a major state university.

Without procedures and practices in place to control and manage the unique nature of iteration, development can spiral out of control, resulting in poor quality, slipped deadlines and unmet user expectations. This paper will describe the challenges we faced and our efforts to "get control" of the development process by defining and implementing practices that focus on managing object oriented, iterative development. This approach we refer to as "controlled iteration".

In our paper we share our experiences with object oriented iterative development, and discuss the practices we implemented to help ensure a quality outcome.

Background

The system we built used an object oriented, client server architecture. The architecture consisted of Microsoft Windows clients, UNIX servers, and an IBM mainframe. The database server was Oracle. The primary development language was Smalltalk.

The system serves as the primary contact tracking system for a major research university. The system also has additional functionality to distribute materials to students and to schedule recruiting events. Although the system was developed to track perspective students, it was designed to be reusable and expandable so that eventually it may be used as the tracking system for all students, alumni, and university employees.

Advantages of Iterative Development

Iterative development (see Figure 1a) has advantages over a waterfall approach (see Figure 1b) that are well documented, they include:

- Better handling of unknown or volatile requirements
- The ability to try things out first such as proof of concepts
  - e.g. Demo end-to-end capabilities.
- More extensive user involvement throughout the system life cycle
- Faster delivery of functionality vs. Nothing to show or validate with users

However, we also experienced significant challenges with iterative development.

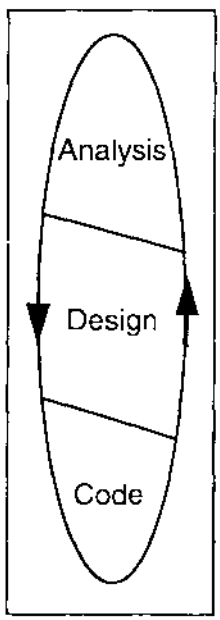


Figure 1a - Iterative Model

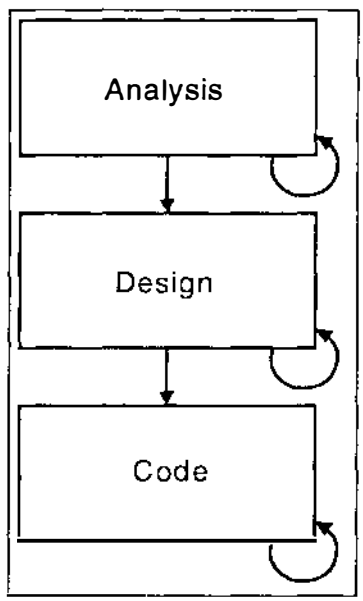


Figure 1b - Waterfall Model

Challenges Faced With Iterative and Object Oriented Development

Some of the challenges that we faced iterative and object oriented development include:

- Knowing when you are done is harder, since you have not done a complete requirements analysis up-front. How do you measure completeness? What metrics do you use to determine scope?
- Meeting interim milestones is more difficult, since you can always go back and rework something. When is a design complete? When is an object coded? When can you declare something is "finished"?

- Developers are unclear on when to apply quality practices such as unit testing and coding standards since "we will have to rework it anyway".
- There are many more versions and releases making configuration management much more difficult, yet at the same time more critical.

We learned from these challenges that our success depended on our configuration management process, our quality plan and our ability to communicate well at all levels.

Object development provides enhanced reuse and a higher degree of modularity, but has side effects due to the level of granularity of the pieces. Objects tend to be at a finer granularity than the modules we assigned traditionally to individual developers. As methods within the objects are developed, new versions of the objects are made available to other developers. As increments are developed, groups of objects are released for testing. The complexity resulting from a finer level of granularity, multiple versions of objects and multiple releases of functionality had the following side effects:

- Object methods are coded iteratively, not all at once as we did with modules in structured languages. Since business functions are delivered by objects collaborating with each other, object development requires objects to be made available to other developers as soon as a subset of their methods are complete. Meanwhile, new methods for the objects are coded and the object will again be made available to other developers. Each version of the object requires testing, integration with other objects, and collaboration with the team.
- Finer granularity also increases the number of configurable items. This further complicates configuration management. The success of any milestone depends on consistency in the methods and objects of the milestone. This becomes a true challenge when developers are doing multiple fixes concurrently as during any testing activity.
- Due to the iterative nature and the frequent releases there are more quality "checkpoints" in the process, stressing the quality assurance roles. There are more tests, more deliverables to review, and more releases to manage.
- Since system functionality consists of many integrated objects, there are many more possible interfaces between objects and developers than with more monolithic structured development. This makes team communication more complicated.

The combination of iterative and object development can, if not controlled and managed, result in developers getting caught up in a: "quick develop, release, fix, release, fix syndrome". The result can be a thrashing that extends development time, results in poor system quality, and frustrates all parties involved.

Lessons Learned/Experiences During Development

In order to address the above, we implemented what we call a "controlled" iterative development model. We define controlled iterative as iterative which is managed to ensure that development scope is effectively maintained, milestones are defined and measurable, and quality is defined and enforced. We established quality practices designed to effectively develop with iteration and objects. Specifically, the activities (see Figure 2) we performed to help control iteration include:

- Modification of the iterative process model
- Specification of detailed functionality early in an increment
- Establishment of cohesive, well organized teams
- Development and implementation of quality procedures such as
  - ◊ Methodology and software engineering practices
  - ◊ Peer reviews and walkthroughs
  - ◊ Configuration management processes and practices
- Continuous process improvement
- Engaged project management

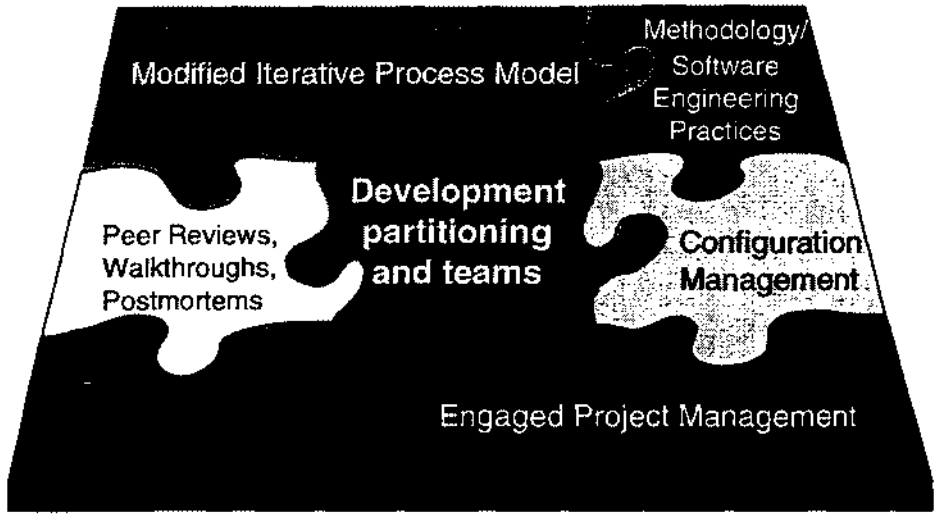


Figure 2 - Activities performed to control iteration

Modification of the Iterative Process Model

Without an understanding of overall project objectives and requirements, how does one manage project scope? How does one know when to stop? We needed to address two separate issues:

(1) What is the completion criteria for an iteration?
(2) What scoping metrics could we use, if we had a limited idea of the requirements?

These issues gave us pause before jumping into an pure iterative development cycle. Yet we knew that a pure waterfall approach would prove too rigid for our purposes, since our requirements could not be fully defined up front. We therefore combined the iterative, incremental and waterfall models. The specific modifications to the iterative nature of the process included:

- A more waterfall oriented front end that included: A system concept, high level requirements, conceptual design, and a technical architecture. This approach helped us control scope and manage the evolution of the system architecture during the iterative development phases. The front end results were used to guide and direct the iterative development effort that followed.
- We made a distinction between incremental and iterative development. see Figure 3. ***Incremental development*** is the development of a system in a series of versions or steps. A subset of functionality is selected, designed, developed, and then implemented. Additional increments build on this functionality until the system is completely developed. ***Iterative development*** is the reworking of existing functionality, and is largely isolated within each increment. On our project each increment was ten to twelve weeks long. The entire development phase of the project lasted about seven months.

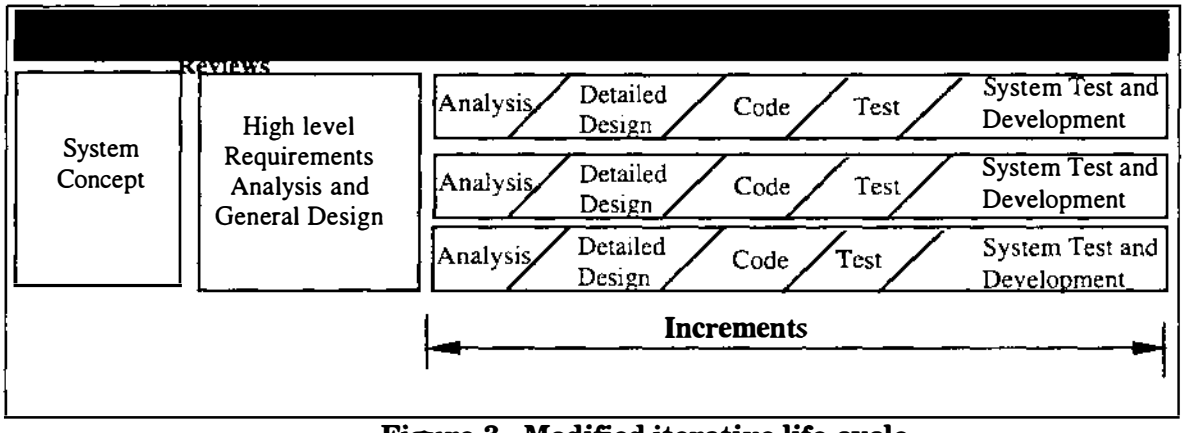


Figure 3 - Modified iterative life-cycle

Specification of Functionality

In a pure iterative approach, time estimates for interim milestones are difficult to establish, since the functionality of an iteration has not been clearly defined. The incremental approach allowed us to take small pieces of system functionality such as: Business requirements, frameworks, system infrastructure, user document and training, and develop them as mini projects. At the beginning of each increment there was a need to clearly identify and isolate the functionality that was to be

developed during the increment. Without this definition it is impossible to know when the increment is complete. Additionally, in the absence of clearly defined increment scope, developers were highly likely to delay function or features until the next release.

We addressed these challenges in the following manner:

At the beginning of each increment the functionality that was to be derived from the increment was selected and refined. The outcome was a functional specification that included among other things: The level and scope of the business functionality, GUI navigation, system performance, and database access.

Going into each increment we selected a subset of functionality from the High Level Requirements and General Design previously developed. We choose this functionality based on: Risk, immediate user needs, and the need for foundation software. At the start of each increment we revisited the functionality selected, then updated and defined the functionality in more detail. For example, at the end of the General Design we had only a broad understanding of what a "student" was. With each increment we further refined the functionality associated with a student and the system design to handle the requirements. To support the iteratively refined requirements, the student object evolved from being a single Smalltalk object to a group of objects.

We then generated a detailed design, specific to the functionality developed in the increment. The designs materials included:

- A physical object description (e.g. object relationships, methods, attributes that were in the syntax of the programming language to be used).

- A physical database design, refined GUIs, and refined test scripts.

- Design packets that were then implemented by the developers.

We developed this detailed set of designs early during each increment (first two to three weeks). We then used the designs as a baseline to determine progress. This did not mean that we had the functionality perfect the first time. We, had several iterations of analysis, design, code and test in each increment. However, by actively working to define detailed functionality and completion criteria early in the increment life cycle, we had a good yardstick to measure completeness and control scope (See figure 4). We found that this prevented getting caught in perpetual, continual refinement. At some point design had to stop and be baselined for developers to develop against. Our developers had found that they needed a static design to develop from for that iteration. This is also necessary for consistency. Otherwise, we could never have gotten through object integration testing.

We found that as increments were developed the understanding of the business functionality, the implication of the design and infrastructure, and the development capabilities grew. This often resulted in requests for changes initiated by users, developers and designers during an increment. We learned there was no such thing as minimal change and that the best approach to these requests was to keep user requirements balanced with design implications and consistent with development capabilities and milestone delivery dates. Even small changes result in more testing and possible changes to test scripts, technical documentation, user documentation, and project schedule. For example, during our first increment we learned that our cross-platform development environment did not provide support for some Windows standards like hot-keys and we decided to add hot key support to the increment. This required changes to the infrastructure, changes to objects, changes to the GUIs, and changes to the testing scripts. Ultimately, the users were pleased with the result however, the lesson we learned was to assess the impact of any change considered once the increment is well underway. We found that the demand for change to be inherent to iterative, incremental development but we learned that we had to evaluate the change and be prepared to adjust the increment delivery date before we agreed to include the change. Since changes affect everyone, the users, designers, developers and testers, we learned to involve everyone in these decisions.

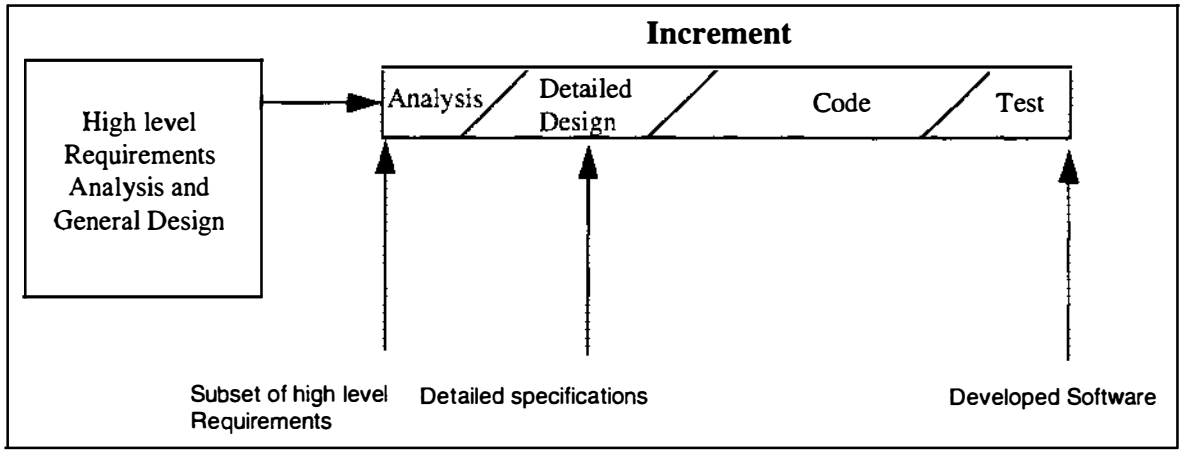


Figure 4 - Increment Life Cycle

Just as important as the definition of functionality, was the level of quality expected from each increment, as well as the quality practices needed to achieve these levels. However, in an incremental model, where progress is measured by completion of increments, if quality is not actively built in as one goes, then the increments cannot be scoped effectively. In an iterative and incremental development effort, it is very easy to skip these quality steps, since you are going to "rework" functionality anyway. Without a firm definition of the quality expectation, developers will say "oh, we will clean it up later", resulting in more work later. The first iteration of our first increment was to implement the prototype developed in

general design as a navigable Smalltalk application. We found different developers had different ideas about what a navigable application was. We addressed this problem by defining more clearly what was expected. In later increments we always defined clearly what we expected in terms of quality and scope and we constantly worked to improve communications. The quality practices we used are discussed in a later section.

Development Partitioning and Cohesive Teams Structure

Because of iterative, incremental, object development, coordination and integration issues are extremely important. We faced this challenge by using small, cohesive teams, effectively partioning the development, building an environment which fostered good communications, and clearly defining project goals.

Our teams of five or six developers fostered effective communications. Initially, the teams were interdependent and effectively we had one large team. When this problem was identified, we repartioned the work so the two teams were more independent. We also physically located teams members in close proximity and found we had fostered real team synergy.

Because of the rapid deadlines, it was vital that developers act as a team. Developers needed to understand project goals and the overall design. This was important since one can't build quality when doing a jigsaw puzzle unless everyone has the picture in front of them. For example, the team developing the student contact functionality had difficulty understanding the interfaces with the distribution materials being developed by another team. We then had to step back and educate the developers on the larger design picture. A fast paced iterative effort requires effective communication as well as communication mechanisms such as designs.

Within a team, we made each developer as independent as possible, both functionally (e.g. in our case student contact functions, material distribution and event scheduling) as well as technically (e.g. by platform, by language etc.). Some of the object oriented literature would make one believe that objects are magically loosely coupled. There is nothing further from the truth. Objects can have many dependencies. During detailed design we identified groups of objects that were tightly coupled, and assigned these groups to individual developers. This allowed them to work as independently as possible. However, team members should be cross trained so that they can help out the other team members in a crunch at end of increment.

We found it a challenge to coordinate and balance the effort within a development team. As the increment begins, developers need to understand the dependencies so

functionality which other developers need is worked on first. Toward the end of the increment, some developers may finish earlier than others. We asked our developers to finish the hard stuff first, so that at the end of the increment, other developers could step in and complete the easier pieces. This approach worked well in helping balance the effort required to meet the deadlines.

Development and Implementation of Quality Procedures

An advantage of iterative development is that it provides you with the opportunities to try out practices in an iterative manner and then make necessary modifications before the next increment.

The development and implementation of quality procedures was particularly challenging on this iterative project. On one hand, we did not wish to so burden or weigh down the iterative process that the benefits of iteration were lost. On the other hand, the development process had to be managed so it did not spiral out of control. For example, with the frequent modification and release of objects, our developers needed a unit test approach that supported the goals of unit testing, but did not require a lengthy unit test activity. We developed a unit testing approach that focused at the object method level. Whenever developers created or changed a method, they were required to define test cases for the method. We used a home grown automated testing tool to allow the developers to define a new test case, run the test case, store the test case along with its association to the method, and quickly rerun existing test cases developed for previous releases.

Among the quality practices we performed and found beneficial included:

Methodology and Software Engineering Practices

Although it seems like it should be gospel that good software engineering methods are important, we still had developers and management wanting to skip important development steps.

In iterative development it is very easy to jump right in and start coding, this is a seductive but dangerous approach. Many problems dealing with the large number of interactions between objects and developers had to be addressed in the design, by identifying cohesive frameworks that had loose coupling. Then managing their rework through a controlled process. Frameworks, which are groups of objects that perform a single function and are designed to be extended and reused, are most often discussed as reuse tools. We utilized them as a development partitioning tool as well. Development responsibilities were divided along framework lines.

During an iterative effort testing will occur frequently. There is a need to perform testing during an iteration, at the end of an iteration and finally at the end of an

increment before system deployment. In addition to the classic unit testing, we defined three levels of testing for our effort:

- Iteration testing (similar to integration testing) is testing of functionality developed during the iterations that occurs within an increment.

- Increment testing, which occurs at the end of an increment, is testing functionality developed during that increment. It validates the functionality developed against the specified functionality defined at the beginning of the increment. Each increment test builds on the iteration tests.

- System testing focuses on testing accuracy and completion of functionality for the entire system, and occurs at the end of the development effort. The system test builds on the previous increment tests.

For each increment test it took us two to four weeks to carry out the test and fix any bugs that were found.

We tested and tested early. Every time an iteration test was pushed off, it added geometrically to the increment test. Based on our experience, if putting a test in place for the iteration costs n hours, waiting to test that functionality until the next iteration or the increment test will cost at least $3n$ hours.

Peer Reviews and Walkthroughs Were Invaluable

Peer reviews and walkthroughs are just as important and necessary in iterative development as before. Peer reviews and walkthroughs are a structured line-by-line review of a design, object or other deliverable against a standard checklist. On our project the purpose of the reviews were to:

- Identify better ways of doing things
- Find discrepancies in cross developer code/design
- Reaffirm the design
- Share knowledge
- Implement and improve standards

During a walkthrough, the deliverable was checked for all types of errors. Since there are many interactions between objects, it was important that reviews, in addition to identifying improvements in the objects, were used to help educate developers about "what else is going on". Also, when developing with an advanced technology, such as object oriented development, it is highly likely that there will be a large number of inexperienced developers. This was certainly true in our case. Code and design walkthroughs helped educate our inexperienced developers. We

found that the time it took to formally test could be significantly reduced if we followed review practices during the increment.

Configuration Management

Configuration management is critical to managing the complexity. We spent a great deal of time and resources on this issue. Before development started, it was essential to have a configuration management plan in place. We evaluated and selected the appropriate configuration tools and defined configuration management processes like process and schedule for releasing code.

Due to the nature of iterative development, we had to address complex configuration management issues such as:

- **More than one developer updating the same object**. This introduced two different challenges: 1) two developers make changes to an object simultaneously and 2) two developers make incompatible changes to an object. In the case of the former, changes made by one of the two developers will be lost and would have to be recoded once the error is discovered. In both cases, the only hope of catching the error early was thorough unit and object integration testing. However, this introduced the challenge of who was responsible for updating and executing the test. Do both developers test? The object owner bears the responsibility that the object is tested according to standards.
- **Having to fix and release an earlier increment while incorporating the fix into current increment**. In our case, we had small enough increments that we decided not to deal with this complex issue. We did not make fixes to earlier increments and re-release the increment. Fixes from one increment were released with the next increment. The primary reason for this decision was because we could not come up with a reasonable way of maintaining different versions of the software.
- **Assessing impact of changing an object on objects and functionality developed in previous iterations**. For example, if changes to object 1 effect another object developed in a previous iteration, the system architect was responsible for assessing and identifying the effect, for testing it, and for communicating the resulting changes to the rest of the team. And since our developers didn't always have the big picture they could not assess the impact of changes.
- **Cross platform development**. Managing code and tools for different platforms and development applications/tools. We used Envy, a Smalltalk configuration management tool for code management, which worked well for managing the Smalltalk objects. However, Envy is a closed environment and we, therefore, needed to manually synchronize the objects in the Smalltalk image and the database tables database schema, test scripts, database population scripts, test data, and design documents across all configurations. Initially, we made each developer responsible for maintaining scripts for

database tables they owned. The result was when trying to migrate from development to testing and from testing to production it would take days of trial and error in creating the composite database. We finally moved to having the configuration manager control all changes to the database scripts even during development.

- **Integrating different software components and creating the system applications**. Even though it only took two hours to cut and strip a Smalltalk image of our application, just getting all the right pieces identified, integrating them, initializing them, and finally cutting the right image generally us two days. We found that after stripping the image, we had to run the regression test again to be sure we were giving the right application to the testing teams.
- **Managing multiple environments**, development vs. testing vs. production. Keeping a handle on what versions of the software and database tables was running in each different environment was a challenge in itself.

In an iterative development effort using object oriented technology, we discovered that configuration management is not a simple or trivial task. A lesson we learned was to put a senior developer in charge of configuration management and have them work closely with the object designers and developers to uncover potential impacts of changing code.

Engaged Project Management

Things happen so fast in iterative development and milestones come more frequently, that management must stay actively involved in tracking developer progress as well as providing needed support to the developers. This is not hands off management.

Since developers are learning as they go, there will be a lot of rework. A key lesson we learned is that if you do not allow time for the rework that is needed, you will not meet your schedule. We iterated through everything, not just development: Procedures, processes, designs, and standards. This is a natural cycle, however, do not wait until you understand sometime perfectly to try it out. Management set expectations both with the team and the customers that there would be rework, and that things would not be perfect the first time.

A characteristic of iterative development is heavy user involvement. On the positive side, there is lots of user feedback and suggestions early in development, resulting in a final system that better meets the user's needs. For example, throughout the development effort, our users were able to try out and work with live data, as well as both use and provide feedback on the GUI look and feel. However, we found that with the increased user feedback comes increased potential for an increase in functional scope. Additionally, continual unnecessary rework adds to developer's and tester's frustration level. For example, each time the tester

retested the same functionality, they got bored and were less likely to be as rigorous as the first time around. Unnecessary continuous change and rework is bad for the team moral. One of the best quality practices our manager's had was the ability and willingness to just say "NO".

Continuous Process Improvement

We constantly evaluated our processes for ways to improve. For example, in the first increment, we discovered problems with our object integration testing process. We reworked this process immediately and improved the product we delivered to the users. In addition, at the end of each increment did a postmortem on what worked and what did not work. Because these sessions helped us learn from our mistakes, they facilitated process improvement. We modified our practices to reflect the lessons learned in the increment. This is an easy step to skip because you will feel pressure to continue right in on the next increment. The developers, not just management, were actively involved in the reviews not only for their valuable insight, but also so they had ownership of the changes. Many of the results that are described in this paper were identified and defined through the postmortem process.

Conclusions

We found that there are unique challenges in delivering quality software during an iterative, object oriented development effort. Quality activities and procedures are need to address these challenges.

Our experiences have motivated us to develop quality approaches that address these issues, including.

- A Incremental/iterative development process model
- Functionality must be clearly defined for each increment
- Establishment of cohesive, well organized teams
- Development and implementation of quality procedures
- Active and engaged project management
- Continuous process improvement

Iterative and object development has many advantages, however, without procedures and practices in place to control and manage the unique nature of iteration, development can spiral out of control, resulting in poor quality, slipped deadlines and unmet user expectations.

References

Alistair Cockburn, Unraveling incremental development, Object Magazine, January, 1995.

Alistair Cockburn, The impact of object orientation on application development, IBM Systems Journal, August, 1993.

John Connell and Linda Shafer, *Object-Oriented Rapid Prototyping*, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

Daniel P. Freedman and Gerald M Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews*, Third Edition, Dorset House, New York, 1990.

Shel Siegel, *Object Oriented Software Testing*, John Wiley & Sons, Inc., New York, 1996.

Software Quality in 1996: What Works and What Doesn't?

Capers Jones, Software Productivity Research Inc.

Abstract

Software quality is subject to many different claims but little solid data. The talk ranks quality methods in descending order of effectiveness.

About the speaker

Capers Jones is Chairman of Software Productivity Research, Inc. (SPR) in Burlington, Massachusetts. Mr. Jones and his colleagues at SPR collect software quality and productivity data on a global basis. SPR also builds software quality estimation and measurement tools.

Mr. Jones is the author of eight books on software quality and productivity topics. His three most recent books are Assessment and Control of Software Risks *(Prentice-Hall, 1994),* Patterns of Software Systems Failure and Success *(International Thompson Press, 1995), and the 2nd edition of* Applied Software Measurement *(McGraw-Hill, 1996). He is working on a new book for 1996 publication with International Thompson Press:* Software Quality—The Key to Competition in the 21st Century.

SOFTWARE QUALITY IN 1996: WHAT WORKS AND WHAT DOESN'T

One New England Executive Park
Burlington, Massachusetts 01803
617 273 0140 Fax: 617 273 5176
http://www.spr.com

NATIONAL IMPLICATIONS OF TECHNOLOGY

- High-technology products are critical to U.S. success
- Quality is the key market factor for high technology
- Computers and software permeate high-technology business
- Quality is the key to software success
- Quality must become part of the U.S. culture
- Senior executive action is needed

FUNDAMENTAL BUSINESS LAWS OF 2000 AD

LAW 1: Enterprises that master computers and software will succeed; enterprises that fall behind will fail!

LAW 2: Quality control is the key to mastering computing and software. Enterprises that control quality will succeed. Enterprises that do not control quality will fail.

LAW 3: Quality cannot be controlled unless it can be measured.

BASIC DEFINITIONS

SOFTWARE QUALITY	"Software that combines the characteristics of low defect rates and high user satisfaction"
USER SATISFACTION	"Clients that are pleased with a vendor's products, quality levels, ease of use, and support"

CAUTIONS ABOUT HAZARDOUS QUALITY DEFINITIONS

"Quality Means Conformance to requirements."

Requirements contain 15% of software errors.

Requirements Grow at 2% per month.

Do you conform to requirements errors?

Do you conform to totally new requirements?

Whose requirements are you trying to satisfy?

CAUTIONS ABOUT HAZARDOUS QUALITY METRICS

"Cost per Defect"

- Approaches Infinity as defects near zero
- Conceals real economic value of quality

COST PER DEFECT PENALIZES QUALITY

	Ⓐ Poor Quality	Ⓑ Good Quality	Ⓒ Excellent Quality	Ⓓ Zero Defects
Function Points	100	100	100	100
Bugs Discovered	500	50	5	0
Preparation	$5,000	$5,000	$5,000	$5,000
Removal	$5,000	$2,500	$1,000	$ 0
Repairs	$25,000	$5,000	$1,000	$ 0
Total	$35,000	$12,500	$7,000	$5,000
Cost per Defect	$70	$250	$1,400	∞
Cost per Function Point	$350	$125	$70	$50

BASIS OF THE "LINES OF CODE" QUALITY PARADOX

When defects are found in multiple components, it is invalid to assign all defects to a single component.

Software defects are found in:

requirements
design
source code
user documents
bad fixes (secondary defects)

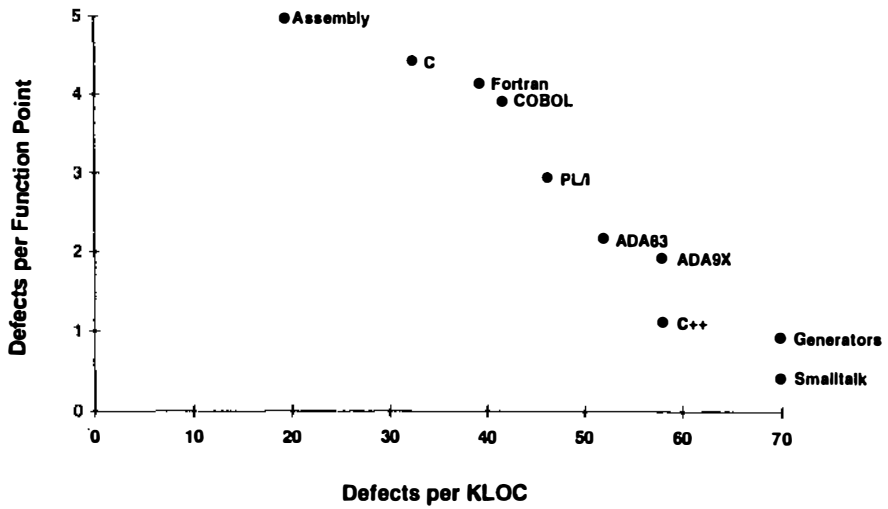
Requirements and design defects outnumber code defects.

"Defects per KLOC" makes major sources of software defects invisible.

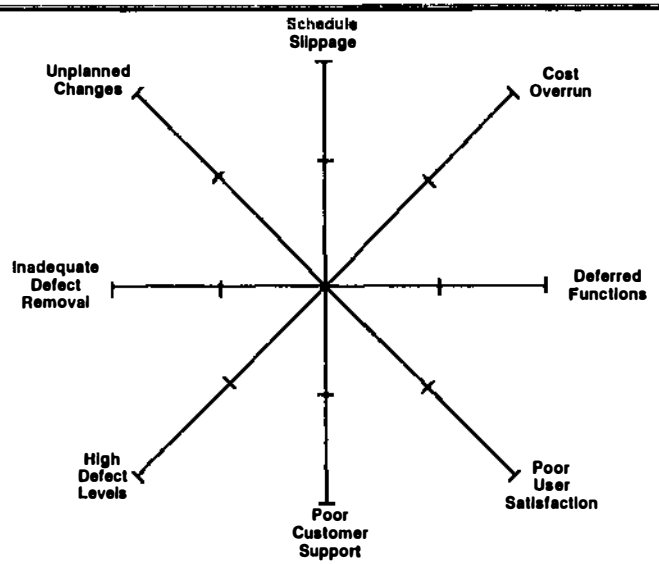
FOUR LANGUAGE COMPARISON OF SOFTWARE DEFECT POTENTIALS

Defect Origin	Assembly	Ada	Objective C	Full Reuse
Requirements	35	35	35	15
Design	75	75	50	6
Code	165	25	10	2
Documents	50	50	50	10
Bad Fixes	25	15	5	2
TOTAL DEFECTS	*300*	*200*	*150*	*35*
Defects per KLOC	30	100	120	140
Defects per Function Point	6	4	2.4	0.7

LOC VERSUS FUNCTION POINT QUALITY LEVELS



KIVIAT GRAPH OF MAJOR SOFTWARE RISKS



CONSISTENTLY GOOD QUALITY RESULTS

- Formal Inspections
- Joint Application Design (JAD)
- Quality Metrics
- Removal Efficiency Measurements
- Functional Metrics
- Active Quality Assurance
- Formal Configuration Control
- User Satisfaction Surveys
- Formal Test Planning
- Quality Estimation Tools
- Automated Test Tools

MIXED QUALITY RESULTS

- Total Quality Management (TQM)
- Quality Function Deployment (QFD)
- SEI Assessments
- SEI Maturity Levels
- Baldrige Awards
- IEEE Quality Standards
- Testing by Developers
- DOD 2167A and DOD 498
- Reliability Models
- Risk Assessments

QUESTIONABLE QUALITY RESULTS

- ISO Quality Standards
- Informal Testing
- Manual Testing
- Passive Quality Assurance
- LOC Metrics

A PRACTICAL DEFINITION OF SOFTWARE QUALITY (PREDICTABLE AND MEASURABLE)

- Low Defect Potentials (< 1 per Function Point)
- High Defect Removal Efficiency (> 95%)
- Unambiguous, Stable Requirements (< 2.5% change)
- Explicit Requirements Achieved (> 97.5% achieved)
- High User Satisfaction Ratings (> 0% "excellent")
  - Installation
  - Ease of learning
  - Ease of use
  - Functionality
  - Compatibility
  - Error handling
  - User information (screens, manuals, tutorials)
  - Customer support
  - Defect repairs

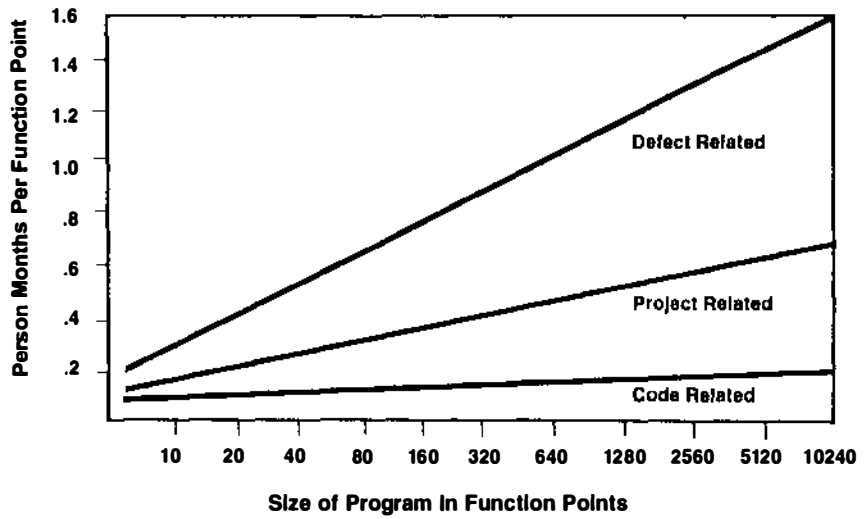
SPR AND ISO QUALITY PROCESSES

	SPR	ISO
Defect Potential Estimation	Yes	Missing
Defect Removal Efficiency Estimation and Measurement	Yes	Missing
Delivered Defect Estimation and Measurement	Yes	Yes
User Satisfaction Measurement	Yes	Yes
Inspections and Reviews	Rigorous	Informal
Testing	Rigorous	Rigorous
Process Analysis	Rigorous	Informal

OVERVIEW OF ISO 9000-9004 STANDARDS TOPICS

ISO assessment accuracy	Emerging topic, needs more work
ISO assessment validity	Emerging topic; needs more work
ISO costs/investments required	No current literature at all; costs may be high
ISO impact on software quality	Many claims, but little empirical data
ISO impact on software productivity	No current literature at all
ISO impact on software schedules	No current literature; longer schedules likely
ISO value to domestic companies	No current literature at all; low value likely
ISO value to internationals	ISO certification now mandatory
ISO certification automation	No current literature; no known tools
ISO impact on paperwork	May increase paperwork volumes

WORK CATEGORIES RELATED TO PRODUCT SIZE



PERCENTAGE OF SOFTWARE EFFORT BY TASK

Size in Function Points	Mgt./ Support	Defect Removal	Paperwork	Coding	Total
10,240	18%	35%	35%	12%	100%
5,120	17%	33%	32%	18%	100%
2,580	16%	31%	29%	24%	100%
1,280	15%	29%	26%	30%	100%
640	14%	27%	23%	36%	100%
320	13%	25%	20%	42%	100%
160	12%	23%	17%	48%	100%
80	11%	21%	14%	54%	100%
40	10%	19%	11%	60%	100%
20	9%	17%	8%	66%	100%
10	8%	15%	5%	72%	100%

U. S. SOFTWARE QUALITY AVERAGES

(Defects per Function Point))

	System Software	Commercial Software	Information Software	Military Software	Overall Average
Defect Potentials	6.0	5.0	4.5	7.0	5.6
Defect Removal Efficiency	94%	90%	73%	96%	88%
Delivered Defects	0.4	0.5	1.2	0.3	0.65
First Year Discovery Rate	65%	70%	30%	75%	60%
First Year Reported Defects	0.26	0.35	0.36	0.23	0.30

CURRENT U.S. AVERAGES FOR SOFTWARE QUALITY

(Data Expressed in Terms of Defects per Function Point)

Defect Origins	Defect Potential	Removal Efficiency	Delivered Defects
Requirements	1.00	77%	0.23
Design	1.25	85%	0.19
Coding	1.25	95%	0.09
Documents	0.60	80%	0.12
Bad Fixes	0.40	70%	0.12
TOTAL	5.00	85%	0.75

CONCLUSIONS

Projects with large volumes of coding defects have the highest removal efficiencies

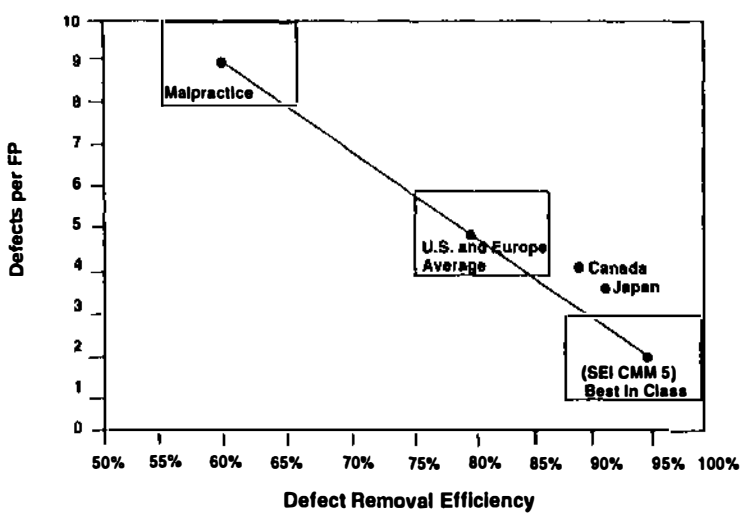
High-level and O-O languages have low volumes of coding defects

SOFTWARE DEFECT POTENTIALS & DEFECT REMOVAL EFFICIENCY SUGGESTED FOR EACH LEVEL OF SEI CMM

(Data Expressed in Terms of Defects per Function Point)

SEI CMM Levels	Defect Potentials	Removal Efficiency	Delivered Defects
SEI CMM 1	5.00	85%	0.75
SEI CMM 2	4.00	89%	0.44
SEI CMM 3	3.00	91%	0.27
SEI CMM 4	2.00	93%	0.14
SEI CMM 5	1.00	95%	0.05

SOFTWARE QUALITY IMPROVEMENT



U.S. INDUSTRIES EXCEEDING 95% IN CUMULATIVE DEFECT REMOVAL EFFICIENCY

	Year 95% Exceeded (Approximate)
1. Telecommunications Manufacturing	1975
2. Computer Manufacturing	1977
3. Aero-space Manufacturing	1979
4. Military and Defense Manufacturing	1980
5. Medical Instrument Manufacturing	1980
6. Commercial Software Producers	1992

U.S. INDUSTRIES MAINTAINING MARKET SHARE INTERNATIONALLY

1. Telecommunications Manufacturing
2. Computer Manufacturing
3. Military and Defense Manufacturing
4. Commercial Software Producers
5. Aero-space Manufacturing
6. Medical Instrument Manufacturing

SPR QUALITY PERFORMANCE LEVELS CUMULATIVE DEFECT REMOVAL EFFICIENCY

(Development Defects + 1 Year of User Defect Reports)

SPR Performance Level	Efficiency Measured at One Year of Usage
1. Excellent	> 99%
2. Good	95%
3. Average	87%
4. Marginal	83%
5. Poor	< 80%

OPTIMIZING QUALITY AND PRODUCTIVITY

Projects that achieve 95% cumulative Defect Removal Efficiency will find:

1) Minimum schedules
2) Maximum productivity
3) High levels of user satisfaction
4) Low levels of delivered defects

ORIGIN OF SOFTWARE DEFECTS

Because defect removal is such a major cost element, studying defect origins is a valuable undertaking.

IBM Corporation (MVS)		SPR Corporation (client studies)	
45%	Design errors	20%	Requirements errors
25%	Coding errors	30%	Design errors
20%	Bad fixes	35%	Coding errors
5%	Documentation errors	10%	Bad fixes
5%	Administrative errors	5%	Documentation errors
100%		100%	

TRW Corporation		Mitre Corporation		Nippon Electric Corp.	
60%	Design errors	64%	Design errors	60%	Design errors
40%	Coding errors	36%	Coding errors	40%	Coding errors
100%		100%		100%	

FUNCTION POINTS AND DEFECT REMOVAL

Function points raised to the 0.3 power can predict the optimal number of defect removal stages.

FUNCTION POINTS	DEFECT REMOVAL STAGES
1	1
10	2
100	4
1,000	8
10,000	16
100,000	32
1,000,000	64

FUNCTION POINTS AND TEST CASES

Function Points raised to the 1.2 power can predict the optimal number of test cases.

FUNCTION POINTS	TEST CASES
1	1
10	16
100	250
1,000	4,000
10,000	63,000
100,000	1,000,000
1,000,000	15,000,000

SPR QUALITY PERFORMANCE LEVELS TEST CASES CONSTRUCTED PER FUNCTION POINT

Test Step	Lowest	Median	Highest
Unit Test	0.2	0.3	0.5
New Function Test	0.2	0.35	0.6
Regression Test	0.2	0.3	0.5
Integration Test	0.25	0.4	0.75
Stress Test	0.05	0.1	0.2
System Test	0.1	0.25	0.4
Field Test	0.1	0.15	0.25
Acceptance Test	0.1	0.15	0.25
Total	1.2	2.0	3.0

RELATIONSHIP OF SOFTWARE QUALITY AND PRODUCTIVITY

- **The most effective way of Improving software productivity and shortening project schedules is to reduce defect levels.**
- **Defect reduction can occur through:**
  1. **Defect prevention technologies**
     - **Structured design**
     - **Structured code**
     - **High-level languages**
     - **Etc.**
  2. **Defect removal technologies**
     - **Design reviews**
     - **Code Inspections**
     - **Tests**
     - **Correctness proofs**

DEFECT PREVENTION METHODS

	Requirements Defects	Design Defects	Code Defects	Document Defects	Performance Defects
JAD's	Excellent	Good	Not Applicable	Fair	Poor
Prototypes	Excellent	Excellent	Fair	Not Applicable	Excellent
Structured Methods	Fair	Good	Excellent	Fair	Fair
CASE Tools	Fair	Good	Fair	Fair	Fair
Blueprints & Reusable Code	Excellent	Excellent	Excellent	Excellent	Good
QFD	Good	Excellent	Fair	Poor	Good

DEFECT REMOVAL METHODS

	Requirements Defects	Design Defects	Code Defects	Document Defects	Performance Defects
Reviews/ Inspections	Fair	Excellent	Excellent	Good	Fair
Prototypes	Good	Fair	Fair	Not Applicable	Good
Testing (all forms)	Poor	Poor	Good	Fair	Excellent
Correctness Proofs	Poor	Poor	Good	Fair	Poor

DEFECT REMOVAL ASSUMPTIONS

	Methods	Training	Experience	Enthusiasm	Management Support
1. Excellent	Formal	Formal	Substantial	Good	Good
2. Good	Formal	Formal	Mixed	Good	Moderate
3. Average	Informal	Informal	Mixed	Mixed	Mixed
4. Marginal	Informal	Informal	Little	Minimal	Minimal
5. Poor	Informal	Informal	None	Negative	Minimal

QUALITY MEASUREMENT EXCELLENCE

	Defect Estimation	Defect Tracking	Usability Measures	Complexity Measures	Test Coverage Measures	Removal Measures	Maintenence Measures
1. Excellent	Yes	Yes	Yes	Yes	Yes	Yes	Yes
2. Good	Yes	Yes	Yes	No	Yes	No	Yes
3. Average	No	Yes	Yes	No	Yes	No	Yes
4. Marginal	No	No	Yes	No	Yes	No	Yes
5. Poor	No	No	No	No	No	No	No

TOOLS USED BY SOFTWARE QUALITY ASSURANCE (SQA)

(Tool Capacity Expressed In Function Points)

Tool Categories	Lagging	Average	Leading
Statistical analysis tools			3,000
Quality estimation models			2,500
Spreadsheet	750	1,250	2,000
Graphics/Presentations	750	1,250	2,000
Word processing	500	1,000	2,000
Configuration control	500	1,250	2,000
Test case generators			1,750
Data base	500	1,000	1,500
Defect tracking/Analysis	500	750	1,000
Reliability estimation models		500	1,000
Symbolic debuggers	250	500	750
Electronic mail	300	500	700
Appointment calendar	100	300	750
Phone/Address file	100	150	500
Complexity analyzers			350
Test path coverage analyzers		200	350
Test execution monitors		200	350
Totals	**4,250**	**8,850**	**22,250**

INADEQUATE DEFECT REMOVAL IS THE LEADING CAUSE OF POOR SOFTWARE QUALITY

- Individual programmers are only 25% efficient in finding bugs in their own software.
- The sum of all normal test steps is often less than 70% effective (1 of 3 bugs remains).
- Design Reviews and Code Inspections however are often 65% effective.
- Reviews and Inspections can lower costs and schedules by as much as 30%.

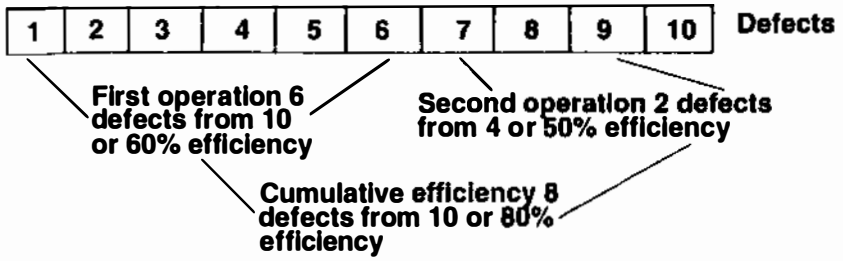
LESS THAN 25% OF U.S. ENTERPRISES USE REVIEWS AND INSPECTIONS

- Most managers have no notion of defect removal rates achieved.
- Reviews and Inspections add significant up-front costs and time.
- Managers do not believe the significant savings gained during integration and testing.
- Most software professionals initially oppose having their work reviewed.

DEFECT REMOVAL EFFICIENCY

- Removal efficiency is the most important quality measure
- Removal efficiency = $\frac{\text{Defects found}}{\text{Defects present}}$
- "Defects present" is the critical parameter

DEFECT REMOVAL EFFICIENCY (cont.)



Defect removal efficiency = Percentage of defects removed by a single level of review, inspection or test

Cumulative defect removal efficiency = Percentage of defects removed by a series of reviews, inspections or tests

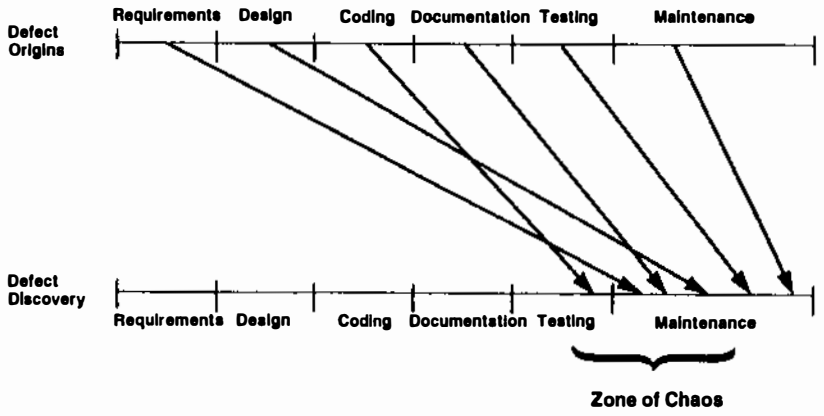
RANGES OF DEFECT REMOVAL EFFICIENCY

	Lowest	Median	Highest
Requirements review	20%	30%	50%
Top-level design reviews	30%	40%	60%
Detailed functional design reviews	30%	45%	65%
Detailed logic design reviews	35%	55%	75%
Code inspections	35%	60%	85%
Unit tests	10%	25%	50%
Function tests	20%	35%	55%
Integration tests	25%	45%	60%
Site/installation tests	25%	50%	65%
	75%	95%	99%

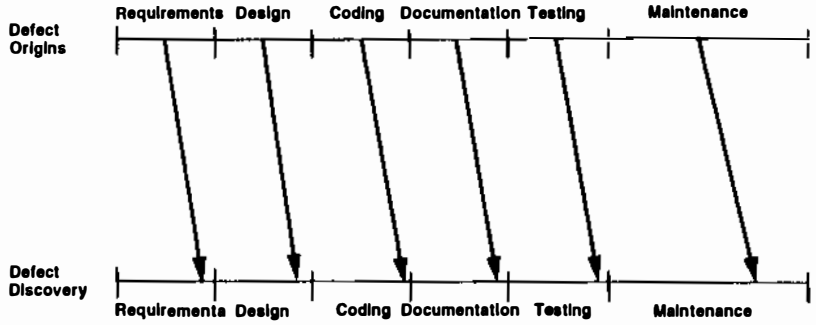
SPR QUALITY PERFORMANCE LEVELS TEST CASES CONSTRUCTED PER FUNCTION POINT

Test Step	Lowest	Median	Highest
Unit Test	0.2	0.3	0.5
New Function Test	0.2	0.35	0.6
Regression Test	0.2	0.3	0.5
Integration Test	0.25	0.4	0.75
Stress Test	0.05	0.1	0.2
System Test	0.1	0.25	0.4
Field Test	0.1	0.15	0.25
Acceptance Test	0.1	0.15	0.25
Total	1.2	2.0	3.0

NORMAL DEFECT ORIGIN/DISCOVERY GAPS



DEFECT ORIGINS/DISCOVERY WITH INSPECTIONS



SOFTWARE DEFECT REMOVAL RANGES

WORST CASE RANGE

TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY		
	Lowest	Median	Highest
1. No Design Inspections No Code Inspections No Quality Assurance No Formal Testing	30%	40%	50%

SOFTWARE DEFECT REMOVAL RANGES (cont.)

SINGLE TECHNOLOGY CHANGES

TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY		
	Lowest	Median	Highest
2. No design inspections No code inspections FORMAL QUALITY ASSURANCE No formal testing	32%	45%	55%
3. No design inspections No code inspections No quality assurance FORMAL TESTING	37%	53%	60%
4. No design inspections FORMAL CODE INSPECTIONS No quality assurance No formal testing	43%	57%	65%
5. FORMAL DESIGN INSPECTIONS No code inspections No quality assurance No formal testing	45%	60%	68%

SOFTWARE DEFECT REMOVAL RANGES (cont.)

TWO TECHNOLOGY CHANGES

TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY		
	Lowest	Median	Highest
6. No design inspections No code inspections FORMAL QUALITY ASSURANCE FORMAL TESTING	50%	65%	75%
7. No design inspections FORMAL CODE INSPECTIONS FORMAL QUALITY ASSURANCE No formal testing	53%	68%	78%
8. No design inspections FORMAL CODE INSPECTIONS No quality assurance FORMAL TESTING	55%	70%	80%

SOFTWARE DEFECT REMOVAL RANGES (cont.)

TWO TECHNOLOGY CHANGES (cont.)

TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY		
	Lowest	Median	Highest
9. FORMAL DESIGN INSPECTIONS No code inspections FORMAL QUALITY ASSURANCE No formal testing	60%	75%	85%
10. FORMAL DESIGN INSPECTIONS No code inspections No quality assurance FORMAL TESTING	65%	80%	87%
11. FORMAL DESIGN INSPECTIONS FORMAL CODE INSPECTIONS No quality assurance No formal testing	70%	85%	90%

SOFTWARE DEFECT REMOVAL RANGES (cont.)

THREE TECHNOLOGY CHANGES

TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY		
	Lowest	Median	Highest
12. No design inspections FORMAL CODE INSPECTIONS FORMAL QUALITY ASSURANCE FORMAL TESTING	75%	87%	93%
13. FORMAL DESIGN INSPECTIONS No code inspections FORMAL QUALITY ASSURANCE FORMAL TESTING	77%	90%	95%
14. FORMAL DESIGN INSPECTIONS FORMAL CODE INSPECTIONS FORMAL QUALITY ASSURANCE No formal testing	83%	95%	97%
15. FORMAL DESIGN INSPECTIONS FORMAL CODE INSPECTIONS No quality assurance FORMAL TESTING	85%	97%	99%

SOFTWARE DEFECT REMOVAL RANGES (cont.)

BEST CASE RANGE

TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY		
	Lowest	Median	Highest
1. Formal design inspections Formal code inspections Formal quality assurance Formal testing	95%	99%	99%

CONCLUSIONS/OBSERVATIONS ON DEFECT REMOVAL

- No single method is adequate.
- Testing alone is insufficient.
- Reviews, inspections and tests combined give high efficiency, lowest costs and shortest schedules.
- Reviews, inspections, tests and prototypes give highest cumulative efficiency.
- Administrative problems need special solutions. Ordinary defect removal is not adequate.
- Maintenance costs are cumulative, expensive and chronic.

Gluing Together Software Components: How Good is Your Glue?

J. Voas, G. McGraw, A. Ghosh
({jmvoas,gem,anup}@rstcorp.com)
Reliable Software Technologies Corporation
21515 Ridgetop Circle, #250, Sterling, VA 20166
phone: (703) 404-9293, fax: (703) 404-9295
http://www.rstcorp.com
and
K. Miller
(miller@uis.edu)
Department of Computer Science
University of Illinois at Springfield
Springfield, IL

Abstract

We have investigated an assessment technique for studying the failure tolerance of large-scale *component-based* information systems. Our technique assesses the propagation of information through the interfaces between objects in order to predict how software will behave when corrupt information gets passed. Our approach is applicable to both source code and executable commercial off-the-shelf (COTS) components. The key benefit of our approach is that it can assess the failure tolerance of legacy systems composed entirely of executable software components.

Biographies

Jeffrey Voas is a co-founder of Reliable Software Technologies (RST) and is currently the principal investigator on research initiatives for ARPA, National Institute of Standards and Technology, and National Science Foundation. He has published over 70 journal and conference papers in the areas of software testability, software reliability, debugging, safety, fault-tolerance, design, and computer security. Before co-founding RST, Voas completed a two-year post- doctoral fellowship sponsored by the National Research Council at the National Aeronautics and Space Administration's Langley Research Center. Voas has served as a program committee member for numerous conferences, and will serve as Conference Chair for COMPASS'97. Voas has coauthored a text entitled Software Assessment: Reliability, Safety, Testability (John Wiley & Sons, 1995 ISBN 0-471-01009-X). In 1994, the Journal of Systems and Software ranked Voas 6th among the 15 top scholars in Systems and Software Engineering. Voas's current research interests include: information security metrics, software dependability metrics, and information warfare tactics. Voas received a Ph.D. in computer science from the College of William & Mary in 1990.

Gary McGraw joined RST's research team in September 1995 after completing a dual PhD in Indiana University in Bloomington. Dr. McGraw's background in in Cognitive Science and Artificial Intelligence. Originally a Philosophy major at the University of Virginia, Dr. McGraw

changed fields to Computer Science for graduate school. At Indiana University, Dr. McGraw's advisor and mentor was Douglas R. Hofstadter — director of the Center for Research on Concepts and Cognition (CRCC) and author of Goedel Escher Bach (among other books). Dr. McGraw's thesis-work on the Letter Spirit project focused on high-level perception and creativity. Dr. McGraw's other work, resulting in several publications, touches on many diverse subfields of Cognitive Science and Machine Learning including Case-Based Reasoning (CBR), Genetic Algorithms (GAs) and Connectionism. His current work at RST focuses on software security. More specifically, the ARPA-sponsored security project involves simulating security threats in such a way that a system's overall security can be empirically measured and numerically rated. Dr. McGraw is a member of the AAAI and the Cognitive Science Society. He is serving on the program committee for ROBOLEARN-96 to be held in conjunction with FLAIRS-96.

Anup Ghosh is a Research Scientist at Reliable Software Technologies. Ghosh is a computer engineer whose research initiatives are in safety-critical system design and analysis, fault simulation, distributed systems simulation, and computer security. Ghosh has published papers in design tools for critical-application systems, analyzing safety from behavioral simulation models, performing fault simulation in the design process, and using fault injection in logic synthesis design. Ghosh completed his Ph.D. from the University of Virginia in Electrical and Computer Engineering. His doctoral research has developed a methodology and supporting set of design tools for performing fault simulation throughout the design process for large systems from high-levels of design abstraction to the logic synthesis level. Ghosh's other research interests are in information coding theory, hardware/software co-design, and software reliability. Ghosh is a member of IEEE and received his M.S. in electrical engineering from the University of Virginia in 1993 and his B.S. in Electrical Engineering from Worcester Polytechnic Institute in 1991.

Keith Miller teaches computer science at the University of Illinois at Springfield. His research interests include testing, reliability, and alternative verification techniques. Dr. Miller has a Ph.D. in computer science from the University of Iowa, and has served as a consultant to Reliable Software Technologies, NASA Langley Research Center, Computer Sciences Corporation, C & P Telephone, Casino International, and others. Recent publications discuss testability, random number generation, and the ethics of computing.

1 Introduction

Information systems are the most complex artifacts in human history. Today's information systems often include distributed hardware and distributed software components, both of which engage in interactions with other machines and humans. The number of different failure modes for just one of these components can be intractably large. When all components are considered in combination, the number of different ways that a system failure can occur becomes "effectively infinite." For example, consider the failure modes for a human operator of a nuclear power plant when 5 different warnings occur simultaneously. Unfortunately, current software reliability theories often make illogical assumptions (*e.g.*, assuming failure independence or completely known failure modes). The truth is that we do not know where the weakest link is in most systems. Furthermore, the location of the weakest link can change with the slightest deviation in the way a system is used. This makes the problem that much harder.

This problem is exacerbated by modern computational approaches such as object-oriented programming and Web-based development. Such new technologies introduce the possibility that programs can no longer be treated as monolithic entities. Instead, modern programs are likely made up of thousands or millions of little parts distributed globally, executing whenever called, and acting as parts of one or more complex systems. These distributed technologies create a scalability problem

for virtually all existing software assessment metrics.

Our approach begins by considering the following initial question: *should we attempt to analyze a system's code when attempting to assess weaknesses in the software's code or might we be able to ignore the code completely?* Since today's systems are by nature very large and source code is often unavailable, the answer to this question is forced on us: *we must create a solution that is source-code independent.* Legacy code introduces a related reason for ignoring the source code in favor of concentrating on analyzing other system aspects — often, legacy source code is either unavailable or the source is written in an out-dated language. But if the code is ignored, what is left to be analyzed for failure tolerance? The answer to this question is: *the behavior of the code* (*i.e.*, how the code runs) and *the interfaces between code objects* (*i.e.*, component interfaces).

Our approach assesses the *robustness* of large-scale systems (that will almost certainly be built from distributed objects) by assessing the robustness of the interfaces between the objects in order to predict how the software will behave when anomalies arise. Our approach does not address the issue of the probability that anomalies will occur. However our definition for robustness will be broader than is generally employed: here, *robustness* is the ability of a system to generate only outputs that are acceptable, regardless of the circumstances that could occur during execution. Furthermore, there are varying degrees of robustness that can be assigned — robustness is not an all or nothing proposition.

The greatest impetus for using component-based technology is the potential gain in productivity. Today, organizations tout modern product-cycle times of 3–4 months — a four-fold decrease over the usual 12–18 months of the past. Speeding the product-cycle requires enormous success in making reusable components of fundamental objects in order to reuse them in a large percentage of software development projects. To make this happen, a development manager must be confident that fundamental objects constructed throughout an organization (and perhaps outside an organization) are compatible and reliable for his or her application. Interestingly, almost 50% of bugs are detected after component integration and not during component development and testing (a number cited as industry standard). Hence developing technologies for decreasing the costs of component composition is imperative.

Complex systems are made up of two key entities: (1) components, and (2) interfaces between components. In her book "Safeware" [2], Leveson acknowledges the necessary role of interfaces in composing complex systems (page 36):

> One way to deal with complexity is to break the complex object into pieces or modules. For very large programs, separating the program into modules can reduce individual component complexity. However, the large number of interfaces created introduce uncontrollable complexity into the design: The more small components there are, the more complex the interface becomes. Errors occur because the human mind is unable to comprehend the many conditions that can arise through interactions of these components. An interface between two programs is comprised of all the assumptions that the programs make about each other ... When changes are made, the entire structure collapses.

Later in the same work, Leveson succinctly describes the importance of thwarting the propagation of failures through the many interfaces that complex systems employ (page 410):

> A tightly coupled system is highly interdependent: Each part is linked to many other parts, so that a failure or unplanned behavior in one can rapidly affect the status of others. A malfunctioning part in a tightly coupled system cannot be easily isolated, either because there is insufficient time to close it off or because its failure affects too

> many other parts. even if the failure does not happen quickly. ...Accidents in tightly coupled systems are a result of unplanned interactions. These interactions can cause a domino effect that eventually leads to a hazardous system state. Coupling exacerbates these problems because of the increased number of interfaces and potential interactions: Small failures can propagate unexpectedly.

The realization that the quality of today's systems is dependent on the quality of component interfaces led us to develop a fault-injection methodology that zeros in on software interfaces. The purpose of our method is to assess the strength (or what we will later call the "failure tolerance") of the interfaces with respect to component failures and problems that may enter the system from external sources. Our method can be applied both to source code and to interfaces between executable components. However, the bulk of this work pertains to analyzing interfaces between executable components.

Interfaces are the mechanisms by which information is passed between components during processing, with the *final interface* being the mechanism through which any output passes to the user. Our technique observes the way that the system reacts when we intentionally corrupt data passing through it. The robustness of an interface under simulated stress provides valuable information about the likelihood that the interface will produce acceptable results when the stress is real (*i.e.*, in real everyday use after release). The robustness of a system is a prediction of the likelihood that malicious or non-malicious anomalies from software components or external sources will result in a loss-of-mission.

Robustness refers to the production of system output states that do not result in a loss-of-mission or loss-of-functionality. Software *propagation* metrics measure the dynamic properties of how corrupt information flows through software as it executes: the idea being to observe corruption in some program state that exists after the original corruption was observed. If it is *impossible* for corrupt information to occur, then robustness is no longer an issue. However, if it *is* possible for corrupt information to enter in the state of a system, then safety and security are at risk.

There are a few additional terms that we must define (somewhat loosely in order to not get bogged down in formalisms). *Propagation of state corruption* occurs from one state to a later state if every state between the two becomes corrupted. The exact manner by which the two states on either "end" of a propagation chain are corrupted is not necessarily important. We say that if component `A` sends information into component `B`, then `A` is the *predecessor* of `B` and `B` is the *successor* of `A`. For each component, we define its *entrance interface* as the interface that reads information in to the component, and its *exit interface* as the interface that passes information out from the component.

The specific type robustness that we will focus on is termed *failure tolerance*. Failure tolerance refers to the ability of a program to produce acceptable output, regardless of what *potential* problems arise during execution. (Fault-tolerance is a very similar term, but usually refers to behavior in the presence of *software* problems. Failure tolerance, on the other hand, encompasses both software problems, human factors, and hardware problems.) Until we are capable of demonstrating interface failure tolerance, the goal of composing high-assurance systems from COTS components will remain unsatisfied.

There are three broad classes of techniques for demonstrating failure tolerance of component-based systems:

Composable/analytical: These techniques construct models of systems based on the failure dependency relationship between components. Then formulae are used to mathematically estimate the failure tolerance of the resulting system based on failure rates of individual components. (These techniques usually assume that components fail independently.)

System-level: These techniques analyze the complete system as an entity composed of sub-entities and employ techniques such as system-level testing and fault-injection.

Formal: These techniques prove both properties about the composition that must be true and properties about the individual components that must be true (including output behaviors that sometimes cannot be observed).

Of these three broad categories, our effort is classified as a **system-level** technique for analyzing robustness of systems composed of COTS software components.

Software components can be combined in series and parallel just as mechanical systems are. The most common way to combine components (whether physical or software) is to connect them one after another in a series. When physical components are connected in a series, the quality of the whole is necessarily less than or equal to that of the worst component (*e.g.*, a chain is only as strong as its weakest link). However, in the case of software, this may not be true — sometimes the interactions between two or more faulty components may actually make the system more reliable (since faults may have the effect canceling each other out).

Another way of composing systems from components involves connecting components in parallel. Parallel component construction of physical systems is employed as a mechanism for increasing overall quality through redundancy (*e.g.*, two chains are better than one). As in the serial case, when the parallel paradigm is applied to software, the expected increase in quality through redundancy may not occur — the two parallel components may each break in the same way. It is apparent that quality-based arguments for designing from components in engineering physical systems cannot be straightforwardly co-opted for use in software systems. Instead, another approach is required.

For this reason, it is correct to assume that hardware engineering reliability theories are not applicable to component-based failure tolerance models. Our approach side-steps the standard series and parallel concerns, and instead focuses on the quality of a system's interfaces.[1] By not looking at robustness solely as a dependability problem, we are able to explicitly ignore the series/parallel construction of components. The results of our interface-based analysis technique are only implicitly dependent on the series/parallel construction of component's interconnectivity.

Our approach is based on a fault-injection method for measuring software attributes such as software testability, safety, and information system vulnerability [4]. Unlike previous implementations of this method, our solution here does not assume access to component source-code. As a result, one key benefit of our technique is that it will analyze the failure tolerance of legacy systems that may be composed of COTS software components. Our technique also provides a prediction of future failure tolerance in case additional COTS components are added to legacy systems. We can provide this prediction without requiring formal semantics for the components or for the system as a whole. Because of this characteristic, our technique can be applied to existing systems that often do not have formal specifications — specifications that often require extensive analysis and/or re-engineering to develop.

We contend that system-wide failure tolerance can be improved through the use of high-quality, robust interfaces. As an illustration, suppose that a system has two existing components, `A` and `B`, connected in series. Suppose further that in the environments where these components were used in the past, they demonstrated high quality. The concerns for the quality of a new composition of the two components will be a combination of:

- the quality of the interfaces between `A` and `B`, and

[1] Global parameters are considered as a part of the interface in our model.

- a determination as to whether the failure of the first component executed (`A`) is likely to cause failure of the second component (`B`) (which in this case would result in a failure of the composite).

2 Interface Propagation Analysis

System integrators are often surprised when a system behaves incorrectly even though previous unit testing had indicated that the components were all independently reliable. This disappointing problem is a direct result of the fact that the testing environment is often different from the eventual integration and user environment. Many times, developers have expectations for software that are unrealistically high. That is, they expect software to work correctly in *all* environments if they show it works correctly in *one*. A similar expectation would never be applied in the case of hardware. For example, we would not expect a computer motherboard to work correctly while immersed in a liquid sodium bath just because it worked correctly in normal atmospheric conditions. Apparently our expectations for software are in need of radical adjustment. We need to more carefully consider the environment in which a software component runs. The environment of a software component encompasses the distribution of inputs that the interface deliver. The environment can also include aspects of the machine, operating system, database, etc., however in this paper we will focus solely on the input information that the interfaces delivers. In addition, a hypothetical test distribution may be dramatically different from the actual use distribution.

Our approach, which we will call Interface Propagation Analysis (IPA), will be applicable to complete computer systems, including interfaces between the hardware and software, as well as interfaces between the different system layers. This new approach involves observing how corruption propagates *across* components and *within* components. This is based on the fact that all corrupt information must originate either: (1) in a component, (2) at an interface between components, or (3) at an interface between a component and the external environment.

In our approach, "propagation across" a component will be analyzed first by applying fault-injection mechanisms termed "perturbation functions" (See [3]) to interfaces feeding into components and then by analyzing component exit interfaces and determining whether the corruption propagated. It is possible to determine this information for any particular component without recourse to knowledge about other system components. Our basic algorithm for finding the proportion of times that a corruption crosses a component C is shown below:

1. Set temporary variable `count` to 0.

2. Randomly select an incoming parameter set for C from the distribution of data values that are likely to enter it. Global parameters make up part of this data tuple.

3. Perturb the sampled value of some variable a in this data tuple if the value of a has already been defined, else set a to a random value. Execute the component on both the perturbed data set and the original data set. (This can be done either synchronously in parallel or in serial order.)

4. Determine if the outcomes from executing with the perturbed data set and executing with the unperturbed original data set are different. If so, increment `count`. Note that `count` may also be incremented if an infinite loop occurs and the component does not terminate in either of the two cases. (We suggest simply setting a time limit for termination. If execution is not finished within the given time interval, we can safely assume that an infinite loop has occurred.)

5. Repeat algorithm steps 2–4 n times.

6. Divide count by $n - 1$ yielding the proportion of times that propagation crosses C' when C' receives corrupt data.

This algorithm can be applied to both source and executable components. The quantity derived by this algorithm is a function of three things: component C, the manner by which the corruption occurs, and the distribution of incoming data tuples that can enter the component. The final quantitative measure will also be dependent on the environment that surrounds C.

2.1 Propagation with Source Code

The central ideas of our approach here have origins in Sensitivity Analysis (SA) [3, 4, 1]. SA is a source-code–based technique that performs fault-injection on source-code constructs such as conditional statements, assignment statements, etc.

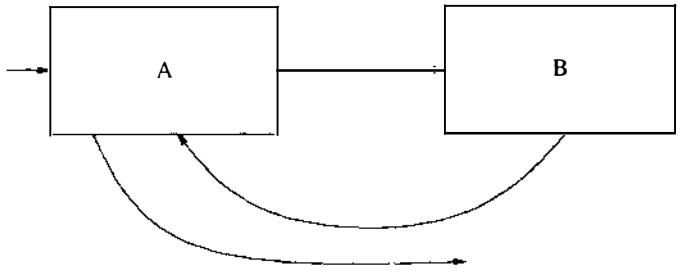


Figure A

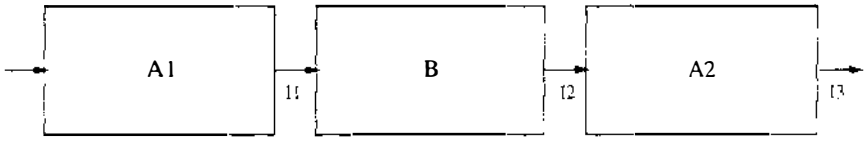


Figure B

Figure 1: Two distinct ways of understanding component connectivity. In both of these examples, component A calls component B, after which A uses B's results to do further computation.

We will begin with a specific example to illustrate our approach for studying "propagation across." For this example, we will assume the existence of component source code. Figure 1A shows two components (A and B) where A calls B, and B passes back some result to A, and A sends output to the external environment or another component. In Figure 1B, a different representation of the same connectivity is shown: here, A is partitioned into two sub-components — those instructions executed before the call to B are placed in a new component A1, while those instructions of A that are executed after the call to B are placed in another new component, A2. By viewing the problem in this manner, we can easily isolate the *two distinct* interfaces that pass information in and out of B, namely I1 and I2, as well as the one interface (I3) that passes the results from A2 to the next component or environment (not shown). Isolating the interfaces and considering their *individual failure tolerances* is essential to the goal of creating failure-tolerant systems. If I1, I2, and I3 are robust enough, this will ensure that the system created by their composition produces acceptable (if not optimal) output *even if the code in* A *or* B *is faulty.*

Suppose that the boxes in Figure 1A represent the following code segment:

```
Function A  (var1, var2, var3, .....);
    :
    [additional code]
    :
    B(param1, param2, param3);
    :
    :
    Return(param3);
End.
```

Further suppose that A and B are correct components, with one caveat in the form of a bug: the call to B inside A should take parameters B(param1, param4, param3) instead of B(param1, param2, param3) as shown. The existence of the bug means that the environment in which B must operate in the actual system (with the incorrect call) is likely to be greatly different than the environment it was coded for (*i.e.*, B's programmers expected it to operate with different inputs than it is actually getting). This faulty interface may be difficult to detect during integration testing, and may not be caught during unit testing of A because B will likely be stubbed out (*i.e.*, assumed to return correct information). However, by corrupting the data in param2 via interface propagation analysis, and assuming that param2 does not equal param4, we can observe what will happen to the composite if information carried by second parameter (from A to B) becomes corrupted.

2.2 "Propagation across" for Executable Code

Now, we will relax the assumption that we have access to source code. Without this, SA will require theoretical adjustments and implementation adjustments in order to be applied to large-scale COTS systems like the ones we are interested in. These adjustments were the main challenges of this project. Of critical concern is the addition of mechanisms that can handle message passing between components.

The problem of assessing the impact of using COTS components in integrated systems has only become a concern within the last 5 years, as the expense of customized systems has sky-rocketed. Our approach separates the problem of fault propagation in a system of COTS components into two subproblems: (1) propagation of corruption across a COTS component, and (2) intra-component propagation that affects the output of a COTS component. Here in Section 2.2, we discuss "propagation across"; Section 2.3 discusses the difficulty of assessing intra-component propagation for executable components.

Figure 2: System S composed of two COTS components in a series.

To explain our approach to COTS systems, consider the system in Figure 2. Figure 2 represents a legacy system, S, comprised of two COTS components in a series. To implement the information-flow shown in Figure 2, suppose that there exists a main program that is composed of a recursive-like series of calls where A returns a result to B.

```
Program
    B(A);
```

```
End.
```

Suppose, however, that a new executable component, D, is needed in S between components A and B, as shown in Figure 3. The code for S is now:

```
Program
    B(D(A));
End.
```

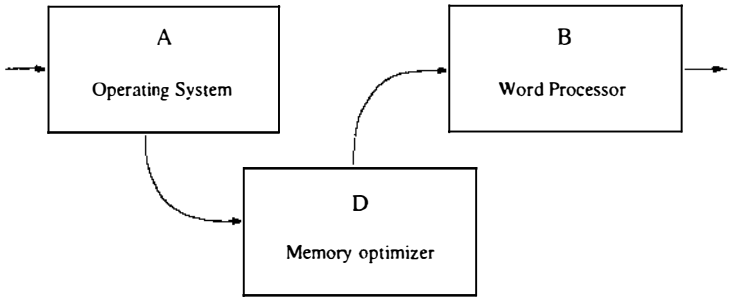


Figure 3: System S now composed of three COTS components in a series.

The designer of S will certainly want to know whether D will seamlessly integrate into S. To discover whether or not this is the case, we will answer the following questions:

Question 1: If D is faulty, will its incorrectness propagate and affect the functionality of S?

Question 2: If D is correct, but the interface between A and D is faulty, will that propagate and affect the functionality of S?

Question 3: If D is correct, but the interface between B and D is faulty, will that propagate and affect the functionality of S?

Question 4: Might D exacerbate any existing problems in A or B that have thus far escaped notice since they did not noticeably degrade S's functionality?

To answer Question 1, IPA will inject simulated faults (representing possible faults in D) into the output of D serving as B's parameter, by instrumenting the main code as follows: B(**fault-inject**$_1$(D(A))). To answer Questions 2 and 3, IPA will simulate incorrect interfaces between A and D as well as between between D and B (including such problems such as incorrect parameter orderings and incorrect parameters) by instrumenting the main code as follows: B(D(**fault-inject**$_2$(A))) and B(**fault-inject**$_3$(D(A))). Note that the fault injection functions which answer Question 1 and Question 3 differ in which parameters are tweaked. To answer Question 4, IPA will attempt various combinations of events, using code such as: B(**fault-inject**$_4$(D(**fault-inject**$_4$(A)))). Once partial answers to all four questions have been established, we gain insight into how likely it is that our modified S will continue to perform satisfactorily after the addition of D.

2.3 Intra-component Propagation for Executable Code

Intra-component fault propagation assesses the likelihood that a component will spontaneously produce corrupted output even when the component did not receive corrupt input information via its input interfaces. For source code components, this issue has been solved, as existing (and commercialized) techniques to assess this behavior have been developed. However, fault-injection

techniques for executable components that would help predict intra-component propagation do not yet exist.

Our failure tolerance metric for "propagation across" is based on *always* corrupting some portion of the incoming interface in order to compute the proportion of times that the outgoing interface of the latter component reflects this corruption. For an operational environment, this assumption may produce lower predictions of robustness. In reality, the proportion of times that an incoming interface is actually corrupt will often be less than 100%. To more accurately determine the fault-propagation probability of a system composed of some component `A` and its predecessor components, we need to assess the intra-component fault propagation for `A`'s predecessors. This information can then be combined with the "propagation across" information for `A` to accurately determine the propagation probability for the composite.

3 Implementing Interface Propagation Analysis

We now describe plans for how `B(fault-inject`$_1$`(D(A)))` and `B(D(fault-inject`$_2$`(A)))` will be implemented.

Perturbation functions change *data* that the program has access to, without changing the code itself. A perturbation function, `flipBit`, will be employed that allows a user to flip any memory bit (or bits) that an interface passes, from 0 to 1 or vice-versa. The first argument to function `flipBit` is the original value that we wish to corrupt. The second argument is the bit to be flipped (we assume little-endian notation). The corruptions generated by `flipBit` simulate programmer flaws, human factor errors, design flaws, and corruptions coming in to a component from other external sources.

The function `flipBit` can be written in C as shown below and then linked with the executable. NOTE: The `^` represents the `XOR` operation in C and the `<<` operator represents a `SHIFT-LEFT` of `y` positions. Also, `~` represents the negation operator.

```
void flipBit(int *var, int y)
{
   *var = *var ^ (1 << y));
}
```

`flipBit` can be used to model various kinds of program state corruptions, including:

- all bits high:
```
void allBitsHigh(int *var)
{
        *var = ((int)~0);
}
```
- all bits low:
```
void allBitsLow(int *var)
{
        *var = ((int)0);
}
```
- random bit patterns:
```
void randomPattern(int *var)
{
        *var = lrand48();  /* where lrand48() returns */
                           /* a pseudo-random integer */
}
```

- off-by-one:

```
void offByOne(int *var)
{
        if (lrand48() & 1)  /* add 1 if random integer is odd */
           *var = *var + 1;
        else                /* otherwise subtract one.        */
           *var = *var - 1;
}
```

- n random bits flipped, ($n \geq 1$):

```
void flipNbits(int *var, int n)
{
        int bits = 0;      int bitPos = 1;
        int i,j,k;         int xbit;

        for (i = 0; i < n; i++)    /* set n bits in the integer "bits" */
        {
           bits |= bitPos;
           bitPos <<= 1;
        }
        for (j = 0; j < sizeof(int) * 8; j++)     /* shuffle the bits  */
        {
           xbit = lrand48() % (sizeof(int) * 8);
           if ((!!(bits & (1 << xbit))) != (!!(bits & (1 << j))))
           {
              flipBit(&bits, xbit);
              flipBit(&bits, j);
           }
        }
        for (k = 0; k < sizeof(int) * 8; k++)
           if (bits & (1 << k))
               flipBit(var, k);
}
```

4 Summary

By corrupting data at the interface level, we free ourselves from worrying about how the information may have become corrupt. We need not determine whether component A produced a wrong value, whether the parameters to and from A were right or wrong, or even whether some component (or user) may have fed bad information into A. Our approach simulates instances of these error classes. If interface testing suggests that operational deployment will result in composite failure when the second parameter is corrupt, for example, we immediately recognize the second parameter's importance to the composite.

Our methodology offers a scalable approach to assessing the robustness of large-scale systems based on COTS components. By starting from traditional source-code–based fault-injection techniques, we have outlined a means for analyzing interface failure tolerance for COTS-based systems. This methodology offers four distinct advantages:

Advantage 1 Once the decision has been made to integrate a new component into an existing system, it is prudent to try to determine what the possible effects of integration will be. Our

methodology employs fault-injection to answer fundamental questions about the integration of new components into both legacy systems and new systems. By attempting to answer these questions at the interface level, it may be possible to extend our technology toward predicting the impact of characteristics such as *common-mode failures*.

Advantage 2 The results of this research should significantly enhance the state-of-the-art in software quality measurement. Considering the relative weakness of software engineering theory by contrast to traditional engineering theories, it is clear that new rigorous approaches are required. Development of better analyses for interfaces may play a role in moving software development (based on software components) away from an art to a science.

Advantage 3 Our approach takes a *black-box* view of software components. This is vital if we are to develop a plausible component-based software engineering theory without falling back on source code analysis. Today's information systems are made up of many different layers, from the innermost kernel to the outermost application layer. If our methodology works as well as we expect, it should be able to predict interface vulnerability between application layers.

Advantage 4 Today's systems are increasingly object-based, with message-passing handling object communication. Our approach is applicable to this programming paradigm.

In summary, predictions of component-to-component interface propagation provide a portion of the information necessary to predict failure tolerance of large-scale information systems. Our methodology can help ensure that the all-important interfaces are robust.

Acknowledgements

This work has been partially support by DARPA Contract F30602-95-C-0282 and NIST Advanced Technology Program Cooperative Agreement 70NANB5H1160. The authors thank an anonymous reviewer and Lora Kassab for comments on earlier versions.

References

[1] M. Friedman and J. Voas. *Software Assessment: Reliability, Safety, Testability.* John Wiley and Sons, New York, 1995. ISBN 0471-01009-X.

[2] N.G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[3] J. Voas. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Eng.*, 18(8):717–727, August 1992.

[4] J. Voas and K. Miller. Software Testability: The New Verification. *IEEE Software*, 12(3):17–28, May 1995.

Investigations into ANSI C Source Code Portability Based on Experiences Porting LaserJet Firmware

Troy Pearse
Hewlett-Packard Company
Business LaserJet Division, Box 15
Boise, ID 83707
(208) 396-4557
tpearse@hpdmd48.boi.hp.com

Paul W. Oman
Software Engineering Test Lab
University of Idaho
Moscow, ID 83844-1010
(208) 885-7219
oman@cs.uidaho.edu

Abstract: Porting software systems across hardware and software environments is an economical way to extend your software product lines. Building and maintaining highly portable software gives you a competitive advantage by decreasing the effort to implement existing systems on new platforms, which decreases the product's time to market. Over the last several years Hewlett-Packard has ported firmware for LaserJet printers across several different compiler environments and hardware architectures. While porting LaserJet firmware to new target platforms we have experienced numerous instances where ANSI C code compiled and executed differently across the platforms. These compilation and implementation anomalies can be attributed to differences in the way the ANSI C compiler is constructed. This paper summarizes the results of investigations into source code portability, relative to the ANSI C Standard and compiler implementations of that standard, that were conducted by HP and the University of Idaho. We identify major sources of non-portable code and explain why code often fails to port between ANSI C compilers. The diagnostic capabilities of several industrial ANSI C compilers is reviewed and related to the quality of compiler implementation.

Keywords: software portability, software quality, software tools, testing & debugging, ANSI C

Biographical sketches:

Troy Pearse is a firmware engineer at Hewlett-Packard, Business LaserJet Division, in Boise, Idaho. He has been with Hewlett-Packard for over eight years, working on embedded software systems and peripheral architectures now embodied in the LaserJet line of laser printers. His most recent responsibilities include measurement and perfective maintenance on large software systems comprising approximately one million lines of code. Through industrial experience with these large, complex systems he has gained an appreciation for software tools that automate software quality assurance of C source code. He has B.S. and M.S. degrees in computer science. His research interests are automated software quality tools for gauging portability, maintainability, and reusability.

Paul W. Oman is an associate professor of computer science at the University of Idaho, and an independent software consultant who specializes in software quality analysis. He has 24 years of experience with computers and software development, and has published over 100 articles and reports covering his contract work for international corporations and government agencies developing software for power production, medical instrumentation, environmental control systems, and personnel tracking and utilization. He occupies the Hewlett-Packard College of Engineering Research Chair at the University of Idaho, where he serves as director of the Software Engineering Test Lab. He has a Ph.D. in computer science and is a member of the IEEE, IEEE Computer Society, and the ACM.

Investigations into ANSI C Source Code Portability Based on Experiences Porting LaserJet Firmware

by

Troy Pearse
Hewlett-Packard Company

Paul W. Oman
University of Idaho

Why Portability is Important

Porting software systems to new platforms (hardware and software environments) can be an economical way to extend your software products to a larger market audience [Leinfuss93, Jaeschke89, Plauger86]. Having highly portable software is a competitive advantage because it decreases the effort to implement existing systems on new microprocessors and their associated compilers. This decreases the product's time to market, further increasing your competitive advantage [Ryan92]. Portability is especially crucial when dealing with embedded systems, such as Hewlett-Packard's line of LaserJet printers. Being able to quickly port LaserJet firmware to microprocessors with higher performance and lower cost is one of the reasons HP LaserJets have continued to be so successful.

While porting LaserJet firmware to new target platforms we have experienced instances where code that previously compiled cleanly on several ANSI C compilers would (1) fail to compile on the target ANSI C compiler, (2) execute differently across the platforms, and (3) abort in compilation or execution on the new target platform. These compilation and implementation anomalies can be attributed to differences in the manner and quality in which the ANSI C compiler is constructed. This paper summarizes the results of investigations into source code portability, relative to the ANSI C Standard and various compiler implementations of that standard, that were conducted by the HP Business LaserJet Division and the Software Engineering Test Laboratory at the University of Idaho. We identify major sources of non-portable code and explain why code often fails to port between ANSI C compilers; including subtle details about the ANSI C Standard's requirement for error diagnostics, which is widely unknown to programmers. The diagnostic capabilities of several industrial ANSI C compilers -- specifically their ability to diagnose and implement non-portable constructs -- is reviewed and related to the quality of compiler implementation.

Porting Experiences

Over the last several years Hewlett-Packard has ported firmware for LaserJet printers to the LaserJet 5, 5L and 5Si printers on three different processor platforms: (1) Intel 960, (2) Motorola 68040, and (3) AMD 29K. Porting LaserJet firmware to these different printers has been an economical way to produce a compatible family of LaserJet printers and an effective way to decrease the time to market for each printer. Using different processor platforms for each printer was important to achieve the price and performance needed for each individual LaserJet printer.

During the porting process, a recurring problem we experienced was source code compilation failure; where a C file which compiled and executed correctly on one processor platform failed to compile on a different platform. These compilation failures were not isolated occurrences, and varied considerably between each compiler we used. There were many cases of C files which compiled cleanly or with a warning on one compiler, but failed to compile on other compilers. And there were other cases where C files which had compiler warnings on one compiler translated cleanly on other compilers with no diagnostic messages. Something we couldn't explain was why these different ANSI C compilers were so different in what they diagnosed as an error or a warning.

Compiler manuals offered little help in identifying what constitutes an error diagnostic, and what separates it from a warning diagnostic. For example, the HP cc compiler manual differentiates between warnings and errors simply by stating:

> *"When the intent of a construct seems clear and unambiguous, the compiler emits a warning and attempts to generate code....When a construct is considered ambiguous, the compiler does not generate code" [HPC94].*

A few of the compile-time porting failures we experienced were documented in various portability books on the C language, such as [Spuler94, Rabinowitz90, Jaeschke89, and Koenig88]. But as a whole, there were many compilation porting failures which were not easily explained. And as we continued to develop and port LaserJet firmware to different compilers, we continued to experience similar compile-time porting failures.

In 1995, HP began a joint investigation into portability with the Software Engineering Test Lab, at the University of Idaho. The purpose of this investigation was to explain the differences in ANSI C compiler diagnostics we had experienced when porting LaserJet firmware, and to identify sources of source code porting problems. Our investigation began with a review of the ANSI C Standard for information concerning diagnostics and non-portable code constructs.

The ANSI C Standard and Portability

The ANSI C Standard was published in 1989, after six years of work by the ANSI X3J11 committee and adopted as a joint ISO standard in 1990 [ANSI/ISO90, Plum87, Plum91]. ANSI C is based on the C programming language as defined by Kernighan and Ritchie [Kernighan88]. The rationale for the standard was to codify existing C constructs and to promote portability:

> *"The Committee's overall goal was to develop a clear, consistent, and unambiguous Standard for the C programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environments [ANSI90]."*

The standard specifies how C programs which conform strictly to the standard should be translated. It defines explicit syntax rules and syntactic and semantic constraints for C programs, and requires translator implementations, such as a compiler, to diagnose any *erroneous* code which violates these syntax rules and constraints [Plauger90, ANSI90]. While it requires compilers to diagnose *erroneous* code, there is no requirement to classify the diagnostic as an error and/or to stop translation. Consequently, individual *compiler implementations* may choose

to translate invalid programs and only issue a warning diagnostic [ANSI/ISO90]. One possible explanation why a compiler would want to diagnose *erroneous* code as a warning and continue translation is backwards compatibility. For example, in K&R C it is allowable to leave out the semicolon after the last member of a structure, or to have a dangling comma after the last member of an enumeration. Although technically these are syntax errors in ANSI C, some compilers still allow them and only issue a warning in order to continue translation of existing K&R C programs [ANSI90, Plum87].

The ANSI C Standard does allow compilers to provide additional diagnostics, which can be warnings or errors. When creating the standard the ANSI X3J11 committee decided that diagnosis of program behaviors, beyond detecting *erroneous* conditions, was considered "an issue of quality of implementation, and that market forces would encourage more useful diagnostics" [ANSI90]. Another reason for not requiring additional diagnostics was that the committee thought many non-portable program behaviors were too difficult for compilers to detect for them to be required [Plauger90, Saks90]. Thus each compiler implementation defines its own set of additional diagnostics.

The back of the ANSI C Standard includes a set of "Annexes" or appendices. Annex G pertains to "Portability Issues" and contains a list of non-portable program behaviors categorized as: (1) *undefined*, (2)*implementation-defined*, and (3) *unspecified*. These program behaviors are non-portable because they are not explicitly defined by the standard and C compilers are not required to diagnose them. Creating the "Portability Issues" annex was also a mechanism used by the X3J11 committee to allow some degree of backwards compatibility to keep from invalidating existing compiler implementations and programs. By placing these non-portable code behaviors into an annex, programs depending on those items were still allowed to conform to the standard. However if a program contains any of the non-portable behaviors listed in Annex G, then it is restricted from what the standard defines as *strictly conforming* [ANSI90, Plum87].

From our review of the ANSI C Standard we learned the following facts about ANSI C and compiler diagnostics:

1. Compilers are required to diagnose *erroneous* code which violates ANSI defined *syntax* and *constraint* rules.
2. Compilers may translate source programs which contain *erroneous* code and issue only a warning diagnostic.
3. Compilers may implement additional diagnostics at their discretion, and can decide which are errors and which are warnings.
4. ANSI does not specify behavior for the non-portable program constructs located in Annex G, "Portability Issues."
5. Compilers are not required to diagnose the non-portable constructs found in Annex G.
6. Programs which use non-portable constructs from Annex G are excluded from being *strictly conforming*.

Compiler Diagnostics Study

In an earlier study of adaptive and preventative maintenance activities [Pearse95], we found that changing source code to eliminate warnings from one compiler made it easier to port that code to another compiler. But it was not clear as to why the compiler diagnostics differed across compiler environments. To better understand the differences in compiler diagnostic implementation (which we had experienced while porting LaserJet firmware) we conducted an investigation into the diagnostic capabilities of five different commercially available ANSI C compilers[1]. Our objective was to tabulate and compare the diagnostic capability of the compilers, and to identify sources of compile-time porting failures.

Based on descriptions of compiler errors and warnings found in several compiler manuals we created a set of 220 small test programs, each containing a different violation of the C language. We compiled these test programs on five ANSI C compilers and tabulated each compiler's diagnostic output into a compiler diagnostics matrix. This matrix shows a large amount of variation between the five ANSI C compilers' diagnostics on the 220 test cases, both in the different numbers of cases diagnosed, and how each test program was classified as an error, warning, or not diagnosed at all. A portion of this matrix which emphasizes the difference between the five different compiler diagnostic implementations is shown in Table 1. Since many of our test programs contained syntactically correct code which was non-portable or suspicious, a *compiler implementation* is not required to diagnose them because they are not technically *erroneous*. However, all of the test programs we wrote were diagnosed as a compiler error or a warning by at least one of the five compilers.

Violation Description	Compiler Diagnostic Error/Warning/None				
Compiler:	1	2	3	4	5
Function defined with register storage class.	W	E	E	E	E
Bitfield size exceeds width of type.	E	W	W	E	E
Value of constant exceeds range for type.	W	N	N	E	E
Integer division by zero.	W	N	N	E	E
Two declarations of an object specify incompatible type.	E	W	W	E	W

Table 1: Diagnostics Differences

Of the 220 test programs created in our investigation, 76 of them failed to port to at least one of the five ANSI C compilers (where a porting failure is defined as a file which compiles with an error on at least one of the compilers and without an error on at least one of the other compilers). We analyzed these 76 programs which did not port across all five compilers for common sources of failures and were able to relate over 50% of the compile-time failures to ANSI C *undefined behaviors* located in Annex G, Portability Issues, of the ANSI C Standard.

[1] Identity of these compilers is Hewlett-Packard confidential.

Other sources of compile-time failures we identified were related to *implementation-defined behaviors*, also located in Annex G, and inconsistent diagnosis of *erroneous code* [Pearse96].

ANSI C Undefined Behavior Study

Since a large number of the compile-time porting failures from our compiler diagnostics study were related to ANSI C *undefined behaviors*, we decided to conduct an in-depth investigation into how compilers diagnose ANSI C *undefined behaviors*. To be able to compare different *compiler implementations* and their ability to detect *undefined behaviors*, we created a complete set of test cases for all of 97 cases of *undefined behavior* enumerated in the ANSI C Standard Annex G.2. One test program for each *undefined behavior* was constructed[2].

Writing code for these *undefined behavior* test cases was problematic because it was often difficult to understand the exact condition being attempted, and because the C language is very powerful, difficult to master, and has many dark and dusty corners which can be unintentionally misused. As we wrote more and more test cases for *undefined behaviors*, it became clear to us that expecting programmers to knowingly avoid all *undefined behaviors* was not realistic, because of the depth and breadth of C language knowledge required to do so. Another problem we encountered was verification that a test program contained an *undefined behavior*. It was difficult to determine whether or not a test program contained an *undefined behavior* because compilers are not required to diagnose *undefined behaviors*, and we found that several state-of-the-art lint-like static analyzers we used during test-program development also do not catch many of the *undefined behaviors*.

The following are detailed examples of four *undefined behavior* test programs, including a description of when a programmer might inadvertently use the behavior, and a figure containing: the test case number, a description of the *undefined behavior* taken from the ANSI C Standard, the source code for the *undefined behavior* test, diagnostic output from the five ANSI C compilers, and program output from running the test case on two of the five ANSI C compilers. In addition to variation in compiler diagnostics, Figures 1 through 4 also show how even when a program containing an *undefined behavior* compiles without errors on different compilers, the program execution can be different.

1. Test Case for Undefined Behavior Number 19.
   Invalid arithmetic operations are undefined in ANSI C. We chose to implement this case by creating a divide by zero statement. To increase the ease of detection by compilers we divided by a constant value, and not a variable. Figure 1 shows that this simple *undefined behavior*, two of the test compilers issued an error diagnostic, one issued a warning diagnostic, and the other two failed to diagnose the problem. Further, the behavior of the code created by the compilers is shown to differ. Compiler One issues a warning but sets the result of the computation to zero, while compiler Two issues no diagnostic and aborts when the code is executed.

[2] Some cases of *undefined behavior* in Annex G.2 contained multiple instances. For these cases we only programmed one instance.

Undefined Behavior Case #	Undefined Behavior Description
udb19.c	"An arithmetic operation is invalid (such as division or modulus by 0) or produces a result that cannot be represented in the space provided (such as overflow or underflow) " [ANSI/ISO]

Undefined Behavior Code	Undefined Behavior Output

#include <stdio.h>
int main (void);
int main(void)
{
int i = 1;
i = 10/0;
printf("i = %d\n",i);
return 0;
}
	Compiler 1
	
	i = 0
	
	Compiler 2
	
	Floating exception (coredump)

Compiler Diagnostics for Undefined Behavior				
Compiler 1	Compiler 2	Compiler 3	Compiler 4	Compiler 5
Warning	No Diagnostic	No Diagnostic	Error	Error

Figure 1. Example Test Case for UDB #19

2. Test Case for Undefined Behavior Number 32.
Bitwise shift operators are often used by programmers as a low level execution efficiency optimization for multiplication and division by a power of two, and to shift and mask bit patterns [Rabinowitz90, Spuler94]. One such optimization is to shift all bits out of a variable in order to set it to zero. Since this optimization shifts the bits in the variable by the width of the variable, the result is undefined. Figure 2 shows that UDB #32 compiles without error (just warnings), but the result of shifting the bits is zero on compiler One and non-zero on compiler Two.

Undefined Behavior Case #	Undefined Behavior Description
udb32.c	"An expression is shifted by a negative number or by an amount greater than or equal to the width in bits of the expression being shifted" [ANSI/ISO]

Undefined Behavior Code	Undefined Behavior Output

#include <stdio.h>
int main(void);
unsigned int i = 0XFFFF;
int main(void)
{
printf("Before shift i= %X\n",i);
i = i >> (sizeof(i) * 8);
printf("After shift i= %X\n",i);
return 0;
}
	Compiler 1
	
	Before shift i= FFFF
	After shift i= 0
	
	Compiler 2
	
	Before shift i= FFFF
	After shift i= FFFF

Compiler Diagnostics for Undefined Behavior				
Compiler 1	Compiler 2	Compiler 3	Compiler 4	Compiler 5
Warning	Warning	Warning	No Diagnostic	Warning

Figure 2. Example Test Case for UDB #32

3. Test Case for Undefined Behavior Number 12.
String literals are like pre-constructed static character arrays, such *as "Hello World."* However, ANSI does not define where in memory string literals must be stored (RAM or ROM), or that each incidence of a string literal must have a unique location in memory. Thus string literals are often stored and pooled together in ROM, and are therefore more like *const* static character arrays. Programs should not modify string literals because, depending on the *compiler implementation*, a write to ROM could occur, or all instances of a string literal could be inadvertently changed from a write to a single string literal. Misuse of string literals is a common error because programmers misunderstand how to correctly use strings, plus the fact that K&R C had specified that all string literals were distinct. Misuse of string literals is easily avoided by using a character array instead of a string literal. For UDB #12 we modify a string literal at runtime to specify the UNIX terminal ID to be used, which is contained in a string literal [Jaeschke89, Rabinowitz90, Spuler94]. UDB #12 is not diagnosed by any of the five compilers. Figure 3 shows that compiler One results in code which modifies the string literal, while compiler Two generates code which fails during execution.

Undefined Behavior Case #	**Undefined Behavior Description**
udb12.c	"An attempt is made to modify a string literal of either form [ANSI/ISO].

Undefined Behavior Code

```
#include <stdio.h>
#include <string.h>
int main(void);
/* overwrites the two XX bytes with the terminal id,
03 */
int main()
{
  char *terminal = "/dev/ttyXX";
  printf("terminal = %s\n",&terminal[0]);
  strcpy (&terminal[8], "03");
  printf("terminal = %s\n", &terminal[0]);
  return 0;
}
```

Undefined Behavior Output

Compiler 1

```
terminal = /dev/ttyXX
terminal = /dev/tty03
```

Compiler 2

```
Execution failed (coredump)
```

Compiler Diagnostics for Undefined Behavior

Compiler 1	Compiler 2	Compiler 3	Compiler 4	Compiler 5
No Diagnostic	No Diagnostic	No Diagnostic	No Diagnostic	No Diagnostic

Figure 3. Example Test Case for UDB #12

4. Test Case for Undefined Behavior Number 09.
   C programs often manipulate and use the addresses of variables. However, if the variable is defined inside the scope of a function, then the contents of that local variable are undefined after the function finishes executing. Therefore any pointer variables that point to the local variable are also undefined after the function completes execution [Spuler94]. UDB #9 is not diagnosed by any of the five compilers, but the code that is generated differs between them. Figure 4 shows the differences in references to the automatic variable between the code generated from compilers One and Two.

Undefined Behavior Case #	**Undefined Behavior Description**
udb09.c	" The value stored in a pointer that referred to an object with automatic storage duration is used" [ANSI/ISO].

Undefined Behavior Code

```
#include <stdio.h>
int main(void);
void  callfunc(void);
void callfunc2 (int v);
int *i;

int main()
{
  callfunc();
  callfunc2(*i+10);
  printf("Value of *i after callfunc returned is %d
\n",*i);
  return 0;
}

void  callfunc(void)
{
  auto int *auto1;
  int j;
  auto1 = &j;
  *auto1 = 0xFF;
  i = auto1;
  printf("Value of *i in callfunc is %d \n",*i);
}

void callfunc2 (int v)
{
  v = v +1;
}
```

Undefined Behavior Output

Compiler 1

Value of *i in callfunc is 255
Value of *i after callfunc returned is 255

Compiler 2
Value of *i in callfunc is 255
Value of *i after callfunc returned is 0

Compiler Diagnostics for Undefined Behavior

Compiler 1	Compiler 2	Compiler 3	Compiler 4	Compiler 5
No Diagnostic	No Diagnostic	No Diagnostic	No Diagnostic	No Diagnostic

Figure 4. Example Test Case for UDB #9

We compiled all of the *undefined behavior* test cases on five different ANSI C compilers, and tabulated the compiler diagnostic output into an *undefined behavior* diagnostics matrix. Since our goal was to compare the five different compiler implementations' abilities to detect ANSI C *undefined behaviors,* when compiling our test cases we turned on all compiler flags relating to ANSI and diagnostics[3]. Table 2 shows a small portion of the total *undefined behavior* diagnostics matrix, where the variation between the five different compilers diagnostics is very apparent.

Test Case	Compiler Diagnostic Error/Warning/None				
	One	Two	Three	Four	Five
udb-a	N	N	N	N	N
udb-b	W	N	N	E	E
udb-c	N	N	N	N	N
udb-d	E	E	E	E	E
udb-e	W	W	W	E	E
udb-f	N	N	N	N	N
udb-g	E	E	E	E	E
udb-h	E	W	N	N	E
udb-i	N	N	N	N	N
udb-j	W	W	W	W	W
udb-k	E	W	W	E	N
udb-l	N	N	N	N	N
udb-m	W	N	N	N	E
udb-n	N	N	N	N	N
udb-o	W	E	E	E	E
udb-p	E	E	E	E	E
udb-q	N	N	N	E	N
udb-r	W	N	N	W	N
udb-s	W	W	W	N	N
udb-t	E	W	N	E	N

Table 2: Example Undefined Behavior Matrix

For the set of 20 *undefined behavior* test cases shown in Table 2, there are several test cases not diagnosed by any of the compilers. Further, none of the compilers diagnosed every test case. Each compiler diagnosed a different set of test cases, and the severity of diagnostics issued (error vs. warning) on most test cases differed between compilers[4].

Figure 5 shows the percentage of *undefined behavior* test cases diagnosed by each compiler, including the breakdown of the percentage diagnosed as errors and as warnings. Of the five compilers we tested, compiler Two diagnosed the highest percentage of test cases and the largest percentage of test cases as warnings. Compilers Four and Five diagnosed the fewest total number of *undefined behaviors.* Although compilers Four and Five caught the least number of *undefined behaviors,* they diagnosed over 90% of the ones they did catch as an error. The last pie chart of Figure 5 shows the number of test cases diagnosed by at least one of the five compilers. Combined, all five compilers diagnosed 50% of the *undefined behavior* test cases, which is more than any single compiler alone. This suggests that portability should increase when testing source code with multiple compilers; a hypothesis which was tested and verified in related studies [Pearse96].

[3] Compiler flags not related to diagnostics were not used, even though diagnostics for some compilers are only obtainable when compiling with certain optimization flags.

[4] The identity of the test cases, and the mapping of *undefined behaviors* to compilers is Hewlett-Packard confidential.

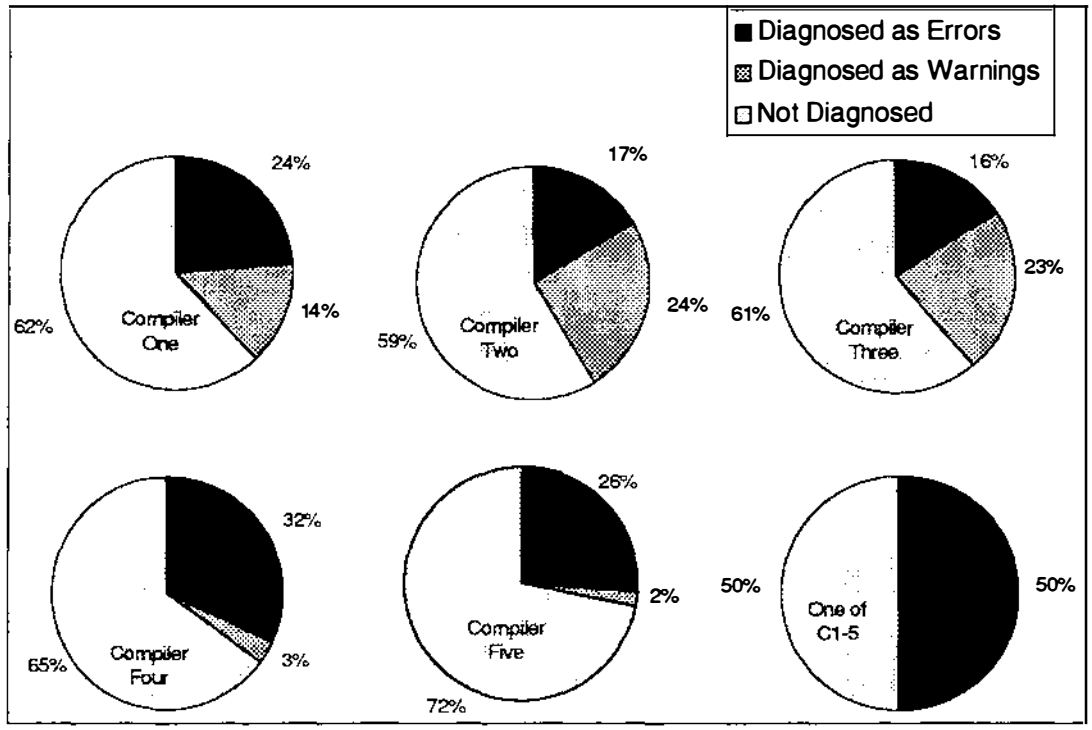


Figure 5. Percentage of Test Cases Diagnosed

We analyzed the *undefined behavior* diagnostics matrix for variation between compiler diagnosis of each test case. The first column of Table 3 shows the number of test cases which were diagnosed as an *error* by each of the compilers, and columns two through six show how those test cases were diagnosed by the other four compilers. For example, compiler Four diagnosed 31 of the *undefined behavior* test cases as an error, and compiler One diagnosed the same 31 test cases as 21 errors, 7 warnings, and 3 test cases which were not diagnosed.

Number Diagnosed As Errors		Diagnosed As Error Warning None By Compiler:														
		One			Two			Three			Four			Five		
Compiler:	E	E	W	N	E	W	N	E	W	N	E	W	N	E	W	N
One	23	-	-	-	13	8	2	12	7	4	21	0	2	16	0	7
Two	17	13	4	0	-	-	-	16	1	0	17	0	0	15	0	2
Three	16	12	4	0	16	0	0	-	-	-	16	0	0	15	0	1
Four	31	21	7	3	17	8	6	16	8	7	-	-	-	20	0	11
Five	25	16	9	0	15	6	4	15	5	5	20	1	4	-	-	-

Table 3: Compiler Diagnostics Comparison

The differences in which *undefined behavior* test cases are diagnosed by each compiler, and how they are diagnosed, emphasizes how compiler dependencies can cause porting problems

between compilers. For example, porting a program from compiler One to compiler Four would result in a compile time failure if the program contained any of the ten *undefined behaviors* diagnosed by compiler Four as an *Error*, but flagged as a *Warning* or *Not diagnosed* by compiler One. This relationship is shown in the leftmost highlighted cell in Table 3. The number of compile-time failures experienced when porting between two compilers can depend on the diagnostic abilities of the current compiler being used. For example, porting from compiler Three to compiler Four could potentially encounter 15 compilation failures related to *undefined behaviors* (8 warnings and 7 not diagnosed). But if you ported from compiler Four to compiler Three there would be no compile-time failures related to *undefined behaviors*. These latter two examples are also highlighted in Table 3.

Validation and Limitations

Through these two compiler studies we have recreated situations similar to our experiences porting LaserJet firmware where source code which compiles cleanly on one ANSI C compiler fails to compile on other ANSI C compilers. In addition, we have shown that code which contains ANSI C *undefined behaviors* is not portable between compilers. To validate that *undefined behaviors* were related to our LaserJet porting compilation failures, we used a lint-like tool and created an output filter to identify diagnostic messages related to 37 different *undefined behaviors*. Using this tool and filter we measured the number of *undefined behaviors* in a LaserJet subsystem consisting of approximately 25 C source files and 78 KLOC before and after porting that subsystem to several new computing environments. We found that as the subsystem was ported to the new computing environments *undefined behaviors* were necessarily removed to get that subsystem to function in the new environment. The net effect was a significant reduction in the number of *undefined behaviors* in the subsystem. This is evidence that *undefined behaviors* contributed to compilation problems experienced when porting LaserJet firmware.

Our set of 97 test cases for *undefined behaviors* was developed by a team of experienced C programmers, where each test case was an attempt to explicitly create a single *undefined behavior*. However, determining that each *undefined behavior* test case was an accurate implementation of the target *undefined behavior* described in the ANSI C Standard was difficult because the content of the diagnostic messages from the compilers and static analysis tools did not relate the message to the *undefined behaviors*. That is, we had to review each diagnostic message and determine if the diagnostic message was describing the *undefined behavior* we were trying to implement. Also, just because a compiler does not detect our version of an *undefined behavior* does not mean it wouldn't identify a slightly different version. For example, the test program shown for udb32.c (in Figure 2) shifted a variable by the width in bits of that variable. As written, the shift width is computed using the sizeof operator and is diagnosed by four of the five compilers. But if the test program is modified slightly to make the shift width a constant value instead of a computation, then all five compilers diagnose the problem.

Our validation efforts also measured the number of occurrences of *undefined behaviors* in a C program using a lint-like tool. However, the relationship between the *undefined behavior* test cases and the lint-like tool diagnostic messages may not be one-to-one, as the lint-like tool may group other error conditions into one of the same diagnostic messages issued for an *undefined*

behavior test case. This could potentially result in a larger number of *undefined behaviors* reported for a C program.

Conclusions and Recommendations

Creating the ANSI C Standard improved the overall portability of C programs, but it left many significant gaps which resulted in different implementations by each ANSI C compiler. This in turn has caused porting problems for programmers who are trying to write portable C programs because they may unintentionally use *undefined behaviors* in their programs. From our compiler diagnostic and *undefined behavior* studies we conclude that many of the problems we experienced when porting LaserJet firmware was related to these ambiguities in the ANSI C standard which allow: (1) *erroneous* behavior to be diagnosed as warnings, and (2) compilers to diagnose other non-portable constructs as errors, warning, or nothing at all. We also conclude that the use of any code which depends on *undefined behaviors* is a serious portability risk, because programs containing *undefined behaviors* may fail to be translated or fail to execute.

From our porting experiences and this portability investigation, we have the following recommendations for software developers and managers who need to write portable code.

1. Programmers should pay attention to compiler warnings, since many *undefined behaviors* were diagnosed as warnings. There were also many cases where a warning on one compiler resulted in a compiler error on another compiler. Of course this implies that programmers should turn on warnings when they compile.

2. Programmers and managers should consider the quality of the compiler and tools they use, as the quality of compiler can impact the portability of the code that is written.

3. Managers should emphasize portability as an important software quality attribute that can be useful when leveraging new products out of existing product lines. Software and firmware portability can increase market share and decrease time to market.

4. Programmers should identify static lint-like tools which can be used to write more portable code. This can take some effort, because of the volumes of output which most lint-like tools produce, and the fact that these tools do not identify specific diagnostics as an *undefined* or *implementation-defined* behavior.

5. Compiler and tool writers could greatly improve their existing diagnostic messages. Current diagnostic messages are confusing to programmers because they all appear equal. For example, compiler warning messages about benign program behaviors appear identical to warning messages about non-portable *undefined behaviors*. This grouping of unrelated warning messages discourages programmers from considering all compiler warnings as serious problems, and encourages programmers to only pay attention to compiler errors. If compilers would simply define and explain several categories of diagnostic messages in their user manual, and then include those categories in their diagnostic messages, then programmers could determine which diagnostic messages are important for their program.

References

[ANSI/ISO90] *American National Standard for Programming Languages -- C*, American National Standards Institute, NY, 1990.

[ANSI90] *Rationale for the ANSI C Programming Language*, Silicon Press, ANSI, 1990.

[HPC94] Hewlett-Packard, *HP C/HP-UX Reference Manual For HP 9000 Series 700/800 Computers*, Hewlett-Packard, CA, 1994.

[Jaeschke89] R. Jaeschke, *Portability And The 'C' Language*, Hayden Books, Indianapolis, Indiana, 1989.

[Kernighan88] B Kernighan and D Ritchie, *The C Programming Language - ANSI C Draft Version*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Koenig88] A. Koenig, *C Traps and Pitfalls*, Addison-Wesley, USA, 1988.

[Lienfuss93] E. Leinfuss, "Portability priorities drive developer picks", *Software Magazine*, October 1993, pp. 103-107.

[Pearse95] T. Pearse and P. Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities," *Proceedings 1995 International Conference on Software Maintenance*, (Oct. 16-20 Nice, France), IEEE Computer Society Press, 1995, pp. 295-303.

[Pearse96] T. Pearse, *A Study of Software Portability to Identify Tangible Characteristics of Portable Programs*, Masters Thesis, Dept. of Computer Science, University of Idaho, Moscow, ID 83843, 1996.

[Plauger86] P. J. Plauger, "The Economics of Portability", *The C Journal*, Winter 1986, pp. 22-24.

[Plauger90] Plauger P. J., "Setting the Standard", *Computer Language*, Vol. 7, No. 3, March 1990, pp. 17-25.

[Plum87] T. Plum, *Notes on The Draft C Standard*," Plum Hall Inc., 1987.

[Plum91] T. Plum, "Building a standard is hard; Testing it is just as difficult," *Computer Language*, Vol. 8, No. 5, May 1991, pp. 39-43.

[Rabinowitz90] H. Rabinowitz & C. Schaap, *Portable C*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[Ryan92] R. Ryan, "A moving target," *Byte*, January 1992, pp. 159-174.

[Saks90] D. Saks, "C and the ANSI standard", *Computer Language*, Vol. 7, No. 3, May 1990, pp. 65-79.

[Spuler94] Spuler, D. A., *C++ and C Debugging, Testing and Reliability*, Prentice-Hall, 1994.

NEW AND IMPROVED DOCUMENTATION PROCESS MODEL

Curtis R. Cook
Computer Science Department
Oregon State University
Corvallis, Oregon 97331-3202

Marcello Visconti
Computer Science Department
Universidad Santa Maria
Valpraiso, CHILE

ABSTRACT

Inaccurate, missing, incomplete, or out of data documentation is a major contributor to poor software quality. A four level Software System Documentation Process Maturity Model and assessment procedure developed by the authors is described in [7]. This paper briefly reports the results of 26 assessments of software projects at 7 different organizations. These assessments have yielded promising results plus informative comments and suggestions about the assessment process. The major part of this paper describes improvements to the assessment process based on these comments and suggestions. It also includes the results of 25 assessments of software projects at 7 different organizations using the new process.

Keywords: Documentation, process maturity model, assessment, key practices.

Curtis R. Cook is professor of computer science at Oregon State University. He received his Ph.D. in computer science from the University of Iowa. Professor Cook has over fifteen years of research and experience in the software complexity metrics and software quality areas. He has over 50 journal publications and conference presentations. He has taught workshops on software complexity metrics and software quality. He is a member of the editorial board of the Software Quality Journal and has served on the program committee for several software quality conferences.

Marcello Visconti is a professor of computer science at Universidad Santa Maria in Chile. He has a Masters' Degree from Universidad Santa Maria in Chile and a Ph.D. in computer science from Oregon State University. His research interests are in software quality and software engineering.

1. INTRODUCTION

A fundamental goal of software engineering is to produce the best possible working software along with the best supporting documentation. Empirical data shows that software documentation products and processes are key components of software quality [1, 4, 6]. These studies show that poor quality, out of date, or missing documentation is a major cause of errors in software development and maintenance. For example, the majority of defects discovered during integration testing are design and requirements defects, e.g. defects in documentation that were introduced before any code was written.

One solution to the documentation problem is to improve the documentation process. Visconti and Cook [7] describe a four level software system documentation process maturity model and assessment process. The model was influenced by the SEI Capability Maturity Model [3, 5] in that key process areas, practices, indicators, and challenges are defined for each of the four levels. An assessment questionnaire that takes only 30 minutes to complete assesses an organization's software system documentation process relative to the model. An assessment report giving the maturity level and documentation process profile is generated from the questionnaire responses. The profile indicates what practices the organization is doing well, what practices need improvement, and challenges to move to the next higher maturity level.

Cook and Visconti [2] reports the results of fourteen project assessments for five different organizations. Although only four out of the fourteen projects classified defect data by development phase, there was one promising data point that supported the model. Three of the four projects were from the same organization. Two projects were assessed as a level 2 and one a level 1. The defect data indicated that for the one project at level 1, 41% of the defects found during integration testing were design and requirements defects whereas for the two projects at level 2 only 25% of the defects found were design and requirements defects. This strongly suggests that defects are detected earlier in projects with a higher documentation maturity level.

This paper focuses on describing improvements of the assessment process and reports the assessment results for 25 software projects at 7 different organizations using the new version of the assessment process. Personnel from projects assessed using the original version provided comments and suggestions about the assessment questionnaire and assessment report. This paper describes changes to the questionnaire, key practices, and the scoring scheme based on this feedback. For example, the questionnaire was revised to eliminate ambiguity and to obtain more specific information about key practices. One key practice was dropped

and two added. Also the degree of satisfaction for key practices was expanded from a three to a five grade scale to provide a more accurate and informative assessment. A comparison of the same projects scored under the old and new system showed the advantages of the new scoring system.

One concern from the original assessments was the maturity level. When the assessment report was presented, we were surprised that the project teams seemed to focus on the maturity level and ignored the detailed information about key practices and the action plan in the report. Also the relation between the maturity level and key practices was not clear. The assessments using the new version of the assessment procedure provided the opportunity to compare various methods of computing the maturity level that related the maturity level more closely to key practices.

2. DOCUMENTATION PROCESS MODEL AND ASSESSMENT PROCEDURE

A solution to the problem of poor quality, out of date or missing documentation is to improve the documentation process. The idea behind the Software System Documentation Process Maturity Model (SSDPMM) is very simple: Most defects discovered during software testing are documentation defects, e.g. requirements and design defects. Hence improving the documentation process will have the most impact on improving software quality.

The SSDPMM is briefly described in this section along with the procedure for assessing an organization's current documentation process. For a more complete description of the model and assessment procedure see Visconti and Cook [8]. It is important to keep in mind that documentation refers to system documentation generated during software development and not to end-user documentation.

The SSPDMM was influenced by the SEI's Software Process and Capability Maturity models [3, 5]. The Software Engineering Institute (SEI) was established by the government to address the problem of accessing the capability of DOD contractors to deliver software on time and within budget. An approach it took was to develop software process maturity models such as the Capability Maturity Model (CMM) that provide a framework for improving quality [3]. The CMM is a five level model of the software development process where each level has key processes areas associated with it. An assessment process gauges the software development organization's maturity level (capability) by measuring the degree to which the key processes are defined and managed.

The SSDPMM is a four level model of the documentation process. A summary of the SSDPMM is given in Table 1. The four column headings

are the names for the four levels. The rows give the key process areas, practices, indicators, and challenges to move to the next maturity level. The succinct descriptions capture the essence of the four levels. The first level (ad-hoc) is the lowest level and reflects a chaotic process. To be at the second level the organization must have some mechanism, such as a check-off list, for determining whether all of the required documentation has been generated. To be at the third level there must be a means of assessing the quality and usefulness of the documentation. At the fourth level, there is some type of quality measurement program in place that provides feedback for continual improvement.

The purpose of the assessment procedure for a model is to determine where an organization stands relative to that model. For SSDPMM this means assessing an organization's documentation process relative to the model by assessing its capability with respect to the key process areas of the model. A questionnaire was developed for this purpose. There are key practices and indicators for each level of the model. The questions in the questionnaire attempt to ascertain which and to what extent these practices and indicators are part of the documentation process. For our model, the assessment maps an organization's experience and past performance to a documentation maturity level and generates a documentation process profile that indicates the maturity level and what key practices the organization is doing well, what practices need improvement, and challenges to move to the next level.

The assessment questionnaire takes only 30 minutes to complete. The questions have been derived directly from the model and the key practices and indicators for each level. Each practice and/or indicator defines one or more questions whose answers determine the degree of satisfaction of the practice/indicator. Thirty-five of the questions have answers in the range of 1 (never) to 5 (always). These questions attempt to determine how often a given action is performed. The remaining 21 questions have yes or no answers. A complete description and copy of the questionnaire is given in [7].

A Documentation Assessment Report is generated from the responses to the questionnaire. The Report contains an executive summary with the maturity level, a documentation process maturity profile and an action plan. The profile indicates satisfactory practices, practices needing improvement, missing practices, and challenges to move to the next level. The action plan describes specific actions to improve existing practices and to address missing practices so that the organization can move to the next level. A complete description and sample form of the report can be found in [8].

	Level 1 Ad-hoc	Level 2 Inconsistent	Level 3 Defined	Level 4 Controlled
Keywords	Chaos, Variability	Standards Check-off list Inconsistency	Product assessment Process definition	Process assessment Measurement Control Feedback Improvement
Succinct Description	Documentation not a high priority	Documentation recognized as important and must be done.	Documentation recognized as important and must be done well.	Documentation recognized as important and must be done well consistently
Key Process Areas	Ad-hoc process Not important	Inconsistent application of standards	Documentation quality assessment Documentation usefulness assurance Process definition	Process quality assessment and measures
Key Practices	Documentation not used	Check-off list Variable content	SQA-like teams for documentation quality and usefulness Consistent use of documentation tools	Minimum process measures Data collection and analysis Extensive use of documentation tools and integration with CASE tools
Key Indicators	Documentation missing or out of date	Standards established	SQA-like practices Consistent use of documentation tools	Data analysis and improvement mechanisms
Key Challenges	Establish documentation standards	Exercise quality control over content Assess documentation usefulness Specify process	Establish process measurement Incorporate control over process	Automate data collection and analysis Continually striving for optimization

Table 1. SSDPMM - Summary Table

3. ASSESSMENT RESULTS

Twenty-six project teams at seven different organizations were assessed This includes the 14 projects at five different organizations reported in [2]. See Table 2. Note that in the first column letters denote the organization and the LEVEL column gives the maturity level for the project. There were 111 persons total in these projects. Ten of the projects were at level 2 and the remaining 16 were at level 1. Note the "W" and "S" in the LEVEL column denote "Weak" and "Strong" respectively.

COMPANY-PROJECT	LEVEL
A-1	2 W
A-2	2
A-3	2 W
A-4	1
A-5	2 S
A-6	1
A-7	2
B-1	1
C-1	1
D-1	1
D-2	1
D-3	1
D-4	1
D-5	1
D-6	1
D-7	1
D-8	1 S
D-9	1
D-10	2
E-1	1
F-1	1
F-2	2 W
G-1	1 S
G-2	2 W
G-3	2 S
G-4	2

Table 2: Assessment Results for 26 projects

The main goal of the research is to show that organizations at a higher documentation process maturity level produce a higher quality product. Cook and Visconti [2] assessed 14 projects at 5 different organizations. They used requirements and design defects and the phase in which they were detected as a measures of quality. Their rationale was that most defects are documentation (requirements and design) defects and it is believed that high quality documentation will lead to early discovery of these defects. Hence a higher documentation maturity level software project will discover a larger percentage of these defects prior to integration testing than a software project at a lower documentation maturity level.

Although only four out of the fourteen projects classified defect data by development phase, there was one promising data point that supported the model. Three of the four projects were from the same organization. Two projects were at level 2 and one was a level 1. The defect data indicated that for the one project at level 1, 41% of the defects found during integration testing were design and requirements defects whereas for the two projects at level 2 only 25% of the defects found were design and requirements defects. This strongly suggests that defects are detected earlier in more mature projects.

Since this paper twelve additional assessments have been conducted. Two of these projects are at level 2 and the rest were at level 1. Unfortunately no defect data was available for these projects.

4. MODIFICATIONS TO THE ASSESSMENT PROCESS

This section reports modifications to the questionnaire, scoring scheme, and key practices based on feedback from software engineers and managers of project teams that were assessed.

4.1 CHANGES TO ASSESSMENT QUESTIONNAIRE

In the original questionnaire, 35 questions had answers in the range 1 through 5 and 21 had yes/no answers. The numbers in the range 1-5 represented how often a given action was performed: 1 = never, 2= seldom, 3= sometimes, 4= usually, and 5 = always. Several questions were in pairs where the first question asked if an action was done and the second question asked how frequently. The questions attempted to determine degree of satisfaction of 18 key practices.

Software engineers and managers pointed out some ambiguities in the questions and to the fact that they were unable to answer a few of the questions because they did not have the information or personal knowledge about that part of the software development process.

Ambiguous questions were reworded and/or questions were added for clarification. To address the second concern a "Don't Know" response choice was added to each question.

4.2 CHANGES TO KEY PRACTICES

The 18 Key Practices broken down by level are given in Table 3. Based on feedback one key practice was deleted and two key practices about required documentation were added. The key practice "Integrate CASE and documentation tools" was dropped because of its ambiguity and the difficulty constructing questions to determine the degree of satisfaction. The key practice "Use of check-off list of required documentation" seemed to be the most important single factor that distinguished level 1 and level 2 projects. (Note that this corresponds exactly to the model as all required documentation must be generated for a project to be at level 2.) But the only question related to it asked "There are check-off lists that indicate which software documents must be created." A positive response indicates the existence and use of such a check-off list. However, there may be several reasons for a negative response, some of which are symptomatic of more serious problems than others. A negative response may indicate that either (1) such a list exists,

Level	Key Practices
1	1. Consistent creation of basic software development documents 2. Documentation generally recognized as important
2	3. Written statement or policy about importance of documentation 4. Adequate time and resources for documentation 5. Adherence to documentation standards 6. Use of check-off list of required documentation 7. Use of simple documentation tools
3	8. Accuracy and reliability of documentation 9. Mechanisms to update documentation 10. Mechanisms to monitor quality of documentation 11. Methods to assess usefulness of documentation 12. Use of common sets of documentation tools 13. Use of advanced documentation tools 14. Documentation-related technology and training
4	15. Measures of documentation process quality 16. Analysis of documentation usage and usefulness 17. Process improvement feedback loop
	18. Integrate CASE and documentation tools

Table 3. 18 Key Practices

but it is not used; or (2) such a list does not exist. Reasons for the latter could be the nonexistence of a policy or standards defining which documents must be created or the policy or standards exist but there is no check-off list. This led to the addition of two key practices that attempt to determine if there is a documentation policy or standard that defines the required documentation ("Existence of documentation policy or standards") and whether or not the policy or standard is followed ("Adherence to documentation policy or standards"). Also the original key practice was changed to "Mechanism to check that required documentation is done". Questions related to the new key practices were added to the questionnaire. Thus as a result of all of these changes the number of questions in the new version of the questionnaire increased to 68.

4.3 SCORING REVISED

The scoring scheme for questions and key practices was modified to provide more information. Degree of satisfaction of key practices were derived from the responses to the questions related to that key practice and a key practice was rated as full satisfaction, partial satisfaction, or none. For example in the old scoring scheme, the degree of satisfaction for Key Practice 9 "Mechanisms to update documentation" is based on the responses to questions 8-14. If the average of each question with responses in the 1-5 range is above 3.0 and the majority of the responses to each yes-no question is yes, then the degree of satisfaction is "Full." If for questions 8, 10, 12, 14 the majority of responses to each questions is no, then the degree of satisfaction is "None." Otherwise it is "Partial."

In the old version questions with "yes/no" responses were scored on a majority basis. Thus if 6 respondents responded "yes" and 4 responded "no" the score for that question was "yes". However, if all 10 responded "yes" this was also scored as "yes". In the new version a "yes" was scored as a "5" and a "no" as a "1" and the score for the question was the average of the responses. Hence the score for 6 "yes" and 4 "no" is 3.4 and for all 10 "yes" is 5.0 which better reflects the consensus of the respondents.

Scoring for questions with answers in the range 1 (Never) to 5 (Always) was also simplified. In the old scoring scheme a high and low score were thrown out before computing the average. In the new scheme the score is merely the average of the "non-Don't know" responses. The net result of these scoring changes was a value between 1 and 5 for the responses to each question. This led to rating the degree of satisfaction of the key practices on an easily interpreted 5 grade system: very high (> 4.5), high (3.5 - 4.5), medium (2.5-3.5), low (1.5-2.5), and very low (< 1.5). Thus if the average of the scores of the responses to the questions related to a practice is 3.67, then the degree of satisfaction score for that key

practice is high and if it is 1.96 then is low. Furthermore, the five degrees of satisfaction (very high, high, medium, low, and very low) are more descriptive and less ambiguous than the full, partial, and none of the old scoring scheme.

5. ANALYSIS OF THE PREVIOUS ASSESSMENT RESULTS USING THE NEW SCORING SYSTEM

To test that the new scoring system provided more information and a more accurate picture of the process, the questionnaires and key practices from 26 projects were rescored using the new system. Table 4 presents the old and new scoring of the 18 key practices for four projects in the same organization. In the New Scoring column both the score and degree of satisfaction (VH = Very High, H = High, M = Medium, L = Low, VL = Very Low) are given.

Notice that generally the two scoring schemes give quite similar ratings of degrees of satisfaction but there are some differences and that the numerical scores provide a finer grain measure. All Partials in the old scoring become either Medium or High in the new scoring scheme. In all but two instances all Highs become either High or Very High. "None" in the old scoring scheme has the biggest range of values in the new scoring scheme as it may become Medium, Low, or Very Low. The numerical scores in the new scoring scheme may indicate difference even when all of the projects have the same degree of satisfaction grade. For instance, Key Practice 1 is rated High for all four projects. But notice that the score for Project 2 is 3.7 while the scores of the other three projects are 4.1 or 4.2. Also for Key Practice 5, Project 1 is rated High because its score is just above 3.5 while Projects 3 and 4 are rated Medium because their scores are just below 3.5 and Project 2 is rated Medium even though its score is 2.9. On the other hand, for Key Practice 4 all four projects are rated Medium and their numerical scores are between 3.1 and 3.3.

In analyzing the results, the key practice 6, "Use of check-off list of required documentation" seemed to be the strongest indicator of maturity level. The maturity levels of four projects (old scoring system) were strong level 1, weak level 2, level 2, and a strong level 2 while the scores for Key Practice 6 for the four projects were 2.9, 3.3, 3.7, and 3.9 respectively.

The manner in which to present the assessment information is a perplexing issue. When the assessment results were presented, groups seemed to focus on the maturity level. The degree to which the maturity level appeared to overshadow the key practices information surprised us.

	Project 1: G-1		Project 2: G-2		Project 3: G-4		Project 4: G-3	
Key Practices	Old Scoring	New Scoring	Old Scoring	New Scoring	Old Scoring	New Scoring	Old Scoring	New Scoring
1	Partial	4.1 H	Partial	3.7 H	Partial	4.2 H	Full	4.2 H
2	Full	3.5 H	Full	4.0 H	Full	3.3 M	Full	4.6 VH
3	None	2.8 M	None	3.0 M	Full	3.3 M	Full	4.1 H
4	Partial	3.1 M	Partial	3.2 M	Partial	3.3 M	Partial	3.2 M
5	Partial	3.5 H	None	2.9 M	Partial	3.5 M	Partial	3.5 M
6	None	2.9 M	Partial	3.3 M	Partial	3.7 H	Partial	3.9 H
7	Full	4.5 VH	Full	5.0 VH	Full	4.6 VH	Full	4.3 H
8	Partial	4.5 H	Partial	4.2 H	Partial	4.5 H	Full	4.6 VH
9	None	2.3 L	Partial	3.3 M	None	1.5 VL	None	2.3 L
10	None	2.2 L	None	2.0 L	None	2.3 L	None	3.0 M
11	None	1.0 VL	None	1.0 VL	None	1.0 VL	None	1.4 VL
12	Partial	3.1 M	Full	3.8 H	Partial	3.7 H	None	2.6 M
13	None	2.0 L	Partial	2.8 M	Partial	3.0 M	None	2.7 M
14	None	2.2 L	None	2.3 L	None	1.7 L	None	1.7 L
15	None	3.0 M	None	2.6 M	None	2.0 L	None	2.1 L
16	None	1.4 VL	None	1.1 VL	None	1.6 L	None	1.2 VL
17	None	2.5 L	None	2.5 M	None	1.7 L	None	2.2 L
18	None	1.0 VL	None	1.0 VL	None	1.0 VL	None	2.2 L

Table 4: Comparison of Key Practice scoring

A plausible reason for this may be comprehending one number is much easier than comprehending 18 numbers. The question then is how to direct more focus to the key practices as they are fundamental to any changes.

One approach is to consider the key practices for each level. The 18 key practices (Table 3) can be broken down by level. The averages of the key practice scores for each level then summarize the key practices by level. As an example Table 5 gives the four level averages for the four projects in Table 4. Recall that these four projects were assessed as a strong level 1, weak level 2, level 2, and a strong level 2 respectively. For the most part, the four level averages correspond to the relative maturity levels of the four projects. Project 1 has the lowest Level 2 and 3 averages and Project 4 has the highest Level 1 and 2 averages.

All of this suggests using the four level averages in computing the maturity level. One method of computing the maturity level is the highest level for which the average score of the key practices that comprise that level is greater than 3.50. Another method of computing the maturity level is to use the degree of satisfaction grade. In this case the maturity is the highest level for which all of the key practices that comprise that level are greater than 3.50 (e.g. High or Very High). A third alternative is to

define weak, solid, and strong maturity by the following: Maturity level is N if all level N key practices scores are at least 3.0. It is a Weak, Solid, or Strong level N if the average of the practices for this level is 3.0 or better, 3.5 or better, or 4.25 or better respectively. Finally an even more radical approach would be to merely present the averages for the 4 levels and not give a maturity level.

Key Practices	Project 1: G-1	Project 2: G-2	Project 3: G-4	Project 4: G-3
Level 1	3.80 H	3.90 H	3.73 H	4.40 H
Level 2	3.39 M	3.50 H	3.65 H	3.80 H
Level 3	2.47 L	2.80 M	2.52 M	2.60 M
Level 4	1.96 L	1.80 L	1.56 L	1.90 L

Table 5. Averages of Key Practices for each level.

6. ASSESSMENT RESULTS USING MODIFIED ASSESSMENT PROCEDURE

Twenty-five projects from seven organizations were assessed using the revised questionnaire, scoring method and key practices. For each project, Table 6 give the minimum and averages of the key practices for each level. One of these projects (Project J-1) was one (F-2 in Table 2) that had been assessed as a weak Level 2 over a year earlier.

These assessment results provided an opportunity to compare the various methods of computing the maturity level. Recall that the four proposed methods were:

1. Highest level for which the average score for all key practices that comprise that level is greater than 3.5.
2. Highest level for which the scores for all key practices that comprise that level is greater than 3.5.
3. Highest level for which the scores of all the key practices that comprise that level is at least 3.0. The maturity level is Weak, Solid, or Strong if average of the practices for the level is 3.0 or better, 3.5 or better, or 4.25 or better respectively.
4. There is not a single number for the maturity level. Just present the four averages of the key practice scores for each level.

The maturity levels for the projects in Table 6 using each of the four methods are:

Method 1: Project J-1 is level 3, projects J-2, K-1, K-3, K-4, O-1, O-2, O-4, O-5 are level 2, and the rest are level 1.

Method 2: Project J-1 is level 2 and all other projects are at level 1.

Method 3: Project J-1 is strong level 2, and all other projects are at level 1. Projects J-2, O-1, O-2, O-3, O-4, O-5, O-6 are strong level 1. Projects J-3, K-1, K-2, K-3, K-4, M-1, N-1, and P-2 are a solid level 1. The other projects are weak level 1.
Method 4: Four maturity levels for each project.

Practices	Level 1		Level 2		Level 3		Level 4	
Organizations	Min	Ave	Min	Ave	Min	Ave	Min	Ave
J-1	4.5	4.7	4.1	4.5	2.9	3.9	1.5	3.0
J-2	3.9	4.5	2.3	3.6	1.5	2.4	1.0	1.7
J-3	2.8	3.6	1.0	3.2	1.5	2.4	1.0	1.6
K-1	2.8	3.9	1.2	3.5	1.0	2.0	1.0	1.3
K-2	2.5	3.8	1.2	3.2	1.0	1.5	1.0	1.1
K-3	3.3	4.2	1.0	3.5	1.0	2.0	1.0	1.3
K-4	3.3	4.2	1.7	4.1	1.0	2.0	1.3	1.5
L-1	2.6	2.6	1.4	3.0	1.0	1.8	1.0	1.4
L-2	1.4	1.5	1.1	1.8	1.0	1.4	1.0	1.1
M-1	2.8	3.9	1.3	3.4	1.0	1.7	1.6	1.8
M-2	2.3	3.4	1.1	2.4	1.0	1.7	1.2	1.5
N-1	2.7	3.9	1.0	2.3	1.0	1.8	1.0	1.0
N-2	1.0	1.8	1.0	1.7	1.0	1.8	1.0	1.2
N-3	1.0	1.6	1.0	1.6	1.0	1.7	1.0	1.1
N-4	1.0	1.8	1.0	2.3	1.0	2.0	1.0	1.2
O-1	4.8	4.9	2.0	4.1	1.0	2.7	1.0	1.3
O-2	4.3	4.4	1.8	3.9	1.0	2.3	1.2	1.5
O-3	4.8	4.9	1.0	3.0	1.1	2.6	1.7	2.1
O-4	3.7	4.4	1.5	3.5	1.0	1.9	1.0	1.2
O-5	3.9	4.5	1.8	4.1	1.0	2.3	1.0	1.1
O-6	3.8	4.4	1.0	2.9	1.0	2.3	1.3	1.6
P-1	2.0	2.3	1.0	2.0	1.0	1.8	1.0	1.0
P-2	3.2	4.1	1.0	2.5	1.0	1.7	1.0	1.4
P-3	1.0	2.0	1.0	1.4	1.0	1.5	1.0	1.0
P-4	1.0	2.2	1.0	1.9	1.0	1.8	1.0	1.0

Table 6. Version 2 assessments.

Which of these methods seems to best reflect the maturity level? To begin to answer question consider Projects J-1, J-2 and J-3. It seems clear that J-1 is more mature than J-2 and J-2 is more mature than J-3. Both methods 1 and 3 reflect this while Method 2 only gives J-1 a level 2.

Method 2 seems much too harsh and only capable of making coarse distinctions. For instance according to Method 2 projects N-3, N-4, O-1 and O-2 are all level 1 even though the latter two have much higher values and appear to be level 2. Method 1 suggests that O-3 and O-6 are less mature than the other O projects and that K-1, K-3, K-4 and O-1, O-2, O-4, and O-5 are about the same maturity. Method 3 suggests the opposite. Although Method 1 appears to best reflect the maturity level differences, its level seems inflated.

A single key practice seems to be highly influencing the maturity level calculations above. Except for J-1 and J-2, the Level 2 Min values are 2.0 or less. Hence for Methods 2 and 3 the maturity level for all of the other projects must be 1. A primary reason for the low Min values is key practice 6. With few exceptions, key practice 6 scores are 2.0 or less. Recall that for the original version, there was only had one key practice (6) related to checking that the required documentation was done and it was the most important factor in distinguishing between level 1 and level 2 projects. In the second version, this key practices was expanded to three key practices: (5) existence of a policy or standard, (6) adherence to the policy or standard, and (7) mechanism to check the required documentation was done. To use all three of these key practices and to ameliorate the influence of key practice 6, the average of the three practices was used in Methods 2 and 3. Thus the level 2 Min is the minimum of (key practices 3, 4, 8 and the average of key practices 5-7).

Using the new level 2 Min calculation for Method 3, J-1 is a strong level 2, J-2, K-4, O-1, O-2, O-4 and O-5 are solid level 2, and the rest are level 1. Note that K-3, O-3, O-6, and P-2 are strong level 1.

7. CONCLUSIONS

We have developed a four level software system documentation maturity model and assessment procedure. For each level of the model there are key practices and indicators. An assessment questionnaire was developed that attempts to ascertain which and to what extent these key practices and indicators are part of the documentation process. A documentation process assessment report generated from the questionnaire responses gives a maturity level, documentation process maturity profile, and action plan to improve the process. This paper has reported our experiences assessing the documentation process of projects at several organizations and changes to the assessment procedure based on feedback from members of the projects that were assessed. It also reported the results of 25 assessments using the new version of the assessment procedure.

As expected most projects assessed were at level 1. But what was most surprising was the focus of the project teams on the maturity level in the assessment report and passing attention to the key practices. We believe that since the goal of the documentation process assessment is to improve the process, it follows that the most important part of the report was how well the project team satisfied the key practices and the action plan to improve their process. However, this was not the case. This suggests changing the assessment report to de-emphasize the maturity level and forming a closer relation between the maturity level and key practices. Several proposed solutions were given. One was to compute the maturity level from a combination of the averages and minimum of the key practices for each level. Another is to not give a single number for the maturity level, but instead use the averages of the key practices for each level.

Perhaps the most attractive feature of the assessment procedure is it only takes 30 minutes to complete the questionnaire. The goal of the assessment procedure is to identify key practice areas needing improvement. The next step to improving the process is investigate these key practice areas more thoroughly and formulate a plan to improve them. We realize the trade-offs between our approach and a more comprehensive approach. The comprehensive approach would delve more deeply into documentation practices through a process audit by trained professionals involving interviews with project personnel, an extensive self-study report prepared by the organization, and a more extensive questionnaire. It would produce an extensive report or findings and recommendations. We are trading an inexpensive and minimal inconvenience process that obtains general information for a very expensive and obtrusive process that obtains more detailed information. The general information can used to point to areas where additional resources can effectively be committed for an in-depth investigation and recommendation.

Several project teams remarked that one of the most beneficial outcomes of the assessment process was the meaningful discussion it generated about the types, role, and utility of documentation among software engineers and managers. They also expressed strong support for assessment of documentation and the need to do something to improve the process. Recognition of the importance and motivation to do something about it are crucial first steps in the improvement of the documentation process.

Finally we mention two difficult questions of a quantitative nature that are not addressed in this paper. Both of these questions are of fundamental importance and require consider data collection. One question is whether project teams at a higher documentation maturity

level produce higher quality software. A way of demonstrating this is to show that organizations at a higher maturity level detect more of the defects earlier. Unfortunately few organizations collect defect data and few of these categorize defects by such factors as severity or when introduced. Recall that for the one organization that did collect and categorize defect data, the data indicated that the two projects teams at a higher maturity level did detect more of the requirements and design defects earlier the project team at level 1. All of this underscores the importance of defect data collection not only for process assessments like ours, but for all process changes. A baseline is needed so that the effect of changes can be quantitatively evaluated. We are anxious to assess projects for which there is defect data.

A second difficult question is what is the Return On Investment (ROI) for documentation process improvement? That is, what is the expected ROI in going from level 1 to level 2 or from level 2 to level 3? Major difficulties here are in the data collection and analysis areas. They would involve accurately determining both the cost of achieving the higher level and the cost estimate for a project without the process improvement and collecting data related to effectiveness of projects at the different levels.

8. REFERENCES

1. Card, David N., McGarry, Frank E. and Page, Gerald T., "Evaluating software engineering technologies", *IEEE Transactions on Software Engineering*, Vol. Se-13, No. 7, 1987.

2. Cook, C.R. and Visconti, M., "Documentation is Important", *CrossTalk* Vol. 7, No. 11 (November 1994), pp. 26-27, 30.

3. Humphrey, W.S., "Characterizing the software process: a maturity framework", *IEEE Software* May 1988.

4. Lientz, Bennet P. and Swanson, E. Burton, "Problems in applications software maintenance", *Communications of the ACM* November 1981.

5. Paulk, M.C., Curtis, B. , et. al., "Capability maturity model for software", Carnegie Melon University, Software Engineering Institute, 1991.

6. Rombach, H.D. and Basili, V.R., "Quantitative assessment of maintenance: an industrial case study", *IEEE Proceedings of Conference on Software Maintenance*, City Press, 1987.

7. Visconti, Marcello A. and Cook, Curtis R., "Software system documentation process maturity model", *Proceedings 21st ACM Computer Science Conference*, Indianapolis, IN, February 1993, pp. 352-357.

8. Visconti, Marcello A. and Cook, Curtis R., "A software system documentation process maturity approach to software quality", *Proceedings 11th Pacific Northwest Software Quality Conference*, Portland, OR, October 1993, pp. 257-271.

All Software Is NOT Created Equally: The Selection and Adaptation of a Formal Inspection Methodology for a Corrective Maintenance Environment

Steven A. March
smarch@urbana.mcd.mot.com
(217) 384-8509

Motorola Computer Group
Motorola, Inc.
1101 East University Avenue
Urbana, IL 61801

Keywords: Software inspection methodologies, formal inspection, software maintenance, corrective maintenance, software development management

BIOGRAPHICAL SKETCH

Steven A. March is currently a software engineer with the Motorola Computer Group, Motorola, Inc. He specializes in the areas of software inspection, root cause analysis, object-oriented test design and verification and validation of fault tolerant systems. He has been working in the software industry since 1989, having attended the University of Illinois at Urbana-Champaign and has previously held several positions in the Department of Computer Science at that university where he wrote and maintained many, many lines of uninspected code.

INTRODUCTION

According to the wisdom of W. Edwards Deming, improvements in quality will lead to a reduction in rework and overall productivity gains. We are constantly seeking out new ways to improve quality in search of productivity gains and increased profitability. Usually, we look to adopt new processes or change existing processes to help us improve quality. The goal is obvious but the pathway to successful quality improvements is often obscured with alternatives. Alternatives must be viewed in the context of the development environment and the product being developed for all software is not created equally. There is no "one size fits all" process. Before adopting or changing processes, it is first important to understand the environment our processes live in and how it effects the quality of the product. Second, it is important to define the goals of the project and product in a tangible form. Third, processes must be evaluated within the environment and their contribution toward achieving the goals determined. Fourth, consideration should be given to tuning or tailoring the process to improve and maximize the opportunity for achieving these goals. Lastly, it is also necessary to focus attention on only one or two areas of the overall process at a time. Simultaneous widespread changes in the overall development process will cloud the contribution of an individual change to achieving the goals.

Adoption of a new process is complex. As an example, this article examines the adoption of a formal software inspection process within an industry setting. Formal software inspections have been identified as a cornerstone of a sound overall software development process. Like any mature methodology or process, there are a few books [Ebenau94, Freedman90, Gilb88] and seemingly countless journal articles written on the topic. Within these books and articles, many styles of software inspections are described and promoted. The adopter of inspections has many apparently successful styles (processes) to choose from. How does one decide which style of inspections to implement? Will the selected style be suitable for all software being produced at your business? How does one answer these questions?

The specific case study presented in this article is the selection of a formal software inspection process for a corrective maintenance[1] environment. The presentation traces the process of identifying a process and using an inductive approach to determine the suitability of the process to our product and our environment. This study determines that the methodology of formal inspections is highly applicable to both the product and the environment but the process by which the inspections are carried out requires some tailoring. The tailoring is presented and discussed. The case study concludes by supporting the argument that while the methodology of formal inspections might be applicable there is no single formal inspection process which is universally applicable to all software products developed within all development environments. Furthermore, without recognition of this fact, successive action and close monitoring, the success of formal software inspections in an organization will be far less likely.

FACTORS INFLUENCING SOFTWARE DEVELOPMENT PROCESSES

Not all software is created equally. That is to say, not all software products are created within the same environment using the same process. There are factors present in all software development environments that directly determine which processes are used during the development of the product. This selection of processes, in turn, directly determines the quality of the software produced. When looking for factors which influence the selection of processes look for contributors or detractors to project and product risk. Risk is a multi-faceted beast and this should be reflected in the characterization of your working environment. The nature of the product itself can effect risk. Certainly, the size and complexity of the product as well as the degree of innovation required to solve the problem effect risk. The style and organization of project management as well as the number and experience of software developers contribute to risk. Let us examine some common factors more closely.

(1) The nature of the product

What do we mean by "nature"? The word "nature", as defined in a dictionary refers to the "inherent character or basic constitution of a [...] thing". What is it? Is it an air traffic control system? Safety critical applications tend to demand more attention during development to ensure high quality. Perhaps there are even contractual requirements placed on the

1. A clarification of terminology may be necessary. In this article, corrective maintenance is the process of diagnosis and correction of errors in the software system. Adaptive maintenance is the modification of software to a changing environment or interface. Perfective maintenance is the enhancement of software to address new requirements.

development processes. Is it an X Windows desktop calculator application? Is it a web based mail-order entry system written in Java? If so, perhaps the development process should include rapid prototyping of the user interface to aid in eliciting interface requirements. Is the product a throw-away? If so, perhaps automated tests are not required. Any tests that are written will also be throw-away and could be performed manually once. What are the functional, reliability, dependability and reusability requirements of the product? How well are these requirements specified? Are these requirements complete? How well specified is the design of the product? Is the design complete? Is your product susceptible to frequent change?

In thinking about the nature of a product we can see that it will influence the effectiveness of any development processes applied to it. It is relatively easy to feel confident that one has fully specified the requirements for a desktop calculator application. Why do you think it is used as an example is so many computer science text books? On the other hand, the nature of an air traffic control system or a fault tolerant telecommunications switch demand more of process to achieve their quality goals.

(2) The nature of the product's market

What is the nature of the product's market? What is the nature of the customer? What is the nature of the competition? When is the marketing window? How risky is the product and the market? What are the short-term costs/benefits of selling into a market? What are the long-term costs/benefits? How long before market forces kill a product or product line?

Companies who produce software don't expend effort and money just for the fun of solving neat problems. They produce software to sell (so they have the money to produce and sell more software strangely enough). Software development and software sales must work, hand-in-hand, to determine how the product will be produced. This certainly influences the quality goals and the development processes used to achieve these goals. If you are focused on process improvement without regard to business impact then you are missing half or more of the picture. This may, in fact, be harmful for the business.

(3) The size and complexity of the product

The size and complexity of the product obviously help determine the amount of risk in the product. The larger and more complex the product, the higher the risk. The ability of the development staff to deal with a large and highly complex product is mostly a function of experience. More experienced staff help to mitigate the risk in the product, less experienced staff can increase risk. In addition, larger products attempting to minimize staff size will rely more heavily on risk-based testing.

(4) The scale and organization of the project

Our air traffic control system is certainly larger in scale than our X Windows based desktop calculator application. How does this effect the processes we use? How does the development risk compare between the two? How does the testing risk compare between the two? How do we compare the testing of each. Should there be a separate testing organization? If so, then we have decided to trade testing effectiveness and productivity for blindness and unbiased testing. Of course, this isn't a trade-off if quality testing practices are institutionalized in our organization. Is configuration management and release management a separate function from development?

What are the project resources? How many developers are developing the product? How much hardware is available to them?

(5) The degree of innovation in the product development

Have the problems we are trying to solve ever been solved? How new is the development organization to the state-of-the-art? If we are very new to the state-of-the-art in a particular domain, do we have experience managing state-of-the-art projects in other domains? Certainly, state-of-the-art development requires creativity and innovative solutions. These can be development solutions or management solutions. The interplay of the necessity of innovation and the necessity of process to mitigate risk is a delicate balance. If

the work space is overly constrained by the process, innovation will be frustrated and development and business goals placed at higher risk.

(6) The history (or lack thereof) of the project and product

How long has the product been under development? One year, five years, two decades, two months? Is the product undergoing corrective or perfective maintenance? How completely specified are the requirements? How many times have the requirements changed? How completely specified is the design? How many times has the design changed? How well documented is the code? How many times has there been developer turnover? How many times has the management changed? How many times has the market changed? How many marketing windows have been missed? What phases of development has the project undergone?

In some cases it is difficult to deal separately with these factors due to their interconnected nature. For example, the size, complexity and innovation of a product could be included when thinking about the nature of the product (or vice versa). In addition, the size and organization of project management could be one of the most difficult aspects of developing a project requiring a high degree of creativity and innovation. The experience of project management in dealing with complex and large scale development projects directly effects risk of delivery and quality. Project and product history can run orthogonally to all the factors mentioned above. It is probably a safe bet that there are other factors which have an influence on risk, on quality and ultimately on the processes which should be applied to development of a quality software product. This list only represents a beginning.

Clearly, selection of processes involves risk analysis and complex cost/benefit trade-offs with few guidelines that always apply. Nevertheless, the following guidelines seem to stick more frequently that others:

(1) Characterization of the working environment as it pertains to process selection should be driven by an analysis of the risks involved in a project. These can be risks to delivery, risks to quality, etc.

(2) Generally, the larger the risk, the more demand for discipline in development and management processes is required.

(3) The larger and more complex the product/project the higher the risk to delivery and quality. Hence, there is an increased demand for discipline in processes. Entropy is ever present and particularly visible on large, complex projects. Discipline and process are the best weapons with which to fight entropy and manage the risk.

(4) The larger the necessity of innovation and creativity, the higher the risk. Discipline might be suggested, however, how much can creativity be constrained and still remain innovative? Certainly, higher risk demands discipline, however, depending upon what is being adhered to, this discipline can stifle creativity to the point where real progress is not possible. If no one has ever solved this problem before, how can we be sure that the discipline we used on other projects will be suitable for this problem as well? It may be, but it also may not be.

(5) The history of a project may hinder process adoption or selection in many ways. In addition to the possibilities mentioned in the text above, the history of a project or product includes the effects of personalities that worked on the project, past development budgets, previous visions which turned out to be insightful or just plain wrong, previous development and management processes, methodologies and philosophies, previously applied tools, and others. This is truly one of the large areas where software is not created equally.

The current trend in the software development community is to give high praise to processes which are successful in one or two or three environments, develop a course or two and run around educating "the uneducated" about how to develop software in their environment. Perhaps this is an over exaggeration but the point should be clear. No single process is suitable for all products in all environments. There is no "one size fits all" process. That is not to suggest that journal articles and books should not be written on process success stories. That is not to say that training should not be developed on the application of processes. On the contrary, processes, methodologies, experiences and experiments in software development must be published and courses must be developed to allow development organizations the opportunity to analyze and evaluate the likely success of those processes in their own environments. These provide the best seeds for process improvement. Ideally, success stories and training should include a detailed descriptions of the rel-

evant environmental factors which contributed to the successful application of a process. This information is a jewel to find, unfortunately, it is rare and seldom complete when given.

DEFINITION OF THE PROJECT AND PRODUCT GOALS

Much has already been published on definition of the goals of process improvements by Basili [Basili84] and others. Briefly, Basili promotes a paradigm called "Goals, Questions, Metrics" which is used to specify the project and product goals, specify questions which help to quantify those goals and metrics used to help answer those questions. Without defining these goals, there is no basis for comparison between competing processes or process changes. Goals can be objective or subjective. Obviously, the ideal is an objective goal from which questions and metrics are easily determined. Nevertheless, documentation of goals whether objective or subjective is essential to process evaluation and tailoring. Without identifying and documenting goals before process evaluation or tailoring it will be more difficult to establish whether the expected improvements were seen or not.

PROCESS EVALUATION AND TAILORING

How can one determine the sufficiency of a process to an environment? To answer this question we take an inductive approach. What are the factors in the environment? What are the project and product goals? Once we have answers to these two questions we can then hypothesize that a particular process or processes will be most likely to contribute to achieving those goals. We can then pilot these processes (hopefully they are few in number) and determine a constructive improvement recommendation. This recommendation may include tailoring of the original process.

Published success stories are key to determining which processes to pilot. Again, hopefully these success stories have detailed their environments, project and product goals and evaluation procedures.

After we have selected a process or processes to pilot we must determine how to evaluate them. The evaluation is directed by the list of goals previously identified. This evaluation can take two forms, quantitative and qualitative.

Quantitative evaluation of a prospective process is always preferred to qualitative evaluation. Quantitative approaches are generally easier to apply to projects which are smaller in scale and further away from state-of-the-art. When a project is small and the technology is completely understood, it is far easier to determine what measures are important to collect. Unfortunately, often, the reality of the situation requires that you must depend upon qualitative assessments more than you would like. It is usually possible to combine quantitative and qualitative analyses into a single improvement recommendation.

Quantitative evaluation typically involves tracking of pre-release and post-release defect profiles, defect densities and effort over time. In [Basili87], Basili and Rombach suggest that development environments can be characterized by the number and type of defects also called defect profiles. Project goals can be characterized by defect profiles. Furthermore, methods and tools can be characterized by their impact on defect profiles. For instance, applying a static analysis tool which examines variable usage will reduce the number of defects due to uninitialized variables. There are two factors to consider when looking at defect metrics. Firstly, unless there is an unusually high degree of discipline among developers, little will be known about defects caught in unit testing. One complication is the interweaving of development, debugging and unit testing. Often these are performed simultaneously. It can be argued that this is approach is effective and cost beneficial. However, knowledge of defects found in unit testing helps to determine relative effectiveness of unit testing with other defect detection activities like formal inspection, integration testing, system testing and user-centered testing. Secondly, post release defect metrics are missing one key element -- the other defects that will be discovered before the product has reached the end of it's life. This fact is true anywhere in the life of a software product. Nevertheless, defect profiles and defect densities are probably the best source of quantified data by which we make comparisons.

Qualitative evaluation might include answering the following questions:

(1) Does the process help to manage the risk involved in achieving project and business goals?

(2) Does the process support or hinder the productivity of the development staff? If you have no hard data on productivity, this becomes a qualitative judgement.

(3) Does the process support or hinder the management of the project?

(4) What is the overhead in the process versus the

benefit of the process? Again, without effort metrics, this becomes a qualitative judgement.

(5) Does the process produce software which meets customer expectations?

Quantitative evaluation is stronger than qualitative evaluation and will produce a more effective process evaluation. Nevertheless, this should not dissuade piloting processes when one must rely heavily on qualitative assessments. Remember, quantitative evaluation tends to be more difficult on a project large in scale and which requires more creativity in solution finding. Likewise, one should not be dissuaded from attempting quantitative evaluation on such large and complex projects. In the end, the final recommendation will likely come from both quantitative and qualitative assessments.

THE CASE STUDY

Two years ago an analysis of post-release defect profiles and defect densities was performed to identify defect root causes. One of the conclusions of this analysis was that our development organization should adopt a formal software inspection process in place of our undocumented and inconsistent desk checking policy. We performed an analysis of formal software inspection methodologies commonly used in industry and selected Fagan's approach [Fagan76, Fagan86]. We then set up a pilot project to evaluate the process. We employed a consultant to train the pilot group and offer advice as we moved forward in adopting formal inspections. Unfortunately, about three months into the use of inspections we had quantitative and qualitative indications that formal inspections where not always productive in our environment. Specifically, formal inspections as defined below failed to be productive on small design and code changes which were in the majority in our environment. Quantitatively, we could see that the effort per size was greater for these small work products owing to process overhead, the number of inspectors and the number of inspections per product feature or defect fix. Qualitatively, developers were complaining of the wasted time inspecting small changes. They were seeing in practice what we could see in the data.

We had failed to properly characterize our environment and determine the influence it would have on our adoption of this new process. As a result of this, we were forced to examine the inspection process in *our* context rather than in the context it was presented in industry journal articles and in books. After this examination, we decided that fundamentally, the methodology was highly applicable to our environment but that the process by which it was applied needed to be tailored.

The following sections roughly follow the logical flow of events as they unfolded in our case study. First, there is the definition of formal software inspections that we used to define the process being piloted. Second, a detailed discussion of our project and product goals as they relate to formal software inspections. Third, a detailed discussion of our environment and the suitability of this formal inspection process to it. Last, we detail the tailoring which was performed and give our reasoning for them.

FORMAL SOFTWARE INSPECTIONS

Formal inspections have a defined and documented process, are scheduled into project plans, are measured, evaluated and managed. Formal software inspection processes generally have a singleness of purpose - to find and document defects in the work product. Discussion of alternatives to or style of the work product are not permitted during the meeting. When the meeting wanders into discussions of style and alternative representations or implementations, the moderator refocuses the meeting on defect finding. The purpose of inspections is not to educate, therefore the inspection teams are only as large as required.[1]

During our industry analysis we examined the common published inspection processes. These processes are defined and distiguished by answers to these ten questions:

(1) What is inspected?

(2) When is it inspected?

(3) Who participates in the inspection?

(4) What are the responsibilities of the inspection participants?

(5) Who facilitates the inspection process?

(6) How is the inspection meeting conducted?

(7) What checklists, if any, are used?

(8) How are defects resolved?

---

1. Certainly, education is one benefit of inspections but it should be perceived as a side effect. It is important to always maintain the focus of the inspection meetings and, indeed, the process on detection of defects. This point should guide the number of inspectors in the meeting rather than the potential educational side effect.

(9) What data is collected and tracked?

(10) What are the indicators of a quality inspection?

We started the pilot project with the following process:

(1) What is inspected?

Typically, all work products are inspected. This includes, for instance, requirements specifications, high and low level design specifications, code, test case specifications and test code.

The management in our organization was concerned about the productivity and scheduling impact of introducing inspections and were unwilling to allow inspection of all work products. This concern was shared by developers as well. Therefore, only code was originally inspected with introduction of requirements, design and test case inspections planned for several months later.

On a finer level of granularity, the actual inspection should focus on inspecting all new and changed parts of the work product as well as any dependencies or references to other work products. In the case of code, this includes the definition and use of interfaces.

(2) When is it inspected?

For code, industry data suggests that formal inspection is more effective at finding defects than unit testing. Therefore, our process applied inspections prior to unit testing. Though, as part of the pilot, we hadn't changed our unit testing process to reduce it's effort, we hoped that if formal inspections were successful that we could reduce it in the future.

Non-code work products such as requirement specifications, designs and test case specifications are inspected after all work has been completed and they are considered in final form. Since our pilot project was initially only inspecting code, this was not an issue at first.

(3) Who participates in the inspection?

The inspectors are peers who typically have domain knowledge and include the author of the work product. The number of participants in the inspections is only as large as necessary to avoid the complications of a large meeting and the ballooning of inspection effort.

(4) What are the responsibilities of the inspection participants?

There are four named roles in the inspection: a moderator, a reader, a recorder and an author. Additional inspectors should only be present when they can apply specific domain expertise to uncover defects. A rule of thumb suggests that no more than two additional inspectors be present in any one inspection. Therefore, the process supports between four and six inspectors.

(5) Who facilitates the inspection process?

The inspection process for each project has an owner and caretaker called the inspection coordinator[1]. The coordinator is responsible for the following aspects:

- establishment of the initial project inspection process
- arranging training in inspection process
- owning and managing the administrative procedures
- maintaining the database of inspection data
- evaluating, anaiyzing and reporting on the inspection data
- answering questions about the theory of inspections, the process and the administrative procedures

Typically, there is a single coordinator per project. A rule of thumb for large scale development projects, however, is one 20% time coordinator per 50 developers. Having multiple coordinators serves to share the load of this responsibility. In addition, migrating this responsibility around an organization from time to time distributes process expertise. In tum, this accelerates institutionalization of the process.

(6) How is the inspection conducted?

1. The role of inspection coordinator is not to be confused with that of moderator. The coordinator is the process "owner". The moderator moderates an inspection.

The inspection is composed of several meetings. Firstly, an overview meeting is held to quickly and efficiently bring the inspectors up-to-speed on the material being inspected. Overview meetings are only required when there is explicit benefits to the effectiveness of inspection preparation. Therefore, after an optional overview, each inspector individually examines the work product in preparation for the inspection meeting. The inspection meeting is held to systematically paraphrase the work product in search of defects.

During the meeting, defects and their location within the work product are recorded. In addition, each defect is typed into one of several defect type categories (e.g. Logic, Data, Feature Interaction, Interface, Internal Documentation, Input/Output, Maintainability, Performance, Standards, etc.), classed as missing, wrong or extra and assigned a severity of major or minor. Strict definitions of defect types, defect classes, major and minor defects are included in the process.

When all work product materials have been inspected a disposition of Accept, Conditionally Accept or Reinspect is determined. This disposition controls which process will be used to check the rework and bring resolution to the identified defects.

(7) What checklists, if any, are used?

There is an ever growing checklist for each work product type. These checklists are applied during preparation and referenced during the inspection meeting. Their purpose is to identify common defects which every work product should be free of.

(8) How are defects resolved?

The process for resolving defects depends solely on the disposition determined at the end of the inspection meeting(s). If no defects are present, the disposition will be Accept and the inspection is complete. If the seriousness or amount of defects suggests a complete reinspection then the disposition will be Reinspect, otherwise it will be Conditionally Accepted. Once the defects have been identified it is the responsibility of the author of the work product to fix the defects.

In the case of Conditional Acceptance, the moderator and the author are then responsible for holding a follow-up meeting to cooperatively crosscheck the defect list with the list of work product modifications to ensure that all defects were resolved. The moderator inspects the defect resolutions for correctness and, if needed, consults with members of the original inspection team.

In the case of reinspection, the author informs the moderator when the necessary rework has been completed and a complete reinspection of the revised work product is scheduled. It, in turn, will have a distinct disposition from the original inspection.

Defect resolution marks the end of the inspection thereby defining the exit criteria for the inspection.

(9) What data are collected and tracked?

During the course of an inspection, various metrics are collected to aid in process control. The following metrics are collected:

- Inspection starting date
- Work product type
- Actual size of inspected material, measured in pages of text or non-commentary lines of code, NCLOC (A definition was given for NCLOC and an automated counting tool was also provided.)
- Number of inspectors
- Actual overview meeting effort
- Summation of actual preparation effort
- Actual Inspection meeting effort
- Number of major defects
- Number of minor defects
- Inspection Disposition

These metrics have been justified by a GQM model by Barnard [Barnard94].

(10) What are the indicators of a quality inspection?

From the metrics collected during the inspection process, a somewhat subjective

assessment of the quality of the inspection is possible. The following indicator metrics are calculated:

- preparation rate
- inspection rate
- work product size

By these metrics, one can monitor how the inspection process is being applied. Evidence suggests that if one prepares or inspects too fast, then the inspection will uncover fewer defects. In addition, if one inspects work products larger than the recommended size, inspectors get fatigued and miss defects, skip sections to shorten the inspection or inspect at a faster pace. The quality indicators used were:

Table 1: Standard code inspection process quality indicators

Preparation Rate	150 NCLOC/hr
Inspection Rate	150 NCLOC/hr
Maximum work product size	300 NCLOC

Quality indicators are helpful in determining how the inspection process is being practiced and are better than examinations of defect densities or defect find rates, etc. For example, the inspection of a high quality work product will record few defects despite being well inspected. On the other end of the spectrum, the inspection of a low quality work product could record several defects despite being poorly inspected. Hence, it is better to judge the quality of an inspection from the perspective of these quality indicators rather than how many defects were uncovered for the effort.

CHARACTERIZATION OF THE CASE STUDY ENVIRONMENT

The environment that provides the basis for this example was studied for approximately fifteen months. Following the categories outlined above, this environment is typified by the following factors:

(1) The nature of the product

- The product is a workstation operating system with generic application which was originally developed over a decade ago at another organization. A complete set of tests would be daunting to develop at this stage of the product's life.
- After porting (adaptive maintenance) the product for use on our hardware platform, the product has entered corrective maintenance mode.
- Some additional adaptive and perfective maintenance is occurring on the product to focus on the fault tolerant market.
- User documentation is available but of varying accuracy. No internal design documentation is available from the original developer.

(2) The nature of the product's market

- The product's primary market is broad and generic. A segment of the fault tolerant market is being courted and functionality is being added to address this.

(3) The size and complexity of the product

- The product is approximately four million lines of code. Some subsystems are very stable and are not susceptible to change. Others are very susceptible to change, particularly recently completed perfective changes.
- Corrective maintenance changes to the product tend to be small (averaging less than 20 NCLOC per change) and serve to correct known defects. Occasionally, a corrective change will be larger than this.
- Perfective maintenance changes range in size from 5 NCLOC to 50,000 NCLOC per change.

(4) The scale and organization of the project

- Changes to the product originate both from within our organization and within the original development organization. This complicates change management, configuration management and release management of the product. We only inspect changes that originate with us.

(5) The degree of innovation in the product devel-

opment

- The product is a legacy system and despite the fact that it has been ported to run on state-of-the-art hardware, there are relatively few unknowns or problems requiring an unusually high degree of creativity.

(6) The history of the project and product

- The product is a legacy system originally developed outside of our organization.
- During porting to state-of-the-art hardware, automated tests were developed or ported and are actively used to regression test new releases. These tests are incomplete, however.
- The first two releases of the product used desk checking rather than inspections.

GOALS

As mentioned earlier, after characterizing one's working environment, it is important to identify project and product goals. These goals help to determine the suitability of a process to a project and product in an environment.

The goals which were identified in this instance fall into three categories and are as follows:

(1) Business Goals:

- On-time delivery
- Delivery of product which meets or exceeds quality requirements, when available

(2) Project Goals:

- Facilitate and improve accuracy of project development planning
- Monitor progress toward milestones
- Facilitate continuous improvement

(3) Specific Inspection Process Goals:

- Identify and distinguish inspections
- Facilitate planning of inspections
- Monitor inspections
- Improve the inspection process

PROCESS EVALUATION AND TAILORING

After performing industry analysis, and selecting Fagan's approach to formal inspections, we employed a consultant to train our pilot group. A group performing similar work was used as a control group. For code, this control group continued to follow the desk checking policies which were the standard operating procedure. The desk checking policy followed no documented process and simple metrics were collected in an attempt to quantitatively compare the relative effectiveness of the two processes. This quantitative comparison was weakened by the fact that the desk checking policy had no definition of defect and no procedure for ensuring the accuracy of defect and effort measurements. Despite the fact that our process evaluation suffered from this, we feel that the combination of our quantitative and qualitative results were informative and contributed directly to our understanding of the interplay between our development environment and the inspection process.

We evaluated the contribution of the formal inspection process to the overall business and project goals detailed above. It was determined by rough analysis of post-release defects versus pre-release defects that the overall quality of the product was increased by the use of formal inspections. Nevertheless, there was a discrepancy between the effort per defect for smaller sized versus larger sized inspections. In fact, there was a 4 to 1 ratio of effort per defect between inspections of large and small work products. This observation was coupled with subjective analysis and it was determined that the process overhead and the number of inspectors were the major contributors. Tailoring would be required to address this.

A TAILORED FORMAL INSPECTION METHODOLOGY FOR THE CASE STUDY ENVIRONMENT

Firstly, we distinguish between corrective maintenance changes, perfective maintenance changes and changes to introduce new functionality (e.g. the development of a new feature or subsystem). All new functionality and large scale perfective maintenance follows the standard inspection process as outlined above. It was found that this process is effective in improving their quality and is cost justified. The inspection of corrective changes and small perfective changes follows the tailored inspection process described below. Corrective changes, as mentioned above, average 20 NCLOC per change. Large scale perfective changes are defined as any change greater than between 20 and 100 NCLOC. Small scale perfective changes are then

defined as anything in the range of one to 20 NCLOC. Ranges are given as guidelines. The moderator and the author have the responsibility to decide on a case by case basis if the standard process or the tailored process is to be followed.

Using the dissection of inspection processes presented earlier, what inspection process is suitable for corrective maintenance and small perfective maintenance changes in our environment? The following describes our tailored process.

(1) What is inspected?

For corrective maintenance and small perfective maintenance changes we allow inspection of design, code and test cases to be coupled. The singleness of purpose (to uncover and document work product defects) remains and discussion of alternatives and style remain forbidden. The idea here is that for small changes the overhead of holding separate inspections for design changes, code changes and test case changes is too great. In addition, if a design defect is uncovered during the inspection, the small size of the change is such that the amount of additional rework of code is still smaller than the overhead incurred by multiple separate inspections.

We also limit the scope of the inspection to new and changed code. The complexity of our product is high enough that demanding that all dependencies be inspected would have us inspecting everything. Therefore, we consciously don't try to uncover the problems were you tickle something here and something "waaaaay" over there breaks. Rather, we depend upon dynamic and regression testing to detect those types of problems.

(2) When is it inspected?

An inspection is held when the design, code and test cases have undergone a technical review (a process distinct from inspections) and are all considered in final form.

Furthermore, since we are modifying legacy code whose original design documentation is either incomplete or unavailable most developers use coding, debugging and unit testing firstly as an exercise in requirements elicitation. Therefore, even more so than in a typical environment, there is a tight coupling of development, debugging and unit testing. This tight coupling makes it difficult to perform inspections prior to unit testing as the industry data suggests. In addition, within this environment, defect profiles, severities and effort are not being collected for unit testing. This prevents us from using quantitative analysis to extricate unit testing from this tangle and place it behind inspections within the overall process. Some amount of unit testing is therefore permitted prior to inspection.

(3) Who participates in the inspection?

Two to three inspectors. Evidence has been published to indicate that using smaller inspection teams does not necessarily degrade the effectiveness of the inspection process. In the case of small corrective and perfective changes, we feel that an inspection team of two to three individuals is more easily cost justified. Again, on a case by case basis, the moderator has the authority and the responsibility to determine if such an inspection team is sufficient. If such a small team size does not address the domain of the change, then a larger team is selected and the standard inspection guidelines for limiting the size of this team come into play.

In addition, to the cost justification advantages of a small inspection team, we find a greater degree of manageability of inspections. With larger teams, scheduling inspection meetings often becomes difficult thereby delaying the integration of a change.

(4) What are the responsibilities of the inspection participants?

Within tailored inspections, there is always an individual who takes on the roles of moderator, reader and recorder. We refer to this individual as, simply, the moderator. In addition, the author is always present to take part in the inspection.

(5) Who facilitates the inspection process?

The project's inspection coordinator (defined above) is responsible for tending both the standard inspections and the tailored inspections. They are responsible for keeping the data analysis distinct when necessary and possible.

(6) How is the inspection meeting conducted?

The inspection meeting itself is optional upon the discretion of the moderator. Published evidence in [Votta93] suggests that a meeting is not always cost-justified. Correspondence of some sort, be it marked up hardcopy, notes or electronic mail is required to collect the defects from the inspectors.

All defects collected are recorded by the moderator taking up the role of recorder. They are responsible for recording the defect, defect location, defect type category, class and severity of the defect.

Finally, a disposition is arrived at by consensus of all inspectors with weighting of the moderator's viewpoint.

(7) What checklists, if any, are used?

There was no detectable consistent use of checklists throughout the pilot project. Hence, there was no tailoring in this area.

An attempt is being made to develop specific product subsystem related checklists which are deemed more useful than generic language oriented checklists.

(8) How are defects resolved?

The resolution of defects for tailored inspections mirrors that of standard inspections. Defects are fixed by the author and final resolution is reached by a process which depends upon the disposition of the inspection. If the disposition was Conditionally Accept, then resolution occurs at a follow-up meeting between the author and the moderator. If the disposition was Reinspect, the author notifies the moderator when all rework is complete and the moderator schedules a completely new inspection.

(9) What data is collected and tracked?

There is no change in the metrics collected for the tailored inspection process. To reiterate, the metrics collected are:

- Inspection starting date
- Work product type
- Actual size of inspected material
- Number of inspectors
- Actual overview meeting effort
- Summation of actual preparation effort
- Actual Inspection meeting effort
- Number of major defects
- Number of minor defects
- Inspection Disposition

(10) What are the indicators of a quality inspection?

From the metrics collected during the inspection, the following indicators or inspection quality are used:

- preparation rate
- inspection rate
- work product size

The numbers determined in our environment for corrective maintenance changes and small scale perfective maintenance changes are as follows:

Table 2: Empirically derived quality indicators for the case study product and environment

Preparation Rate	70 NCLOC/hr
Inspection Rate	70 NCLOC/hr
Maximum work product size	35 NCLOC

CONCLUSIONS

Over 300 inspections have been performed since the inception of the pilot project, approximately two thirds of which followed the tailored inspection process. In comparison to our established desk checking practice, formal inspections had a higher defect per NCLOC than desk checking with a lower effort per defect. Therefore, the applicability of formal inspections to our process and environment had been proved. Nevertheless, the process was far from optimal for smaller changes.

We originally started with a standard formal inspection process for all types of changes in our environment. This includes perfective changes which introduce new functionality, corrective changes and adaptive changes. After some application of the process, it was noted that the effort per defect for corrective changes and small

perfective changes exceeded that for larger perfective changes by a factor of 4 to 1. We determined the cause of this and further characterized our environment in preparation for tailoring the process. We then defined and adopted a tailored formal inspection process for corrective and small perfective maintenance changes.

Since adopting inspections and tailoring them, we have had two major and two minor product releases. In comparing the previous major release in which desk checking was used, we have decreased the post release defect density by 21%.

Our experience supports the argument that while the methodology might be applicable, there is no single formal inspection process which is applicable to all products being developed within all environments. There is no "one size fits all" process. Recognition of this fact was key to the overall success of introducing the inspection process into the established development process.

REFERENCES

[Bamard94] Barnard, Jack and Art Price, AT&T Bell Laboratories, "Managing Code Inspection Information," IEEE Software, March 1994.

[Basili84] Basili, Victor R. and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, vol. SE-10, no. 6 (Nov 1984), pp 728-738.

[Basili87] Basili, Victor R. and H. Dieter Rombach, "Tailoring The Software Process To Project Goals and Environments," Proceedings of the 9th International Conference on Software Engineering, Monterey, CA, 1987.

[Ebenau94] Ebenau, Robert G. and Susan H. Strauss, "Software Inspection Process," McGraw Hill, 1994.

[Fagan76] Fagan, Michael E., "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, vol. 15, no. 3, 1976, pp 182-211.

[Fagan86] Fagan, Michael E., "Advances In Software Inspections," IEEE Transactions on Software Engineering, vol. 12, no. 7, July 1986, pp 744-751.

[Freedman90] Freedman, Daniel P. and Gerald M. Weinberg, "Handbook of Walkthroughs, Inspections and Technical Reviews, 3rd edition," Dorset House Publishing,1990.

[Gilb88] Gilb, Tom, "Principles of Software Engineering Management," Addison-Wesley,1988.

[Glass59] Glass, Robert L., "Editor's Comer: In Search of an Obvious Yet Radical Idea," Journal of Systems Software; 31:1-2, 1995, Elsevier Science Inc.

[Votta93] Votta, Lawrence G., "Does Every Inspection Need A Meeting?," Proceedings of the1st ACM SIGSOFT Symposium on Software Development Engineering, ACM Press, New York, N.Y., 1993, pp 107-114.

OBJECT ORIENTED TESTING: A HIERARCHICAL APPROACH

Submitted by Shel Siegel to the PNWSQC - 1996

This paper is a short summary of the ideas I present during the 45 minute track session at this conference. These ideas are fully developed (511 pages worth) in my book:

Object Oriented Software Testing: A Hierarchical Approach

John Wiley & Sons, 1996 - ISBN 0-471-13749-9

I have also started a web site dedicated to this topic: **ootest@dgap.com**

Please feel free to visit and contribute.

The Hierarchical Approach

The *hierarchical approach* is at the heart of the object-oriented testing system. This test approach uses and builds upon several well-understood testing techniques, tying them together into a comprehensive testing system. The hierarchical approach leverages the fact that "everything is a system." It defines and applies testing standards for several levels of software component: objects, classes, foundation components, and systems. The hierarchical approach designates as *SAFE* those components that meet the testing standards for that kind of component. Once you designate a component as SAFE, you can integrate it with other SAFE components to produce the next-level component. In turn, you test this component to the level of safety associated with the component level it represents. SAFE is always a relative state. It depends entirely on the standards you choose to enforce, your application, your attitude toward risk, and the specific risks and risk management practices you adopt in your project. The hierarchical approach provides guidelines for minimum safety; You decide what is right for you.

The hierarchical approach focuses on foundation components. A *foundation component* may be one complete class hierarchy or some other cluster of classes that performs a core function or that represents a logical or physical architectural component.

After you test a foundation component to a safe level, you can integrate it with other foundation components. Integration testing of safe foundation components then needs to address

only the interconnections between the foundation components and any new composite functionality. The hierarchical approach eliminates the need to test all of the combinations of states during integration testing, thus improving productivity.

This approach views integration testing as a daily activity for the development team. The integrated and SAFE foundation components combine with each other, eventually combining into systems. Foundation components and systems appear as baseline components in the WBS and as milestones on the project schedule. The project milestones are the actual expected deliveries of architectural baselines of the final software product. The hierarchical approach is completely consistent with and supports the iterative incremental and recursive development process. You can adopt, adapt, and apply any piece of the hierarchical approach to work with any other software development process.

Earned value and its associated measures and metrics are a consistent way to track project progress as well as product quality. As you designate foundation components SAFE, they earn value points. You award points based on the relative value of the foundation component to other system components within the context of the overall project. Foundation components eventually integrate into systems. Each system earns its own value upon SAFE designation. These systems correspond to the functional, logical, or architectural layers that are major project milestones and baselines. The development effort progresses up the pyramid in a parallel, iterative, and recursive manner. The goal of the hierarchical approach is to optimize development team productivity and the final value the product delivers to the customers.

The Pyramid Figure (attached at end of this paper) is a graphical representation of the hierarchical approach. Time moves forward by ascending the pyramid. At the base of the pyramid, you thoroughly test individual methods to a SAFE designation. The methods integrate into classes and you test whole class hierarchies to a SAFE level. The collections of SAFE classes (foundation components) form the foundation of your final product. The large integration area of the pyramid is where you integrate the foundation components into the different architectural, functional, and logical structures. They form the systems. At the apex of the pyramid is a short and concentrated system test.

The shortened system test phase represents the harvest of the testing labors sown lower in the pyramid. System test concentrates on the "Illities" (ill at ease) system tests (usability, reliability, flexibility, performance ability, securability, and so on). You spend minimal time regression-testing a system you successfully develop with the hierarchical approach. Most if not all of the regression and system tests suites are already complete from test cases, scripts, and use cases developed and executed lower in the pyramid, so you reuse them in various combinations.

The pyramid figure has been designed with a rhomboidal foundation. This pyramid actually existed in ancient Egypt and was one of the most stable forms of construction. When the basic building blocks of your system are sound and your mortar (integration material) is applied correctly, your system can survive, even thrive for a long time (perhaps not millennia, though) with minimum maintenance and maximum functionality. Implementing the optimization approach, the basis for the hierarchical test approach results in longevity.

Leadership

Vision: A comprehensive method of testing object-oriented software components that you can reuse in many projects to achieve an acceptable level of risk with a high degree of productivity.

Mission: To provide a testing approach optimized for object-oriented software components that you can reuse in many projects to get an acceptable level of risk and a high level of productivity.

Objective 1: To provide a hierarchical test method that you can reuse to achieve an acceptable level of risk in your object-oriented software projects.

Approach 1: Develop a written, standard test method using hierarchical incremental, conditional, integration, and system test models that gives the specific objectives, approaches, and measurements for those objects that will achieve your acceptable level of risk.

The details of this hierarchical test method are presented in my book.

Measurement 1.1: Number of projects that reuse the test approach.

Measurement 1.2: Level of risk for project using test approach.

Objective 2: To provide a hierarchical test method that you can reuse to achieve a high level of productivity in testing your object-oriented software projects.

Approach 2: Develop a test method as in 1 with the additional objective of efficient testing: taking optimal amounts of effort to create and run the tests against the components.

Measurement 2: Schedule performance index for testing tasks in the project (earned value divided by baseline cost of work scheduled).

Structure

Method

The hierarchical approach provides a structured set of test suites and test models. When you scale and apply these suites and models to your project, it will reduce your risk to acceptable levels and will increase your testing and development productivity.

The following test suites make up the structure of the pyramid:

- **Conditional Test Suite:** Tests classes using the conditional test model and its accompanying assertions, exceptions, concurrent test operations, and message polling test scripts.
- **Hierarchical Incremental Test Suite:** Tests foundation components using various test models and scripts (see my book for details), possibly with stubbed references to other components.
- **Integration Test Suite:** Tests combinations of foundation components using the hierarchical incremental models with scripts that use all the components, not just stubs.
- **System Test Suite:** Tests systems of foundation components using system test models.
- **Regression Test Suite:** Tests foundation components and systems.

The hierarchical incremental test suite is the most complex of the test suites. You can build the suite using any or all of the following test models:

State-Transition Test Model: Models the dynamic behavior of classes as states and transitions, with the test object being the life-cycle flows through the model.

Transaction-Flow Test Model: Models the dynamic behavior of classes as black-box transactions, with the test object being transaction flows.

Exception Test Model: Models the exception behavior of classes.

Control-Flow Test Model: Models the dynamic behavior of methods as a flow of control through a method; the test object is a single flow through the model.

Data-Flow Test Model: Models the dynamic behavior of methods as a flow of both control and data through a method; the test object again is a single flow through the model.

See individual chapters in the book for a comprehensive, detailed method for choosing which models and objects to use, building the test scripts into test suites, and automating the execution and result-gathering procedures. Automation issues are discussed in the book. Each object has its own optimization approach details; the operational objectives for each object constitute the test standards for that object.

Resources

The proverbial good news and bad news applies here. The bad news first. The hierarchical approach represents a complete evolution beyond the best current software engineering practices. The Software Engineering Institute's Capability Maturity Model (SEI CMM) would refer to successful development with the hierarchical approach as level 5, if it addressed testing or quality adequately, which to date it does not. In quality terms, the hierarchical approach implements quality optimization principles for software development. To implement the hierarchical approach completely, you need an organization that has mastered all the capabilities presented in the book. Mastery of many of these skills requires in-depth technical training and detailed technical capabilities beyond the scope of my book.

There is also plenty of good news. The hierarchical approach is not an all-or-nothing approach. You can use it in an incremental and piecemeal manner. You can benefit from any and all of the approaches and build upon them as you move toward a vision of software engineering that encompasses many if not all of them. You can implement the full hierarchical approach in phases, in parallel, and incrementally. If you apply the principles from the quality

system (also described in detail in my book) and practice and implement optimization cycles, you will be able to evolve quickly into the hierarchical approach. A list of what I believe to be the minimum level of training and technical capabilities for a project team to start working with some of the objects in the hierarchical test system follows:

- Object-oriented analysis and design (minimum of 3 to 5 days of seminars or the equivalent in experience)
- Mastery of object-oriented programming language
- Object-oriented Testing: A Hierarchical Approach - minimum of 3-4 days of training
- Basic test techniques - (Boris Beizer is still my recommendation for this although there are many competant seminars and a few good books on the subject)

Infrastructure

Adoption of any method benefits from tools that support and automate aspects of it. At press time several tool vendors have expressed interest in developing tools to implement aspects of the hierarchical approach. Some of the pieces have already been implemented by project teams in various companies. I invite the readers to use the web site address listed on the first page of this paper to get the latest information on using and automating the hierarchical approach.

Teaching Software Quality and Leadership: Experiences and Successes

Judy Bamberger
Process Solutions
Adjunct Professor,
Oregon Graduate Institute

bamberg@eaglet.rain.com
+1-503-690-1206
+1-503-690-1548 (fax)

12440 NW Haskell Court
Portland OR 97229 USA

James Hook
Oregon Graduate Institute of Science & Technology

hook@cse.ogi.edu
+1-503-690-1169
+1-503-690-1548 (fax)

Oregon Graduate Institute
PO Box 91000
Portland OR 97291-1000 USA

http://www.cse.ogi.edu/~hook/

Keywords

Education, Software Quality, Software Process, Software Process Improvement, Process Definition, Formal Inspection, Leadership

Abstract

Is it possible to teach software quality and leadership concepts and skills at the graduate level? Can this be done simultaneously within the nurturing environment of the classroom and the risky world of industry? Is it possible to provide students with enough skills and techniques to demonstrate immediate results and, at the same time, provide them with enough background and concepts that they become intelligent consumers and decision makers when it comes to software quality issues?

The Software Process Practicum: Lessons in Software Quality and Leadership, taught at the Oregon Graduate Institute during fall 1994 and 1995, provides a clear demonstration that the answer is an emphatic, "YES!"

This paper covers the following areas:

- The background of the Software Process Practicum and the premises on which it is designed
- The overall Practicum framework and modules taught
- An overview of the Software Skills and Competency Model on which the Practicum is based
- The impact the course has had on some of our students and the results that they, and the organizations for which they work, are seeing
- Future directions

1. Background of the Software Process Practicum

In 1993, the Oregon Economic Development Department funded an effort to:

> ... create a Software Engineering Model that characterizes performance excellence (current state and future state), create the capability to assess software engineers against that model, and provide a means to assess individual and organizational progress toward excellence on an ongoing basis.[1]

The organizations participating in this effort were the four leading software/technology companies in the state of Oregon: Intel, Mentor Graphics, Sequent Computer Systems, and Tektronix.

In January 1994, this group made its first public presentation. At that meeting, Judy Bamberger (a report author from Sequent) and James Hook (an educator from OGI) met each other, began discussing the model, and soon found out they had many interests in common, including taking action to address the ideas in the model.

In April 1994, we[2] committed to teach a new course, which was to become the Software Process Practicum: Lessons in Software Quality and Leadership. Moreover, we agreed to leverage what we could from the *Report of Findings* to guide the structure and content of the course.

2. Basic Premises behind the Software Process Practicum

We agreed upon a set of guidelines for our selection of topics, readings, in-class exercises, and homework assignments. These were:

- Use the "adult learning paradigm" to make the learning experience and the materials relevant.

  Recognizing that adults bring a multitude of ideas, job worries, motivations, and rich experiences with them when they come to class, we decided to leverage that background both in the classroom and in homework assignments. We also used a variety of teaching methods (lecture, discussion, hands-on labs and problem solving activities, student presentations; visual, aural, tactile; etc) to maintain interest.

- Ensure the topics are applied and reinforce each other.

  We used in-class labs to practice new skills. Many homework assignments required the students to take those skills and apply them in a real-world setting with a partner organization. We took time in class to discuss and apply readings and ideas. In general, our approach was: teach, do, and discuss.

- Teach the "soft" stuff, the social processes, as well as the "hard" stuff, the quality processes; and emphasize the inherent interdependency of the two.

  We leveraged the fact that there already are plenty of courses and materials that concentrate on basic quality tools, techniques, and frameworks. The value that we added was to integrate and apply many of these existing concepts and techniques to software engineering.

---

[1] From the *Report of Findings: Joint Software Engineering Needs Analysis*, 18 February 1994, available from Oregon Economic Development Department, Salem OR.

[2] From this point on, "we," "us," and "our" refer to the authors.

- Balance education (concepts, long-term focus) and training (skills, short-term focus) needs.

  This was, for each of us, our most interesting challenge, and a key risk. The challenge stemmed from the fact that each of us represented a community with a different primary driver for consumption of education: academia having a more research, theoretical, long-term tradition; industry have a more practical, skill-oriented, short-term tradition. One component of the resulting risk was how we were going to bridge that gap ourselves, as we collaborated in developing and delivering the course. Another component was how to build that bridge with our students, who we anticipated would be representing both academic and industrial communities. And a third component of the risk involved the selection of which topics to teach, and how to present them.

  We put this to the test as we built the framework of the Practicum and populated it with specific readings, lectures, discussions, and homework assignments. We decided to teach multiple techniques, methods, and frameworks wherever possible, and to provide students with the ability to make decisions about what is appropriate under which circumstances. When it came time to practice (i.e., homework), the students were free to choose the specific techniques or concepts to apply to address the scenario we provided for them.

- Build on a variety of materials.

  We distributed many of the seminal papers (such as Fagan on inspections, Radice on software process definition), outrageous editorials (such as the Bach/Curtis debate on the Capability Maturity Model), classic writings (such as Crosby, Deming, and Juran on quality), books taking a "system" view of software quality issues (such as Weinberg), current findings and issues, cartoons. We required the students to be exposed to some real-world practices in the industry (such as interviewing practitioners and working with a "partner organization" on a multi-part project) and to leverage experiences from their own work environment.

- Use information from the *Report of Findings* to select or validate what and how we are teaching.

  This is discussed briefly later in this paper, and is presented in detail in an OGI technical report.[3]

- Ensure the students have fun while learning, and that we, too, both have fun and learn.

  We recognized that *software quality* may not be viewed as a "glamorous" topic by most people, even though it is increasingly recognized as being important. We also knew that to create an effective learning environment for students and instructors who had already put in an eight-hour day would be a challenge. We faced that challenge and, within the scope of our basic premises, consciously added some irreverent, off-the-wall items.

3. Framework of the Software Process Practicum

The Practicum resulting from this partnership shows its heritage: it is a synergistic mixture of tools, techniques, and practices important to practitioners in industry (as represented by Judy) and concepts, models, and data important to more theoretically-oriented academics (as represented by Jim). It is structured to meet both short-term training and long-term education needs. The resulting Software Process Practicum is a unique and highly effective course covering

[3]Technical Report number CS/E 95-006, *Software Process Practicum: Lessons in Software Quality and Leadership*; March 1995, Oregon Graduate Institute of Science & Technology, Portland OR USA.

both the "hard" issues of quality processes, the "soft" issues of social processes, and techniques that require effective knowledge of both. Figure 1 illustrates this framework.

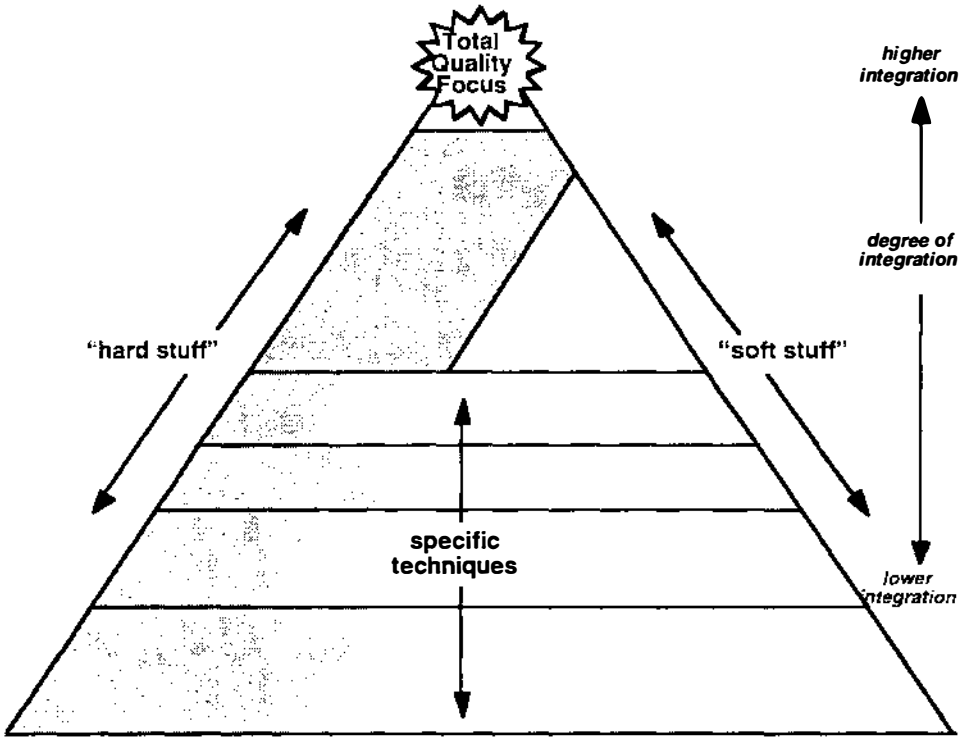


Figure 1. Framework for Software Process Practicum.

- The "hard stuff" represents *quality processes*: basic models, tools, and frameworks related to effective software process. Most of this is provided initially via readings and class lectures.
- The "soft stuff" represents *social processes*: the individual and team social concepts, models, and skills required to apply the hard stuff successfully. Most of this is provided initially via readings and class lectures.
- Specific techniques represent the *integration of the quality processes* and frameworks *with the social processes* and team skills in an applied setting.

  Most of the specific techniques are introduced via readings and class lectures and reinforced and practiced via hands-on labs during the class, a Saturday workshop, and individual and team homework assignments. Moreover, team homework assignments focus heavily on the integration of the quality and social processes in an applied, real-world environment. Two homework assignments require the students to work in teams with a partner organization for an extended period.

Figure 1 also demonstrates our assertion that the overall goal of the course was to provide the students with a total quality focus, the integration of all the parts into an effective and usable whole. This is not to be mistaken to be TQM - Total Quality Management. The Practicum does not advocate any single way of going about achieving "quality"; rather, it focuses on providing the concepts, knowledge, and skills from which the students may select - as critical consumers - to apply as they determine appropriate to each context.

The course objectives we defined stated that, after this class, the students will understand and have demonstrated that:

- Software processes can be managed and controlled.
- Software engineering is a social process, too.
- The skills learned in the Practicum can be applied today at work and can be used effectively.
- The concepts, knowledge, and skills provided in the Practicum can be used to make well-informed decisions about applying software quality principles and tools to personal, project, and corporate software activities.

Given these objectives, we looked at the possible topics to teach. We recognized that there was a "hierarchy" of models, concepts, and skills: beginning with the simple and self-contained, and continuing to the more complex and more highly integrated.

When we started to populate this framework with actual topics, we found we had identified many more topics than there was time to teach in 30 contact hours. To determine which topics we would teach in the first offering of the Practicum (fall 1994 term), we used two primary sources for guidance: the competency model defined in the *Report of Findings*, and our knowledge of the local software engineering industry from which our students would come. The resulting framework populated with the topics we taught in fall 1994 is shown in Figure 2.

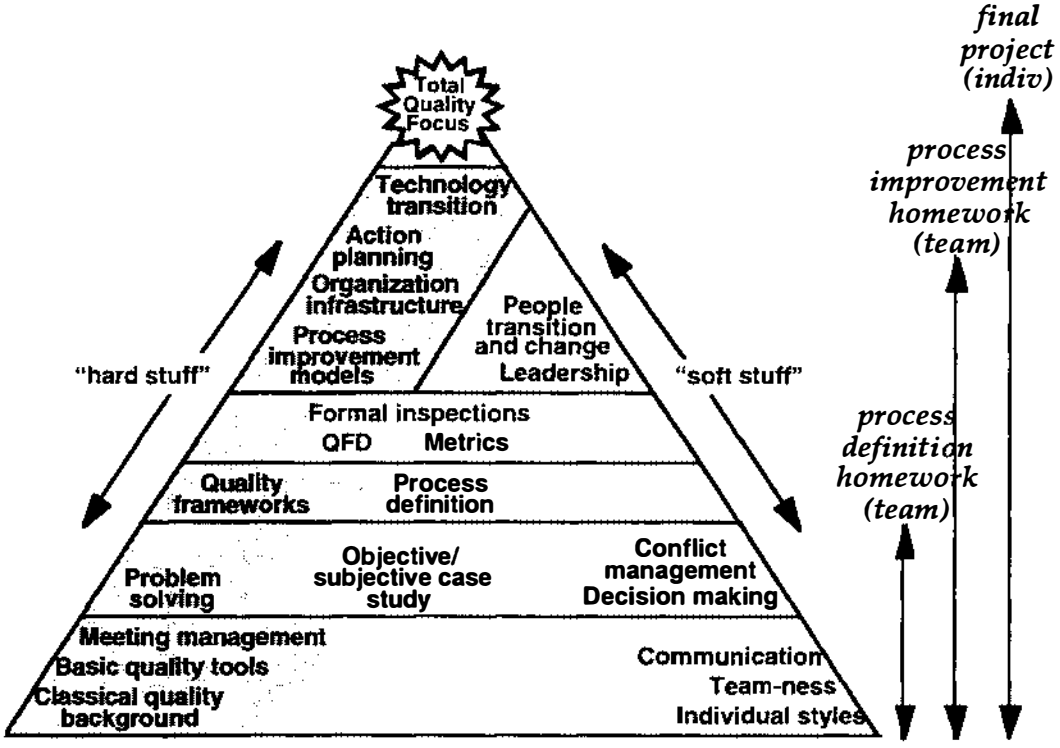


Figure 2. Topics Taught in Fall 1994 Software Process Practicum.

These same topics are being taught today, as the students continue to tell us they are relevant and useful to them.

4. Overview of the Software Skills and Competency Model

The *Report of Findings* contains a matrix of technical skills and competencies. These skills and competencies were identified via a thorough and structured information gathering process.[4]

The four technical skill areas (requirements) that were identified are:

- Process and Management: software process, software review techniques, project management, business literacy, configuration management
- Design: object-oriented design/programming, data structure design, design methods/simplicity, formal analysis techniques, client-server architecture, human interfaces/graphical interfaces
- Implementation: software optimization, debugging techniques, understanding foreign code, technical writing, software testing techniques
- Specific Technical Skills: windows application programming, CAE related technical areas, communications and networking, natural data types, device drivers, real-time systems

The identified competencies were organized into five groups:

- Concern for the Process (information gathering, efficiency, systematic thinking, discipline/rigor)
- Concern for the Team (collaboration, team building, technical leadership)
- Concern for Ideas (communication skills, influence/persuasion)
- Concern for the Company (risk management, results orientation, user orientation)
- Concern for the Solution (persistence, creativity, learning by doing, pattern matching, initiative)

The power of the software engineering competency model is demonstrated via the intersection of the technical skills and competencies. The Software Process Practicum satisfies the technical skills area of Process and Management very highly and three of the competency areas – Concern for the Process, Team, and Solution – very highly. These results are summarized in Figure 3.

The process and management technical skill area was the only one relevant to our course.[5] While all the competency areas were somewhat relevant, we needed to restrict our scope. As a result, we chose to focus very heavily on the Concerns for Process, Team, and Solution.

---

[4]This structured information gathering process is based on a job competency analysis developed by David McClelland (*The Achieving Society*; 1961; Princeton, NJ; Van Nostrand), a critical incident interviewing technique developed by J C Flannigan (*Critical Incident Interviews*, Psychological Bulletin, 51; pages 327 - 358; 1954), and an extension of this technique into critical behavior interviewing (Cambria Consulting, Inc; Boston MA; 1993). It is described in detail in the *Report of Findings*.

[5]OGI has issued a document mapping its curriculum to the Competency Model. This document is available from Oregon Graduate Institute, Computer Science and Engineering Department, Portland OR USA.

	Technical Skills			
	Process and Management	*Design*	*Implementation*	*Specific Technical Skills*
Software Process Practicum		None	None	None

	Competency Areas				
	Concern for the Process	*Concern for the Team*	*Concern for Ideas*	*Concern for the Company*	*Concern for the Solution*
Software Process Practicum					

Figure 3. Mapping of Software Process Practicum to the Skills/Competency Matrix

Each competency area summarizes several specific competencies, each of which in turn summarizes multiple specific and observable behaviors identified via the in-depth interviewing and questionnaire steps described in the report.[6]

5. Impact: Student and Instructor Results

The students who have taken the Software Process Practicum have used many of the concepts and techniques we taught in the classroom in their work environments. They are continuing to demonstrate objective evidence of improvement on personal and organizational processes, which is visible to many of their peers, managers, and Vice Presidents. They tell us the products are being built and released on-time and with high quality.

5.1. Student-Based Results

Nearly all of our students cited meeting management as the most important single topic we taught, and one that was immediately applicable in their work environments. One student was holding weekly meetings with his project team to share the Practicum information through the entire term.

Many students also cited the process definition representations and techniques as something they are using already at work. One student shared with us an observation from the process definition and improvement homework assignments done with her partner organization. The organization's staff stated that they finally saw the relevance of and tangible results from many quality techniques in which they had been trained, but had not used to that point. Another student used the information we provided to help revise many of his company's procedures in preparation for ISO 9000 registration.

---

[6]OGI Technical Report number CS/E 95-006 provides a detailed mapping of each course module to each of the 17 competencies. The mapping indicates whether this was accomplished via readings, class lecture, class activities or labs, or individual or team homework assignments. Also, for each of the 17 competencies, Technical Report describes: what the class sees (specific topics covered via reading, lecture), how the students use it (reinforcement techniques used to ensure key points are mastered, skills built, and general awareness achieved), and how the instructors do it (application of these competencies to us as we built and taught the Practicum).

Two students shared with us that their industry managers told them that the topics and methods being taught in our course are of significant importance to them. Two other students have used their newly discovered organizational skills and have been promoted to team leaders.

Not all results have been limited to the work environment. One student described how he gently applied some of the basic meeting management and interpersonal-oriented concepts we taught to a jury on which he was serving. Other students mentioned using some of the social style profiling models with their families and friends. In all cases, the students reported improvements in understanding of and insight into others.

Many students have been reaping the benefits of the concepts behind formal inspections by increasing their personal discipline and how they produce their software work products. One student used the information from the Practicum to update and refine his team's on-going inspection process, and to provide additional training in the improved process.

Two students are trying to introduce formal inspections into their workplaces. For example,

> A project at ADC Kentrox scheduled a three week period specifically for code inspections, and they are already benefiting from the results of preparing for and conducting three code review sessions. While not as formally conducted as were the sessions taught in class which used uniform codes for defect categorizations and which also focused on accumulating defect metrics, the time spent has widened the scope of understanding of the software by each team member and has resulted in positive, identifiable improvements to the software.[7]

Another student, in her final project, prepared a three-year plan for the introduction of formal inspections across her organization.

We received many positive comments about the exposure to the wide variety of quality frameworks we used, for example:

> ... exposure to the Trillium Model (which integrates and adds to aspects of SEI's CMM [Software Engineering Institute's Capability Maturity Model], ISO 9000, Malcolm Baldrige, and IEEE concepts, and targets these concepts to the Telecommunications Software industry), has led to intense interest in making use of the model for identifying specific areas for potential improvement at ADC Kentrox.[9]

One of the student process definition and improvement projects has been presented to two Vice Presidents. The instrumentation (metrics) requirements identified by this team of students will be included in the partner organization's metrics. These metrics will help the partner organization measure how well it is meeting its specific goals in the area of time commitments to customers. The proposed metrics are defined precisely and tied to specific process steps. Other process improvements were also proposed and will be adopted. These include better control of certain software products, and taking five, piece-meal, information-gathering process steps and merging them into a single step done once at the beginning.

Another student (full-time graduate student with no prior industry experience) who recently took a job with Motorola, which has a strong reputation for its emphasis on process and

---

[7]Personal correspondence from Gary Hanson, February 1995.

[8]This plan was not fully executed. Unforeseen, and real-world, events such as reorganizations and job changes have kept this from happening. However, the student tells us she is using the discipline she learned in many other areas, and this has been having a positive effect overall.

[9]Personal correspondence from Gary Hanson, February 1995.

quality, let us know that the Practicum had set him up nicely. When he was going through the "quality school" at Motorola, he was way ahead of the pack, as he had already seen and used many of the techniques and skills introduced in the class.

One student was assigned leadership of a team trying to get a new product developed and delivered on an extremely tight schedule. He describes how he is leveraging many of the interpersonal skills and quality tools he learned in the Practicum, and is pleased to say that the project is still on-time and well within the desired quality.

One of the intangible benefits to the students - or a benefit that we see - is how they grow and develop as people throughout the term, and beyond. Two students in particular, who could have easily been "hidden" in project activities, have become peer leaders - a characteristic that has been noticed favorably by one of the team's customers.

5.2. *Instructor-Based Results*

We asked for, and received, feedback throughout the course. In fact, on the first day of the course, we asked the students, "What are your expectations from this course? What do you expect to get out of this?" As most of our students had never been in a classroom where someone asked them, the customers, about their needs and wants, they were somewhat surprised. Also, from the outset, we indicated that we were going to ask for feedback at specific points, and that we welcomed feedback and suggestions for improvement at any point. We asked, and the students responded.

From the 1994 term, the most significant piece of information we received was "too much, too fast!" When we heard this, we dropped many of the individual homework assignments, and pruned the readings by setting some aside as optional. To address this issue in the longer term, we increased the credit hours earned for this course from three to four, and we hold the class for two two-hour sessions each week. We did not add more material; we slowed down and spent more time making explicit ties with the readings and individual homework assignments. This, in fact, addressed another issue that was raised.

The independently administered end-of-term course evaluations yielded no surprises. We received very strong ratings in the following areas overall in both 1994 and 1995:

- Instructors knowledgeable
- Lectures well prepared, organized; presentations made for easy note-taking; topics appropriate
- Good communication skills
- Concerned about students; opportunity to ask questions; students treated fairly
- Course material challenging
- Increased my interest
- Built/augmented student competency

In 1994, we received lower ratings in the following areas:

- Related course concepts in systematic manner
- Textbook and supplementary materials valuable
- Pace of the course (it was too fast)

To address these, we included a version of Figure 2 in each lecture, and we systematically used it as a roadmap to explain the relationship of the topics as they were presented. We increased credit hours and other activities described above, and we took more time to integrate the results

of the readings into class time. As a result, the ratings for the first two of these areas showed marked improvement. We still are grappling with the issue of pacing.

Other comments we received included:

- "Keep it as a pair of instructors." coupled with "Jim provided a good 'sounding board' and 'plant' for Judy's lectures. Jim's PacSoft work was good background."
- "As [a] professor, [Jim] gives a strong testimonial for the material's importance."
- "More on metrics." (We are considering adding one more session on metrics and removing something that the students identify as being of lesser value to them.)
- "Course content very important for OGI and community. Get more industry involvement."
- "[Aspects you would like to stay the same] Team projects. Software processes. Subject matter. Instructors excellent - retain and reward. Saturday labs very good."
- "I liked the 'practice what you preach' mentality ... "
- "The team exercises were exceedingly valuable. It gave us an opportunity to try out techniques in a non-threatening team before applying them in our own work environments."

6. In Retrospect: Revisiting the Basic Premises behind the Software Process Practicum

So how well did our basic premises serve us? How do they still affect us after two offerings of the Practicum and as we prepare for our third?

- Use the "adult learning paradigm" to make the learning experience and the materials relevant.

  In retrospect: This continues to prove to be one of the most important decisions we made. By providing a combination of the traditional "lecture/homework" and experiential learning environments, we have been able to ensure the key lessons "stick." We have provided the students with a safe environment to practice new knowledge and skills - this is the class environment. And our students then tell us that they are able to use their new-found knowledge in the workplace with increased confidence. Another way of reinforcing this was to invite our 1994 students to the last class of the 1995 term. There, they shared their key learnings and described their own "real-world" experiences of how the information presented in Practicum helped them to achieve personal, team, project, and organizational goals.

- Ensure the topics are applied and reinforce each other.

  In retrospect: There is an oft-cited guideline in marketing - it takes five to six contacts for a new message to be adopted. We recognize and exploit this in the Practicum. We introduce some basic concepts and skills early (e.g., meeting management, social styles, effective teaming) and discuss them in many different contexts throughout class, linking and reinforcing their importance. We also ask the students to use them and write about them in several homework assignments through out the term. This, too, has contributed to the student successes in using this information in the "real world."

- Teach the "soft" stuff, the social processes, as well as the "hard" stuff, the quality processes: emphasize the inherent interdependency of the two.

  In retrospect: This has proven to be the biggest strength of the Practicum, the area the students most often cite as leading to their most significant growth. Most of our students are well versed in technical concepts, models, knowledge, and skills. They are

most adept in the Innovator role of leadership.[10] By focusing on the social processes as well, the Practicum introduces the students to many of the key skills needed by the Motivator role (and, to a lesser degree, the Organizer role) and provides the opportunity to practice it.

- Balance education (concepts, long-term focus) and training (skills, short-term focus) needs.

  In retrospect: We discussed this throughout our first two offerings of Practicum, and continue to do so, mostly as we do a post mortem from the previous offering and again when we plan for the next offering. In fact, we find these discussions cover much more than just education vs training; they cover many of the issues crucial to building a successful academic-industrial collaboration. We recognized this early, and we remind ourselves of it continuously. We are both strongly committed to supporting the students, the department faculty, and OGI goals of meeting the needs of the industry in the local community; we have been able to build an effective partnership to meet those goals.

- Build on a variety of materials.

  In retrospect: This is a continuing challenge - how to cover all the "interesting" material without overloading the students?! The other challenge is a truly administrative one - getting clean originals, getting the copyright permissions, and getting copies into the students' hands, and so forth. We can (and do) plan ahead for the "classical" writings. The more current and "breaking" information causes us more difficulty, both on an on-going basis during the term and at the beginning of each term as we put the "course pack" (readings) together. While preparing for the 1996 fall term offering of Practicum, we invested substantial time and effort into collecting and reproducing all the readings ahead of time; we believe this will minimize the overall effort in future years, as we just "swap out" some of the writings for more current information.

- Use information from the *Report of Findings* to select or validate what and how we are teaching.

  In retrospect: When the *Report of Findings* was published in 1994, we were not surprised at the skills and competencies that were identified as leading to success in industry. Even though some of the specifics have changed over the past three years, the basics remain constant. Because we built the Practicum around the more basic, longer-lasting, conceptual items (e.g., process, teamwork, learn-by-doing), the Practicum is still meeting the needs of the industry community we are serving.

- Ensure the students have fun while learning, and that we, too, both have fun and learn.

  In retrospect: We keep looking for humorous items to integrate in our class. We have found additional ways as well, things that some years ago we would probably never have considered doing - such as having "toys" available to the students during many class sessions. As we have set the tone for allowing fun in class, our students contribute in unique ways. They sometimes bring in relevant comic strips or "real life humor" from their work. One student, in the fall 1995 term, brought in an exercise to define a process to make a peanut butter and jelly sandwich. Initially, we wove this into our last session on process definition. In the fall 1996 term, it will become a regular part of our class.[12]

---

[10]See *Becoming a Technical Leader*, Gerald M Weinberg, Dorset House Publishing, for a discussion of the MOI leadership model - Motivator, Organizer, Innovator.

7. Future Directions

We have begun a periodic follow-up this year, open to all those who have completed the Practicum. While we do not have much data to report yet, we find the former students still talking about the value of and using much of the information they learned in the Practicum. In this next series of sessions, we will be exploring more of the leadership aspects that are key to ensuring any type of quality effort has "staying power."

In any case, we will continue to host periodic follow-up gatherings. We have several purposes for this:

- We will obtain an "N-months after" perspective from our past students. This will be very useful, as we will factor that information into our preparation of course modules for the next offering.
- We will try to understand progress to date in some of the longer-term improvement efforts. This will feed into our course material and presentation improvement process (above) and enable the students to leverage the experiences of each other (below).
- We will provide an opportunity for the students to continue learning and supporting each other, growing a network of quality and process knowledgeable resources within the local software community. This will build on one of the key premises behind the Practicum - that adults bring much to the learning experience, and can learn from each other.

We are teaching this course again in the fall 1996 term. That gives us sufficient time to execute, once again, our own process improvement cycle. And we continue to be in touch with local industry groups (e.g., Software Association of Oregon) to describe this educational opportunity.

The improvement efforts that our students began are continuing - albeit slower than anticipated. They are making progress; they know we are available and able to assist them; and they have contacted us for assistance and have used our materials. We believe strongly that it is not the classroom performance that will prove the benefit of our course, but rather how that information is taken back into the workplace and applied successfully. Our continuing follow-up will provide us - and the students - with more information.

In the meantime, we have been pursuing additional liaisons with local industry. Through the Oregon Graduate Institute Computer Science and Engineering Department and the Continuing Education organization, we initially targeted the four companies that participated in creating the software competency model - Intel, Mentor Graphics, Sequent, and Tektronix. Our goal is to improve collaboration between industry and academia. Industry has stated its needs; it is now academia's opportunity to put a program together - like we did for the Software Process Practicum. And then it is back to industry to provide the opportunities and encouragement to its employees to participate in the educational opportunities that have been created.

The OGI Continuing Education organization offers a number of short courses to organizations of all sizes, often at their site. We are exploring several alternatives to making all or portions of the Software Process Practicum available, on-demand, to local industry. This activity clearly supports the goal of building an effective, highly productive, extremely competitive workforce in this area. Many of these short courses are also being presented as "workshops" at national and international conferences.

Our conclusions are straightforward:

- We have demonstrated that it is possible to have a working, effective, collaborative partnership between industry and academia wherein everybody wins - especially the students.

- We have demonstrated that via this collaboration, academia (the Oregon Graduate Institute) can respond to industry needs effectively, once those needs are articulated (as they are in the *Report of Findings: Joint Software Engineering Needs Analysis*).
- Our students tell us that they are using what we taught them, that it is working for them, and that they are making improvements and changes in their personal processes, their projects, their organizations.
- We have demonstrated that students can be taught software quality concepts, knowledge, and skills in a way that enables them to begin to use it immediately and to demonstrate objective and quantitative results.
- We believe as strongly as ever that the "soft" stuff is just as important as the "hard" stuff. In fact, it is the interpersonal and team techniques and concepts that underlie and enable effective process definition and improvement techniques.
- Learning by doing – again – and again – and again – is a very powerful teaching technique. The academic environment provides a "safe haven" to experiment with new concepts and methods. By tying our learning-by-doing assignments to real-world issues, we have provided a safe and relevant environment that fosters effective learning.

8. Some Closing Thoughts

The Software Process Practicum is the winner of the 1996 Pacific Northwest Software Quality Conference's award for Software Quality Excellence. This process has been ardently supported by the Computer Science Department Head at the Oregon Graduate Institute and by our students. We continue to learn from our students many exciting ways they have used the materials and opportunities provided in the Practicum to "make a difference." Specifically, we heard how the students have truly changed some of their own professional lives and how they have had positive impact on the quality and timeliness of the products they are involved in producing.

What we did "isn't rocket science"; we have demonstrated that process improvement concepts and practices can be taught to a mixed group of students in a for-credit, graduate class. Thus, the software engineers of the next century are being exposed to state-of-the-art and state-of-the-practice process improvement literature, tools, and methods. In addition, by bringing such a course into the academic setting, into a class of students both from academia and industry, by working hands-on projects with both academia and industry, we are definitely broadening the perspective of where process improvement is applicable.

9. Acknowledgements

We appreciate the efforts of the State of Oregon and the team from Intel, Mentor Graphics, Sequent, and Tektronix for their efforts in creating an outstanding model that proved very useful to us. Thank you to the support staff of OGI who went beyond the call of duty to support this initial teaching of the Software Process Practicum, notably Phyllis Raymore and Kerri Burke (in 1994) and Kelly Atkinson (in 1995) and Jeff West (in 1995 and already in 1996). Most important thanks go to our students, who took the material we offered and made brilliant use of it in their own work environments.

From 1994: Jim Bindas (Intel), Debbie Blanchard (Consolidated Freightways, now with Con-Way Transportation Services), Michael P Gerlek (OGI, now with Intel), Gary Hanson (ADC Kentrox), Allyn Jackson (ADP Dealer Services, now with NEC), Celeste Johnson (Mentor Graphics), Alexei Kotov (OGI), Christos Mandalides (OGI), Thom Parker (Intel, now back in school), and William Trost (OGI, now an independent consultant)

From 1995: Tito Autrey (OGI), Jef Bell (OGI), Kathy Fieldstad (now with Revision Labs Inc), Priyadarshan Kolte (now with Motorola, Austin), Eldon Metz (now with Rogue Wave), Joe Mueller (Mentor Graphics), Sudarshan (Sun) Murthy (Tiger Systems Inc), Ian Savage (CFI ProServices), Ingrid Sutton (now with Intel), Jim Teisher (Credence), Doug Vorwaller (Wacker Siltronic Co), Tanya Widen (OGI), Chad Willwerth (Intel)

Our course, and this paper, reflects many of their suggestions for improvement. Our appreciation also to those partner companies who provided a real-world laboratory for our students, and who benefited from the results.

Appendix A. Course Logistics

We are often asked for a description of the course logistics; this Appendix provides a summary of that information.

Ten students enrolled in the fall 1994 offering of the Software Process Practicum, an acceptable enrollment for a first offering of a course like this, based on past history; there were 13 in fall 1995. Two of the 1995 students came as direct referrals by a 1994 student. Much to our surprise, this was from a research lab at OGI as opposed to being from industry.

In both 1994 and 1995 offerings, the industry students were from various sized companies, and had a variety of job responsibilities, including software engineering, customer support, process, and training. Of the 1994 students, very few students had any prior software engineering education beyond unstructured "on the job" training. On the other hand, of the 1995 students, a larger percent had received some software engineering education in school and/or at their companies.

Attribute	*1994*	*1995*
Total Enrollment	10	13
Full-Time OGI Students	4	3
Full-Time Employed (vs student)	6	10
Pursuing Masters or Doctorate Degree	6	4

In 1994, the class was scheduled to meet twice each week for one and one-half hours, 5:30pm - 7:00pm; several classes ran over by up to 15 minutes. The class met for 10 weeks, with one all-day, Saturday lab. Class time was mostly lecture, along with significant time for discussion and hands-on classroom activities. This same basic structure was retained in 1995; however, in response to student suggestions and a reality check on our part, we increased the number of credits from three (in 1994) to four (in 1995), and ran class times from 5:30pm - 7:30pm, with almost no classes running over.

Readings were assigned each night; individual homework assignments were assigned intermittently; there were two major team homework assignments. As soon as the students indicated the load was just too much, we reduced the amount of reading and eliminated many of the remaining individual homework assignments. We faced this issue both years, even though we reduced the required readings for the fall 1995 offering.

In the fall 1995 term, we also began asking the students to keep metrics on the time spent on out-of-class activities. As this was our first time collecting these numbers in a defined way, we did not use them very actively in 1995. However, we got enough information to validate that, on the average, many students were within the bounds of a four-credit, graduate course at OGI

(about two hours of outside work for each hour of contact), with notable peaks and valleys. This is one area we are addressing more proactively in the fall 1996 term.

The two centerpiece homework assignments were to be done in teams and with the participation of a partner organization. These were to work with a partner organization to define one of their real-world process (mid-term homework) and to work with them to create a process improvement plan for that process (end-term homework). Both these assignments required significant effort on the part of the students - including identification and enlistment of the partner organization, interviewing and working with its staff, and providing information back to various levels of the organization. In addition, the students used meeting management, interpersonal, communication, and team skills, and they gave presentations of these two major homework assignments.

As it happened, the teams basically stayed constant throughout both terms; one of the 1995 teams lost some members and had to address those team issues as well. For some of the in-class activities, we specifically selected other team arrangements to ensure the students had an opportunity to work with a variety of people.

The Saturday lab was primarily a hands-on formal inspection workshop, concluding with the presentation of the process definition projects. The students presented their process improvement projects at the penultimate class session. The final session was a review and synthesis session in preparation for the final project.

The students had a final project to complete, to be done individually although peer review was allowed. This project was to produce a process improvement plan for a process in which the student was involved - a personal process or a larger, organization-wide process. We proposed a cap of eight hours to be spent on this final project; some students chose to spend more time than this because they intended to use the results directly in a team environment at work.

Grades were computed as follows, with about half the grade based on individual performance and half the grade based on team performance:

- Homework - individual/team 20%
- Mid-term homework - Process Definition - team 20%
- Mid-term homework - Process Improvement - team 30%
- Final project - individual 30%
- Class participation (individual) subjective

Appendix B. Bibliography and Syllabus

The Bibliography and Syllabus for the fall 1996 Software Process Practicum is attached.

Course Number	CSE 503
Course Title	Software Process Practicum: Lessons in Software Quality and Leadership
Instructors	Judy Bamberger and James Hook
Days	Mondays, Wednesdays (and one Saturday)
Times	5:00pm - 7:00pm
Room	Cooley Center (CC) 371
Number of Units	4 credits

The software process practicum is designed to immerse the working student in topics relevant to software process improvement and quality management, and to introduce them to the supporting theory. Topics include process management frameworks (capability maturity model, ISO 9000), measurement for process improvement, and key team skills necessary for effective collaborative software engineering efforts. At the end of the course the student will be able to demonstrate that the software development process can be managed and controlled, leading to increased software quality. In addition to lectures and in-class "labs," the class will include one Saturday workshop.

OBJECTIVES / VISION

After this class, the students will understand and have demonstrated that:

- Software processes can be managed and controlled.
- They understand that software engineering is a social process, too.
- They have real skills they can apply today at work.
- They have a framework on which to build their own educated decisions about applying software quality principles and tools to personal, project, and corporate software activities.
- They have identified three things to improve at their own work place (or within their own personal process) and are working on them.

To the Students:

This is a list of required and recommended books.

Readings will be derived from the three required books throughout the semester. We will be discussing some of them as part of the class session. They present a unique view of many of the concepts, models, and skills we will be covering - often a different view than would be found in most computer science courses.

The recommended books will provide additional breadth and assistance throughout the class.

We have selected these books because we believe they will be useful to you after this course, in your work environment and in your professional activities. Your feedback throughout the course and afterward will be appreciated.

Required Books

(1) Grady, Robert B and Caswell, Deborah L, *Software Metrics: Establishing a Company-Wide Program*, P T R Prentice Hall, 1986

(2) Scholtes, Peter R et al, *The TEAM Handbook*, Joiner, 1998

(3) Weinberg, Gerald M, *Quality Software Management, Volume 1, Systems Thinking*, Dorset House Publishing, 1993

Recommended Books

(1) Brassard, Michael, *The Memory Jogger Plus+*, GOAL/QPC, 1989

(2) Weinberg, Gerald M, *Quality Software Management, Volume 2, First-Order Measurement*, Dorset House Publishing, 1993

ORGANIZING FOR SUCCESS

Things to think about from the beginning:

- Who would you like on your team? Teams are not required to keep the same members throughout the entire course. However, especially once the mid-term project is begun, this does have significant advantages. We suggest you begin now, and think about how you could build an excellent, effective, and high-performing team (we will be giving you some hints, too). We also suggest that you use your first team project to try some of those ideas, and set the tone for success.
- With which "partner organization" would you like to work? The mid-term and end-term assignments will focus on working with what we call a "partner organization." This could be a team with which you work at your company or school (highly preferred), or it could be a team we recommend to you. You will be collaboring with them to define a process and then to create a plan to improve that process. This will involve some of their time - in the past, it has been about 2 - 12 hours total over the entire term (depends on number of people involved, and depth of their involvement).

GRADING CRITERIA

There will be individual and team assignments. We have tried to give most assignments on a Monday, with the turn-in date generally on the following Monday. Team assignments will be given a single grade, which will be assigned to each team member. Team assignments will have an individual component associated with each (graded individually) to analyze team effectiveness overall, and the individual's effectiveness within that team.

> The goal of these assignments is to allow you to reinforce the concepts and skills learned in one or more Practicum sessions.

There will be a mid-term and end-term project to be done as a team. The mid-term and end-term projects are related (general descriptions are included in the syllabus). The mid-term project focuses on working with a partner organization to define a software-related process using the techniques we learn in Practicum. The end-term project focuses on working with that partner organization to identify and plan for improvements to that process.

> The goal of these projects is to allow you to synthesize the concepts and skills learned in several Practicum sessions and practice them in a real-world setting. Past projects have also resulted in significant benefits to the partner organization as well, a secondary goal.

There will be a final project to be done individually. This project will be to create an improvement plan for a process in which you are involved personally - individually or as part of a team at work or outside of work.

> The goal of this project is to allow you to synthesize the information learned in Practicum and apply it in a real-world, relevant context.

There are two un-graded elements as well.

> The goal of both of these are to help us continuously improve the Practicum - both for you, this term, and for future offerings of the Practicum.

We will be asking you to keep a Timelog - the amount of time you spend preparing for each class (e.g., reading) and doing the homework assignments. This will have absolutely no bearing on any grade. In fact, we will not look at it until any related assignments have been graded. We will use this to help us assess and tune the overall workload, week-by-week, assignment-by-assignment. A Timelog template (with instructions) is included in this syllabus.

> Please turn in your Timelog sheets each class session.

We will also be asking you to keep a Journal - short notes about the readings and learnings. We will ask to see this three times during the term. We will use this to help us assess the impact of the readings and the messages you take from the classes. Again, this will have absolutely no bearing on any grade, and we will not look at it until related work has been graded. We will use this to help us identify "what works" and "what doesn't," as well as those articles, class sessions, exercises, etc that have the most/least impact. A set of suggested items to cover in your Journal entries is included in this syllabus.

> Please try to make your Journal entries each day, or as new learnings come to you.

• Homework - individual/team	20%
• Mid-term project - team Process Definition	20%
• End-term - team Process Improvement	30%
• Final project	30%
• Class participation (individual)	subjective
• Timelog	0%
• Journal	0%

We will make every effort to return homework to you within one calendar week. You will see comments from us and the following notations in the upper right corner, with the following meanings:

Φ	"not"; no grade noted; perhaps our instructions were not clear; goals of the assignment were missed; please see us and let's get straightened out; OK to rework and resubmit for success
–	"minus"; does not meet minimal criteria; OK to rework and resubmit for success
√	"check"; meets goals of homework assignment
+	"plus"; exceeds goals of homework assignment
++	"double-plus"; exceeds our wildest dreams !!!

OFFICE HOURS and CONTACT INFORMATION

- Judy Bamberger: Monday, 7:00pm - 8:00pm at OGI and by appointment
- Jim Hook: Wednesday, 7:00pm - 8:00pm at OGI and by appointment

Judy Bamberger	Jim Hook
690-1481	690-1169
bamberg@cse.ogi.edu	hook@cse.ogi.edu
room *TBD*	room CSE 143

We are here to ensure you get the most out of this class, so please come and talk to us when you need to! If you use Email to communicate or ask questions, then please send it to both of us.

BIBLIOGRAPHY AND SYLLABUS

In the following:

(R#) means there is reading to be done before this class

In general, the reading is intended to be done to a level where you are confident you can discuss the key themes (maybe not all the details), as we will begin many of the classes with a discussion of the readings. The goal of assigning the readings is to broaden your background, and to provide you with "intellectual pointers" to key references for the class (short term) and for your future as a professional (longer term).

Those few times where detailed understanding of the readings is required (e.g., to prepare for a specific class discussion or a homework assignment), we will indicate that explicitly. Whenever in doubt, one way or another, please ask.

(H#) means there is homework to prepare after this class

*(H#) The asterisk (*) before the homework indicates there is an individually-done assignment to be turned in

&(H#) The ampersand (&) before the homework indicates there is a team-done assignment to be turned in (i.e., one single assignment per team, with all team member names on it)

(H#) The lack of any leading marking before the homework indicates this is reading, other material, or other activities related to completing some other homework assignment to be turned in

(JOUR) means there is a journal-related activity here

The • -ed and – -ed lists summarize the topics to be covered in this class

When you turn in your homework, please make sure the following are done:

- Please put your name clearly on the front page, and indicate which homework this is (at least the H-number you see in the syllabus and on any relevant handouts)
- Please put page numbers on each page (hand-written is OK)
- Please type (vs hand-write) your assignments (with many papers to read, we find it difficult and slow with a lot of hand-written papers)
  - Do not spend a lot of time on formatting; just leave us enough space to write some comments and ask some questions
- Please run some sort of spell-checker on your papers and make corrections (some of the typographical errors significantly decrease our ability to "figure out" what you are trying to say)

Handouts, Articles, Readings, and Other Materials Passed to Students

(1) Introduction, Ice Breaking, and Motivation

Week: 1	Class: 1	Day/Date: Monday, 30 September	Key Presenter: Jim and Judy	Readings? no Homework? yes

- Copy of class materials
  - Develop common expectations about the class
  - Communicate class mechanics
  - Briefly survey "quality"
  - Discuss quality in the software context by introducing an example

*(H1a) Write your "process biography" (per instructor-provided questionnaire, handed out in class)
[turn in at class #2]

*(H1b) Interview three software development organizations (per instructor-provided questionnaire, handed out in class)
[turn in at class #3]

(JOUR) Write a "learning contract" for yourself - what do you want to learn; how do you want to learn it; how, when, where, with whom do you want to practice it; how will you claim "success" for yourself after Practicum is over; make this your first Journal entry

(2) Background: Statistical Process Control; Several Basic Quality Tools

Week: 1	Class: 2	Day/Date: Wednesday, 2 October	Key Presenter: Jim and Judy	Readings? yes Homework? yes (thinking only)

(R2a) *Deming Management at Work*; Mary Walton; Chapter 2, "Florida Power & Light"

(R2b) *Quality Planning and Analysis - From Product Development through Use*; J M Juran and Frank M Gryna; Chapter 1, "Basic Concepts" and Chapter 2, "Companywide Assessment of Quality"

(R2c) *Quality is Free*; Philip B Crosby; Chapter 2, "Quality May Not Be What You Think It Is," Chapter 3, "The Quality Management Maturity Grid," and the Browser's Guide

(R2txt) *Quality Software Management, Volume 1, Systems Thinking*; Gerald M Weinberg; Chapter 1, "What Is Quality? Why Is It Important?"

- Copy of class materials
  - Introduce several quality tools (brainstorming, consensus, affinity diagram, Pareto diagram)
  - Discuss classical Statistical Process Control (SPC)
  - Point to other quality and management and planning tools (flowchart, check sheet, run chart, historgram, scatter diagram; interaction digraph, tree diagram, prioritization matrices, matrix diagram, process decision program chart, activity network diagram)

(H2a) Begin thinking about how you will build your team for class projects

(3) How does the "I" fit into "TEAM"?

Week: 2	**Class: 3**	**Day/Date:** **Monday, 7 October**	**Key Presenter: Judy**	**Readings? yes** **Homework? yes**

(R3a) *Social Style Profile - Feedback Booklet*; Wilson Learning

(R3b) *Enterprise*, Winter 1991/92; "Unleashing People Power - Innovation comes from the Individual at Chaparral Steel"

(R3c) *Training & Development Journal*, April 1991; Richard Wellins and Jill George; "The Key to Self-Directed Teams"

(R3txt) *The Team Handbook*, Chapter 4, "Getting Underway"

[optional] *The Team Handbook*, Chapter 6, "Learning to Work Together"

- Copy of class materials
  - Meeting management techniques
  - Different style preferences
  - Team development and growth
- Handout - "Teams Need a Common Goal" (Hagar the Horrible cartoon)
- Handout - "Why Some Teams Don't Fail" (from *Manage*, July 1993)

*(H3a) Analyze the strengths and weaknesses of each of the social styles (per instructor-provided scenario, handed out in class)
[turn in at class #5]

(H3b) [readings to be handed out in class] *Please Understand Me - Character & Temperament Types*; David Keirsey and Marilyn Bates; Chapter 1, "Different Drums and Different Drummers" and Appendix: The Sixteen Types

(H3c) [reading to be handed out in class] "MBTI Short Summary", Judy Bamberger

*(H3d) Do the readings above; see what the instrument indicates as natural tendencies, and discuss (3-5 pages) how these characteristics manifest themselves in your team interactions at work
[turn in at class #5]

(4) Problem Solving Paradigms

Week: 2	Class: 4	Day/Date: Wednesday, 9 October	Key Presenter: Judy	Readings? yes Homework? yes

(R4a) [optional] *Problem-Solving Process*; Xerox, 1992; "Participant Guide"

(R4txt) *Quality Software Management, Volume 1, Systems Thinking*; Gerald M Weinberg; Chapter 2, "Software Subcultures" and Chapter 3, "What is Needed to Change Patterns?"

- Copy of class materials
  - One problem solving model (useful tools at each phase, decision making styles and tools)
  - One conflict resolution model (identifying, managing, and resolving conflict)
- Handout - Role Play: Veginots (if used in class)

&(H4a) As a team, using the basic tools, discuss how you would solve the problem of whether or not to inform the customer of a potential schedule slippage (per instructor-provided guidelines, handed out in class)
[turn in at class #6]

(5) Process Definition Techniques

Week: 3	Class: 5	Day/Date: Monday, 14 October	Key Presenter: Judy	Readings? yes Homework? optional

(R5a) *Managing the Software Process*; Watts Humphrey; Preface, Part One, "Software Process Maturity", Chapter 2, "The Principles of Software Process Change", Chapter 14, "The Software Engineering Process Group", Chapter 20, "Conclusion"

(R5b) *IBM Systems Journal*, 1985; R A Radice *et al*; "A Programming Process Architecture"

- Copy of class materials
  - Several techniques for representing defined processes
- Handout - Process Definition Examples
- Handout - "Rules of thumb for developing processes - Lessons learned from the trenches"; Mary Sakry

*(H5a) [optional] Demonstrate the process representation and definition techniques on the sample process description we provide (handed out in class)
[turn in at class #7]

Mid-Term Homework

- Define a process with your partner organization (per instructor-provided guidelines; draft attached, handed out in class)
[turn in at class #10; make presentation at class #11]

(6) Continuation of Topic from class #5

Week: 3	**Class: 6**	**Day/Date:** **Wednesday,** **16 October**	**Key Presenter: Judy**	**Readings? yes** **Homework? nothing new**

Please do the following readings very carefully. The key idea to learn is the method Weinberg uses to represent and analyze processes (the drawings with "blobs" and annotated arrows). We will be using this in class and/or as part of a future assignment.

(R6txt) *Quality Software Management, Volume 1, Systems Thinking*; Gerald M Weinberg; Chapter 4, "Control Patterns for Management" and Chapter 5, "Making Explicit Management Models"

(7) Continuation of Topic from class #5

Week: 4	**Class: 7**	**Day/Date:** **Monday,** **21 October**	**Key Presenter: Judy**	**Readings? yes** **Homework? nothing new**

(JOUR) Please turn in your Journals.

Please do the following readings very carefully. The key idea to learn is the method Weinberg uses to represent and analyze processes (the drawings with "blobs" and annotated arrows). We will be using this in class and/or as part of a future assignment.

(R7txt) *Quality Software Management, Volume 1, Systems Thinking*; Gerald M Weinberg; Chapter 6, "Feedback Effects," Chapter 7, "Steering Software," and Chapter 8, "Failing to Steer"

- Note: If we complete planned in-class exercises early, we may do a student-provided activity or an exercise based on the Weinberg reading, or we may start the next topic
- ANNOUNCING MAJOR READING FOR class #9!!!

  Select two quality models (in class) and be prepared to present and discuss a set of comparison issues (per instructor-provided criteria); this will involve careful reading (vs detailed skimming)

(8) Capability Maturity Model for Software (CMM)

Week: 4	**Class: 8**	**Day/Date:** **Wednesday, 23 October**	**Key Presenter: Judy (Jim is out)**	**Readings? yes** **Homework? nothing new**

(R8a) *Capability Maturity Model for Software, Version 1.1*; Software Engineering Institute; Chapter 1, "The Process Maturity Framework", Chapter 3, "Operational Definition of the Capability Maturity Model", Appendix A, "Goals for Each Key Process Area"

(R8b) *IBM Systems Journal*, 1985; W S Humphrey; "The IBM large-systems software development process: Objectives and direction" and R A Radice *et al*; "A programming process study"

(R8c) *American Programmer*, September 1994; James Bach, "The Immaturity of the CMM"

--- and optional readings (helpful references) ---

(R8d) *IEEE Software*, July 1994; Michael K Daskalantonakis; "Achieving Higher SEI Levels"

- Copy of class materials
  - Basic process management concepts
  - How they apply to software engineering
  - Characterization of "immature" and "mature" software engineering processes
  - Five levels of maturity as defined by CMM
  - Details of Repeatable Level
  - Framework of CMM - applicability across many disciplines
  - Some anecdotal data on ROI

(9) Quality Frameworks: Applying the Concepts to Process Improvement

Week: 5	Class: 9	Day/Date: Monday, 28 October	Key Presenter: Judy	Readings? yes Homework? nothing new

(R9a) *International Standard, ISO 9000-3*; ISO; "Quality Management and Quality Assurance Standards - Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software"

(R9b) *Trillium*, 1994; Bell Canada; "Telecom Software Product Development Process Capability Assessment"

(R9c) *Quality System Review - Guidelines*, March 1995; Motorola Corporate; "Introduction", "QSR General Scoring Maturity Matrix", "Subsystem 10 - Software Quality Assurance" and "Scoring Reference", "Quality Policy for Software Development"

(R9d) *Malcolm Baldrige National Quality Award*, 1996

- Completion of previous material and compare/contrast all quality models

(10) Quality Technique #1 - Formal Inspections

Week: 5	Class: 10	Day/Date: Wednesday, 30 October	Key Presenter: Judy	Readings? yes Homework? yes

****** It's the day before Halloween; dress up; party!!! ******

(R10a) Neal Brenner; "The ST Inspection Handbook"

(R10b) *IBM Systems Journal*, 1976; M E Fagan; "Design and code inspections to reduce errors in program development"

(R10c) *IEEE Software*, July 1994; Robert B Grady and Tom Van Slack; "Key Lessons in Achieving Widespread Inspection Use"

(R10d) *IEEE Software*, March 1994; Jack Barnard and Art Price; "Managing Code Inspection Information"

--- and optional readings (helpful references) ---

(R10e) *IEEE Software*, September 1993; Edward F Weller; "Lessons from Three Years of Inspection Data"

(R10f) *IEEE Transactions on Software Engineering*, July 1986; Michael E Fagan; "Advances in Software Inspections"

(R10g) *Software Validation - Inspection - testing - verification - alternatives*; A F Ackerman, P J Fowler, and R G Ebenau; "Software Inspections and the Industrial Production of Software"

(R10h) *Software Inspection*; Tom Gilb and Dorothy Graham; "Software Inspections at Applicon" by Barbara Spencer

- Copy of class materials
  - One defined process for formal inspections
  - One set of metrics that can be obtained from formal inspections
  - One set of forms, guidelines, rule sheets for formal inspections
- Handout - Inspection Package
- Other handouts to support the lab will be provided as needed

(H10a) Prepare for formal inspection workshop, Saturday (use inspection lab materials, handed out in class)
[class #11]

(11) Inspection Workshop and presentation of Mid-Term Homework: Process Definition

Week: 5	**Class: 11**	**Day/Date:** **Saturday,** **2 November**	**Key Presenter: Judy and students**	**Readings? nothing new** **Homework? nothing new**

End-Term Homework

- With your partner organization, define a process improvement plan for the process you defined in the Mid-Term Homework (per instructor-provided guidelines; draft attached, handed out in class)
[turn in at class #16; presentation at class #17]

(12) Process Improvement Models

Week: 6	**Class: 12**	**Day/Date:** **Monday,** **4 November**	**Key Presenter: Judy**	**Readings? yes** **Homework? optional**

(R12a) selections from the *Quality* issue of Business Week, 1991

(R12b) [optional] *Total Quality Improvement System*; ODI; "Quality Action Teams - Team Member's Workbook"

(R12c) [optional] *Total Quality Improvement System*; ODI; "Quality Action Teams - Project Booklet"

(R12txt) *The Team Handbook*, Chapter 5, "Building an Improvement Plan"

- Copy of class materials
  - Several process improvement models and strengths of each
  - Principles of action planning and "how to"
- Handout - Organizational Climate Survey, Judy Bamberger

*(H12a) [optional] Leveraging the readings from Business Week and comparing it with your experience, discuss what appears to you to be the "top six" characteristics of high-quality organizations
[turn in at class #14]

(13) Software Metrics

Week: 6	**Class: 13**	**Day/Date:** **Wednesday, 6 November**	**Key Presenter: Judy**	**Readings? yes** **Homework? yes**

(R13a) *Software Modeling and Measurement: The Goal/ Question/Metric Paradigm*, Victor R Basili

(R13b) [optional] *Software Quality*, "Software Metrics that Meet your Information Needs," Linda Westfall

(R13txt) *Software Metrics: Establishing a Company-Wide Program*, by Robert B Grady and Deborah L Caswell, chapters 5-6 and 12-15 (please use this as a minimum guideline; we would have like to have assigned the entire book)

- Copy of class materials
  - Why measure software processes
  - What are some things that can be measured in software processes
  - Goal, Question, Metric Paradigm
  - Examples

*(H13a) Identify a problem at work and do a "detailed impact case study" and a "subjective impact study" following Weinberg, Volume 2, sections 8 (especially 8.4 - 8.6) and 9 (especially 9.3 - 9.5) (handed out in class)
[turn in at class #15]

Week: 7	**NO CLASS**	**Day/Date:** **Monday,** **11 November**		**Readings? nothing new** **Homework? nothing new**

Week: 7	**NO CLASS**	**Day/Date:** **Wednesday,** **13 November**		**Readings? nothing new** **Homework? nothing new**

(14) Organizational Infrastructure for Sustained Process Improvement - Leadership and Technology Transition

Week: 8	**Class: 14**	**Day/Date:** **Monday,** **18 November**	**Key Presenter: Judy**	**Readings? yes** **Homework? nothing new**

(JOUR) Please turn in your Journals.

(R14a) *Quality Planning and Analysis - From Product Development through Use*; J M Juran and Frank M Gryna; Chapter 7, "Organization for Quality" and Chapter 8, "Developing a Quality Culture"

(R14b) *The Leadership Challenge*; James Kouzes and Barry Posner; Part One, "Knowing What Leadership Is Really About", Chapter 1, "When Leaders Are at Their Best: Five Practices and Ten Commitments", Chapter 2, "What Followers Expect of Their Leaders: Knowing the Other Half of the Story", Part 7, "The Beginning of Leadership", Chapter 13, "Become a Leader Who Cares and Makes a Difference"

(R14c) "Leading Change: Why Transformation Efforts Fail"; *Harvard Business Review*; March/April 1995

(R14txt) *Quality Software Management, Volume 1, Systems Thinking*; Gerald M Weinberg; Chapter 18, "What We've Managed to Accomplish"

- Copy of class materials
  - Examined some management/leadership issues to sustain process improvement
  - Discussed key points of effective leadership
  - Understand your role as leader for process improvement
  - Getting information out about improved process
  - Getting improved process adopted and used

(15) Quality Technique #2 - Quality Function Deployment (QFD)

Week: 8	Class: 15	Day/Date: Wednesday, 20 November	Key Presenter: Judy	Readings? yes Homework? nothing new

(R15a) *Harvard Business Review*, May-June 1988; John Hauser and Don Clausing; "The House of Quality"

(R15b) "QFD for Software - Satisfying Customers"; Richard Zultner

- Copy of class materials
  - Increasing importance of focus on quality
  - Voice of the customer
  - Exercise using QFD
- Handout - Leemak, Inc; "Problems That QFD Solves"
- Handout - Leemak, Inc; "OK, So How Long Does It Really Take?"

(16) Managing Change

Week: 9	Class: 16	Day/Date: Monday, 25 November	Key Presenter: Judy	Readings? yes Homework? nothing new

(R16a) *IEEE Software*, January 1990; Barbara M Bouldin, "The nature of change agents"

(R16b) *Harvard Business Review*, January-February 1992; Robert H Schaffer and Harvey A Thomson, "Successful Change Programs Begin with Results"

(R16c) *Group and Organization Studies*, SAGE Publiciations, Group and Organizational Studies, December 1982; J Scott Armstrong, "Strategies for Implementing Change: An Experiential Approach" (Delta process)

- Copy of class materials
  - People issues about key principles of effective change

(17) Team Presents: End-Term Homework

Week: 9	Class: 17	Day/Date: Wednesday, 27 November	Key Presenter: students	Readings? yes Homework? nothing new

(R17a) *Teaching the Elephant to Dance*; James Belasco; Chapter 1, "Teaching the Elephant to Dance - The Manager's Guide to Empowering Change", Chapter 2, "Getting Ready to Change", Chapter 11, "Empower Individual Change Agents", Chapter 12, "Change Happens - The Elephant Learns"

(R17b) Article from Wall Street Journal, 13 September 1994 on Chinese Quality Managers

(18) Surprise !!!

Week: 10	**Class: 18**	**Day/Date:** **Monday,** **2 December**	**Key Presenter: Jim, entire class (Judy is out)**	**Readings? nothing new** **Homework? nothing new**

(19) Review and Summary

Week: 10	**Class: 19**	**Day/Date:** **Monday,** **4 December**	**Key Presenter: discussion (Judy is out)**	**Readings? nothing new** **Homework? nothing new**

(JOUR) Please turn in your Journals.

Final Project

- As an individual, define a process improvement plan for a personal, project, or organizational process in which you have a stake (per instructor-provided guidelines, handed out in class)
  [turn in to Jim Hook or Judy Bamberger at OGI no later than Thursday, 12 December 1996]

Journal Entry Ideas

To the Students:

The readings we suggest are intended to meet the goal of providing growth and broadening, rather than something we will "test" in class. To help us evaluate the effectiveness of the readings in meeting this goal, we would like you to use a "journal" to capture your reactions to the readings.

This is also a perfect opportunity to capture some of your thoughts about the class sessions themselves. While we do a course evaluation at the end of the term, and while we ask for input at various places throughout the term, we have heard from our students that they would find it easier to capture evaluation thoughts as the class progresses. You can use that information when it is time to do the final course evaluation.

This is your journal to keep - any way you want - hand scribbled notes and diagrams, typed and indexed on a computer - whatever. We will ask to see it three times throughout the course (approximately week 4, week 7, and week 10) so we can assess where we are going with the course. If you would rather we did not see your Journal, as it may contain private insights, we request to meet briefly with you to assess the impact of the readings and class sessions.

No grades will be given. No comments will be made, unless you explicitly request us to do so. We will use your input to make mid-course corrections and improvements in the Practicum.

Some topics you might want to consider:

- How does a reading, homework, or class session help you meet your learning contract?
- What key points did you get out of the reading, homework, or class?
- How does this synthesize with previous readings, homeworks, and classes - what new "ahas" came?
- What new knowledge did you gain from the reading, homework, or class?
- What new puzzles are opening up for you - areas where you want to experiment or get more knowledge?
- ... anything that strikes you as important at the time you write.

TIMELOG TEMPLATE

To the Students:

We would like to collect information on the time each student spent on each class preparation and assignment. We will use this ***ONLY*** to help us evaluate and tailor the workload we are asking from each of you. The preferred format to give us time spent is:

hours : minutes

with a granularity of 15 minutes.

We will ***NOT*** use time spent as a marking criterion in any way. It will be maintained in a separate database, and not examined until after all marks are given for that assignment.

Our goal is to have an "achievable class" - with some planning on your part and our part. Your providing us with this information can help us determine if we are meeting that goal. Thank you.

General Information

Student Name	
Class #	

Readings

Reading #s					
Time spent					

Homework

Homework #s					
Time spent					

Projects (Mid-Term, End-Term, Final)

Project ID					
Time spent					

Other

Description					
Time spent					

Complete only if turning in homework today

Homework ID					
Due Date					
Turn-In Date					

Mid-Term Project: Process Definition

To the Students:

This exercise is to be done as a team. Once again, "team" is defined as about three people. Effective meeting management and team collaboration skills are key to success of this exercise.

This exercise builds on classes #5, #6, and #7 (process definition) and additional readings and information about process definition. It is due, in writing, at class #10, Wednesday, 30 October 1996. Each team will present its results at class #11, Saturday, 2 November 1996, in the afternoon.

The assignment is to produce a process definition, using the techniques and representations we will be sharing with you this week, or others you may have used or know. You are not constrained to follow the process we will be teaching; however, we would like to see certain products in certain formats ... all of which can be produced multiple ways.

To do this exercise, you will need to identify two things: (1) an organization with which to work (perhaps, your own); and (2) a process to define with that organization. If you have any problems with this, see Jim or Judy; we will try to find an organization and a process for your team.

If all contacts fail, Jim said that he has a few processes within PacSoft that could be defined and improved.

- Pick a simple, bounded process within an organization - it has to involve multiple roles (disciplines, groups).

  The two examples given below indicate the "level" to which you need to go (not very deep) and the "breadth" across the organization you need to cover (multiple organizations). (Note that these are written like "scenarios" not like a real process definition; that will become clear through the lectures and readings.)

  For example, managing a requirements (specification) change - customer calls, marketing fields the call and turns it into a requirement which is passed to engineering; they analyze it and feed impact back to their management and to marketing; ... design / code ... the independent test organization tests the code; defects are fed back into software engineering ... the ready-to-ship code is configured and handed off to the release group to cut the CDs; the CDs are passed to shipping which will pull the correct documentation, package all the contents, and ship the product to the customer.

  As another example, a customer calls the service organization with an urgent problem; the service organization verifies it is a critical software defect, files a defect report, and passes the issue to software engineering, requesting a fastpatch; software engineering evaluates the defect, the impact, and the resources needed to make the fix; software engineering management allocates resources to fix the defect; the defect is fixed and a fastpatch is created; peer reviews are held to verify it; service tests the fastpatch to verify it; the

fastpatch is baselined (captured in a configuration management system); the defect report is updated to indicate a fastpatch was created and that it must be fixed in the next regular software release; a tape is cut, logged in the fastpatch tracking system, and shipped to the customer.

Your team is to turn in the following information at class #10:

(1) A process definition in all of the following representations:

- Value-added map ("context diagram")
- Time-x-role map
- EITVOX (only three process steps required to be done EITVOX; you may do as many as you want)

(2) A discussion of the team effectiveness, as described, to include:

- Team strengths, weaknesses
- Individual styles and how they contributed to the solution
- How decisions were made, how conflict was handled
- If you changed team members, discuss the impact
- Improvements you can make as a team, as individuals, for future team exercises
- Observations on overall team effectiveness
- How effectively you used your "together" time (e.g., meeting management)
- An indication of where you believe your team is on the Team Growth Model

You may also turn in any other "team stuff" you think would help us understand how you operated.

Your team is to present the following information at class #11:

- How your team did the homework; the team process and organization you used - the who, what, when, where, how, why of your team process (not effectiveness; that comes later!)

  [quick summary; be brief; ensure team consensus]

- What your team produced; a summary of the process your team defined - examples of all three representations (a summary of (1) above)

  [not expected to be a full presentation, simply a visual presentation to the class of what you produced for your partner organization and what you turned into the instructors]

- Observations on the reaction of your partner organization - to the process of defining processes, to the resultant process definition, to you as a team, etc

  [reflection, observations; ensure team consensus]

- Discussion of team effectiveness (a summary of (2) above)

  [short presentation to the class of what you produced as part of the assignment turned into the instructors]

- A brief summary (team or individual members of the team) as to the utility of process definition techniques where you work (or have in the past, or would like to in the future).

  [briefly reflect on the techniques, the processes, the experiences, and your/team/organization's reactions]
- Be prepared to answer questions of clarification, curiousity, and envy from other students and the instructors on the above.

End-Term Project: Process Improvement

To the Students:

This exercise is to be done as a team. Once again, "team" is defined as about three people. Effective meeting management and team collaboration skills are key to success of this exercise.

This exercise builds on mid-term process definition homework, class #12 (process improvement models) and other lectures, readings, and information about process improvement (in fact, the running theme throughout the course). Many of the remaining classes have no homework other than, "factor these concepts into the end-term process improvement homework"; we intend to provide information to help you address some of the questions we put to you. The homework is due, in writing, at class #16, Monday, 25 November 1996. Each team will present its results at class #17, Wednesday, 27 November 1996.

The assignment is to build on the process definition from "Week Five" homework, to work with your partner organization to:

- Identify metrics that can be used to measure the performance of the process today (to establish a baseline for improvement)
- Establish measurable goals for improving the process, and
- Develop an action plan to improve the process.

The information for the team to turn in is: the process improvement/action plan.

As individuals, you will look at individual and team dynamics and effectiveness. A list of topics to consider is included below.

The information for each individual to turn in is: a 2 - 4 page discussion of individual and team contributions.

You are not constrained to follow the models we will be teaching; however, we would like to see certain products with certain content, which is described below. We will provide additional templates for finished products (optional to use) at appropriate points in the remaining classes.

To do this exercise, you will need to reconfirm the willingness of your partner organization to continue collaboration. If you have any problems with this, see Jim or Judy; we will try to find a partner organization for your team.

You are encouraged to use the process improvement planning process and template introduced in class #12, unless your partner organization has a method and/or template of its own or you have a defined method and template of your own. If you do not use the method/template we teach, please let us know what you will be using, so we can ensure the necessary components will be covered.

Be sure to provide us with a complete action plan - all sections. The following list provides some additional hints and references to help with some of the sections.

1.1. Problem Statement

A set of metrics that can be used to measure the performance of the process today.

- Describe any metrics used today to measure the performance of the process, and how they are collected, reported, and used.
- If none, describe what metrics could be used to establish a baseline so that the organization will know on what to base its improvement. Describe how they could be collected, reported, and used.

1.2. Vision after Success and

1.3. Goal Statement

A goal (or goals) that your partner organization would like to see for process improvement. You are encouraged to validate the goals with your partner organization, if they did not participate in their creation.

- Using the Goal/Question/Metric paradigm (class #13), develop the goal, questions, and metrics. Ensure that the metrics are quantifiable (either objectively, or subjectively; see homework/reading Weinberg, Volume 2, Chapters 8 and 9). For each of five metrics, describe how it could be collected, what is its scale (or value), any known or suspected data integrity issues, how it could be used to answer the questions to test achievement of the goal(s).
- Ensure that the goal(s) is a "SMART" goal (lecture #12), or discuss why they do not need to be, or the risk incurred if they are not SMART.

11. Roll-Out/Training Plan

Consideration for "self-sustaining improvement" (aka, institutionalization factors, discussed in classes #8 and #9)

ENABLERS for sustained improvement

- Commitment to Perform (policy, leadership needed)
- Ability to Perform (tools, training, resources needed)

ENFORCERS for sustained improvement

- Measurement and Analysis (how your proposed metrics will help demonstrate achievement of the goal, how the organization will know to what degree the organization is complying with the change)
- Verifying Implementation (progress/status reports to sponsoring management, independent review, etc)

Transition considerations for the new-improved process (e.g., information dissemination, newsletters, all-hands meetings, formal training, mentoring, brown-

bag lunches, etc). This needs to reflect sensitivity to the culture of your partner organization - how they learn and retain best.

How to Get There

You are encouraged to involve your partner organization in this as much as they can be (or want to be). Since they are the ones who really practice the process being improved, they are likely to have many good ideas about how to improve it. Involvement can be: brainstorming during information/data collection; participation in the force-field analysis; reviewing intermediary products; etc.

You are encouraged to use one or more of the process improvement or problem solving methods discussed in class or offered as reading, unless there is another method with which you have much experience. (In this case, please consult with the instructors ahead of time to ensure the goals of this assignment are met.) Options include:

- The general problem solving method (and tools) from class #4 (remember to focus it on process improvement)
- The action planning techniques discussed in class #12
- The ODI method and tools from the readings for class #12 and class discussion
- The methods and tools described in *The Team Handbook*, chapter 5 (one of the readings for class #12)

Individual and Team Contributions

As individuals, please provide a discussion of ***your view*** of your contribution to your team's effectiveness throughout the course, to include:

- Your style and how it contributed to the strength of the team
- Observations on areas where you would like to improve in team activities (styles you might want to try; roles you might want to play, etc)
- Techniques and skills you used to keep the team moving forward, to resolve conflicts, to be creative, etc
- How effectively you used your "together" time (e.g., meeting management)
- An indication of where you believe your team is on the Team Growth Model
- Effectiveness of team activities in reinforcing the key themes of this course
- Lessons you have learned from your class-team that you can take into your work-place-teams

Team Presentation

Your team is to present (30-45 minutes; depends on the total number of teams) the following information at class #17:

- How your team did the homework; the team process and organization you used - the who, what, when, where, how, why of your team process

  [quick summary; be brief; ensure team consensus]

- What your team produced; a summary of the action plan

  [not expected to be a full presentation, simply a visual presentation to the class of what you produced for your partner organization and what you turned into the instructors]

- Observations on the reaction of your partner organization - to the process of improving processes, to the resultant process improvement plan, to you as a team, etc

  [reflection, observations; ensure team consensus]

- Be prepared to answer questions of clarification, curiousity, and envy from other students and the instructors on the above.

INDEX

INDEX

INDEX

1996 PROCEEDINGS ORDER FORM

PACIFIC NORTHWEST QUALITY SOFTWARE CONFERENCE

To order a copy of the 1996 proceedings, please send a check in the amount of $35.00 to:

PNSQC
PO Box 10142
Portland, OR 97296

Name _____

Affiliate _____

Mailing Address _____

City _____

State _____

Zip _____

Daytime Phone _____